Innovative **Technology**
for a **Connected** World

BL600 *smart*BASIC Module

User Manual

Release 1.2.54.0r3

global solutions: local support ™

Americas: +1-800-492-2320 Option 3
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

CONTENTS

Revision History 3
Contents 4
1. Introduction 6
Why Do We Need Another Language? 7
What Are the Reasons for Writing Applications? 7
What is in a BLE Module? 7
smart BASIC Essentials 8
Developing with *smart* BASIC 9
Operating Modes of a *smart* BASIC module 9
Types of Applications 11
Non Volatile Memory 11
Using the Module's Flash File System 12
2. Getting Started 13
What You Need 13
Connecting Things Up 13
UWTerminal 13
Writing a *smart* BASIC Application 19
3. Interactive Mode Commands 30
4 *smart* BASIC Commands 48
Syntax 48
Functions 48
Subroutines 48
Statements 49
Exceptions 49
Language Definitions 50
Command 50
Variables 50
Constants 54
Compiler related Commands and Directives 55
Arithmetic Expressions 56
Conditionals 57
Error Handling 64
Event Handling 65
Miscellaneous Commands 67
5. Core Language Built-in Routines 71
Information Routines 71
Event & Messaging Routines 74
Arithmetic Routines 75
String Routines 76
Table Routines 95
Miscellaneous Routines 98
Random Number Generation Routines 99
Timer Routines 101
Circular Buffer Management Functions 107
Serial Communications Routines 112
Non-Volatile Memory Management Routines 142
Input/Output Interface Routines 145

User Routines........149
6. BLE Extensions Built-in Routines........153
Events and Messages........153
Miscellaneous Functions........162
Advertising Functions........163
Connection Functions........171
Security Manager Functions........177
Bespoke GATT Server Functions........181
Predefined GATT Server Functions........236
Pairing/Bonding Functions........253
Virtual Serial Port Service - Managed........255
7. Other Extension Built-in Routines........265
System Configuration Routines........265
Miscellaneous Routines........266
8. Events & Messages........268
9. Module Configuration........268
10. Miscelleaneous........269
Index........271

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 BLE-enabled end products into a straightforward process and 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 applications which were not structured and liberally used 'GOTO' statements.

At the outset, *smart* BASIC was developed by Laird to offer structured programming constructs and, because of this, is not line number based; 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 will always be 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:

1. Variable declarations and initialisations
2. Subroutine definitions
3. Event handler routines
4. 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 Another Language?

Programming languages are designed predominantly for arithmetic operations, data processing, string manipulation, and flow control. Where a program needs to interact with the outside world, line in a Bluetooth Low Energy device, it inevitably 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.

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. That makes it simpler to construct and understand, particularly if you're not using it every day.

Our other aim in developing *smart* BASIC is to make wireless design of products simple and contain a common 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 that are stored there, allowing a complete product design to be implemented without the need for any additional **external** processing capability.

What Are the Reasons for Writing 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 every 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 like 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. These present a consistent API interface defined to your host system and allow 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 further speed up the development process.

What is in a BLE Module?

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 available)
- Bluetooth Low Energy 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 modules from a hardware perspective on the left and a firmware/software perspective on the right:

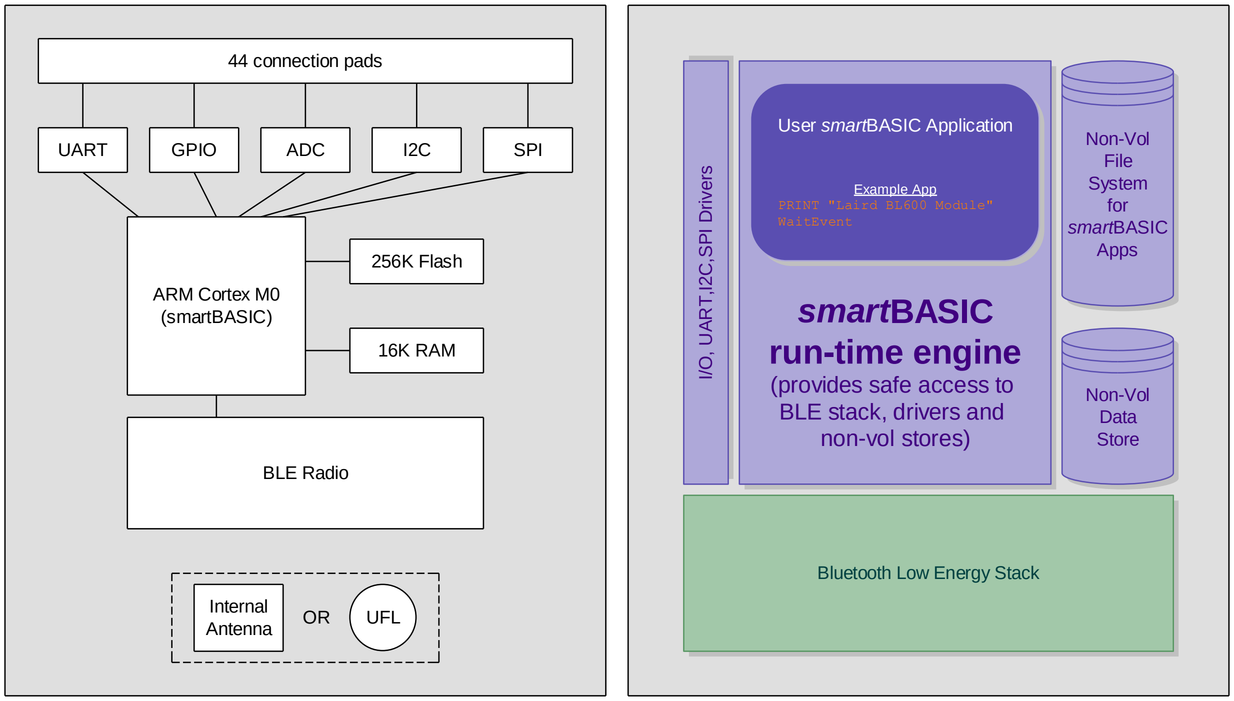

44 connection pads
UART
GPIO
ADC
I2C
SPI
ARM Cortex M0 (smartBASIC)
256K Flash
16K RAM
BLE Radio
Internal Antenna
OR
UFL
I/O, UART,I2C,SPI Drivers
User smartBASIC Application
Example App
PRINT "Laird BL600 Module"
WaitEvent
smartBASIC run-time engine
(provides safe access to BLE stack, drivers and non-vol stores)
Non-Vol File System for smartBASIC Apps
Non-Vol Data Store
Bluetooth Low Energy Stack

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 usage, 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 like 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 to develop on because a lot of functionality comes prepackaged for you. 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, which will be referred to as bytecode, that can be stored to a custom designed flash file system, and then the run-time engine interprets the application bytecode in-situ from flash.

To simplify development further, Laird provides its own custom developed application called UWTerminal which is a full blown customised terminal emulator for Windows, available on request for free. Chapter 2 – UWTerminal provides a Quick Start Guide to writing *smart*BASIC applications using UWTerminal.

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

Operating Modes of a *smart*BASIC module

Any platform running *smart* BASIC has up to three different 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 analogous 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**

***smart* BASIC applications in the flash file system that will subsequently be 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 on 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 capability of being able to run the application from flash ensures that as much RAM memory as possible is available to the user application to be used 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 an application called **$autorun$** 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 will enter Interactive Mode.

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

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

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

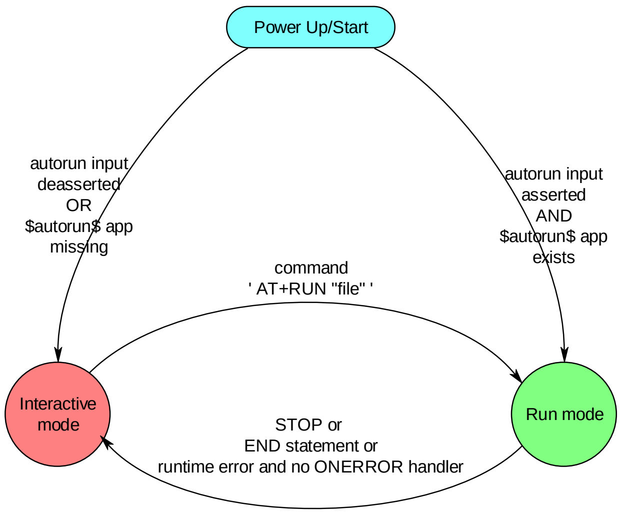

Power Up/Start
autorun input deasserted OR $autorun$ app missing
autorun input asserted AND $autorun$ app exists
command ' AT+RUN "file" '
Interactive mode
Run mode
STOP or END statement or runtime error and no ONERROR handler

Figure 2: Module modes & transitions

Types of Applications

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

- **Autorun Application** – This is a normal application that is given the specific name "**$autorun$**" and is case insensitive. When a *smart* BASIC module powers up, it looks for an application called "$autorun$". If it finds this application and 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 usage cases, it is expected that the autorun application never terminates and so it will be typical that the last statement in an application will 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 such as UART, I2C and SPI to interface with peripherals such as displays and sensors.

  NOTE: When the autorun application starts up, by default, 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 will be automatically opened using default comms paramaters.
- **Other Applications –** Applications can be loaded into the BASIC module and be 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 be stored in flash memory too, for example X.501 certificates. The most common non-application files are data files for use by applications.
- **Application Storage –** Storage of user applications and the 'AT+RUN' command is used to select which application runs.
- **Non-volatile record s –** 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. 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.

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

Note that the '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 to 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 series module which is a Bluetooth Low Energy module. However the principles apply to any *smart* BASIC enabled module.

What You Need

To replicate this example, you need the following items:

- A BL600 series development kit
- A copy of the latest UWTerminal application (Contact Laird for the latest version). The version of UWTernimal 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 8 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 will be 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 20mA. 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 write and run *smart* BASIC applications locally. This allows you to write *smart* BASIC applications which use the terminal emulation extensions that will enable you to automate the functionality of the terminal emulator.

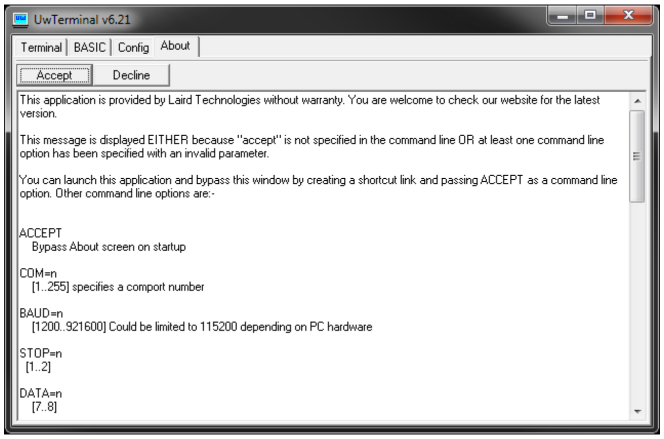
It may be possible in the future to add BLE extensions so that when UWTerminal is running on a Windows 8 PC, which has Bluetooth 4.0 hardware, an application that runs on a BLE module will also run in the UwTerminal environment.

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

Note: The driver for the USB to Serial chipset on the development kit generates a virtual COM port. You can check what this is by selecting **My Computer > Properties > Hardware > Device Manager > Ports (COM & LPT)**.

To use UWTerminal, follow the steps below and 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).



This application is provided by Laird Technologies without warranty. You are welcome to check our website for the latest version.

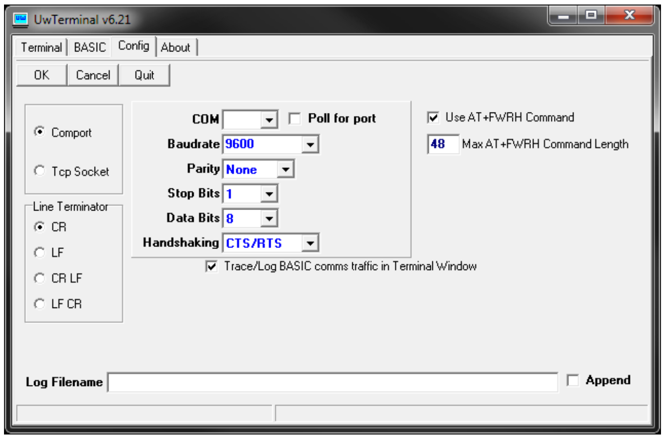
This message is displayed EITHER because "accept" is not specified in the command line OR at least one command line option has been specified with an invalid parameter.

You can launch this application and bypass this window by creating a shortcut link and passing ACCEPT as a command line option. Other command line options are:-

```
ACCEPT
   Bypass About screen on startup

COM=n
   [1..255] specifies a comport number

BAUD=n
   [1200..921600] Could be limited to 115200 depending on PC hardware

STOP=n
   [1..2]

DATA=n
   [7..8]
```

Figure 3: UWTerminal opening screen

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



(● Comport) ○ Tcp Socket

Line Terminator: (● CR) ○ LF ○ CR LF ○ LF CR

COM: ☐ Poll for port
Baudrate: 9600
Parity: None
Stop Bits: 1
Data Bits: 8
Handshaking: CTS/RTS
☑ Trace/Log BASIC comms traffic in Terminal Window

☑ Use AT+FWRH Command
48 Max AT+FWRH Command Length

Log Filename: ☐ Append

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**

Please note: **Comport** should be selected on the left and not 'Tcp Socket'.

5. Check **Poll for port** to enable a feature in UWTerminal 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 will be sent when you type **ENTER**.
7. Once these settings are correct, click **OK** to bring up the main terminal screen.

Getting around UWTerminal

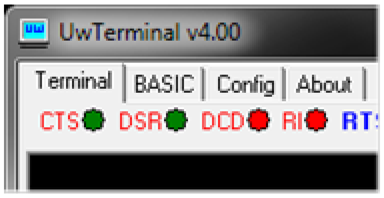


Figure 5: UWTerminal tabs and status lights

The following tabs (with four status lights below) 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 colour will be 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 Sense 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 4 are white, it means that the serial port of the PC has not been opened as shown below and the button labelled "OpenPort" can be used to open the port.

Please note on the BL600 Development kit, at the time of this manual being written, the DSR line 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.

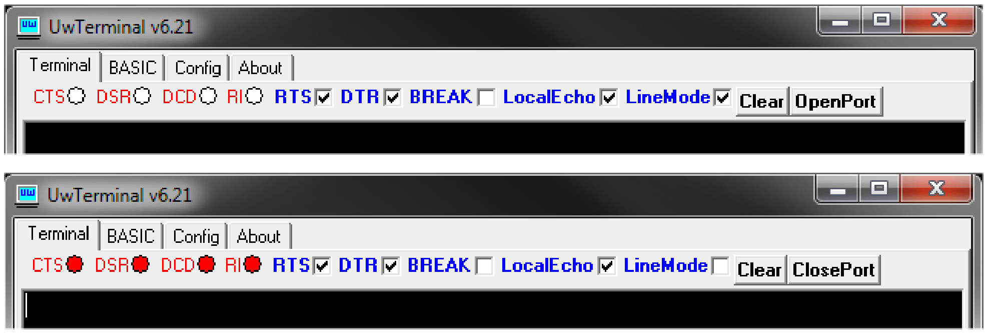


Figure 6: Control options

Next to the indicators are a number of control options (**Error! Reference source not found.**) 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** – Close 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-open 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.

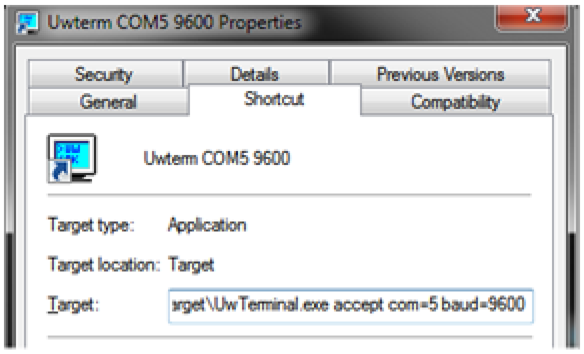
Each time UWTerminal starts, it asks you to acknowledge the Accept screen and to enter the COM port details. If you are not going to change these, you can skip these screens by entering the applicable command line parameters in a shortcut link.

To do this, follow these steps to create a shortcut to UWTerminal on your desktop:

1. Locate the file UwTerminal.exe and right click, then drag and drop onto your desktop, whereupon you will get a dialog box and from there select "Create Shortcut"
2. Right-click the newly created shortcut.
3. Select **Properties**.
4. Edit the **Target** line to add the following commands (Figure 7):

   accept com=n baud=bbb linemode

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

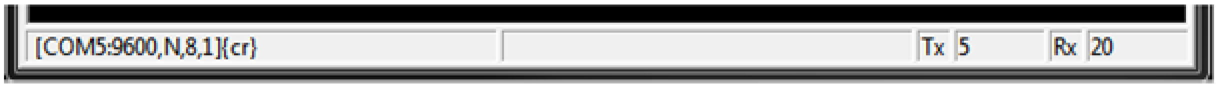


Target type: Application
Target location: Target
Target: arget\UwTerminal.exe accept com=5 baud=9600

Figure 7: Shortcut properties

Subsequently, starting UWTerminal from this shortcut launches it directly into the terminal screen. At any time, the status bar on the bottom left (Figure 8) 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.



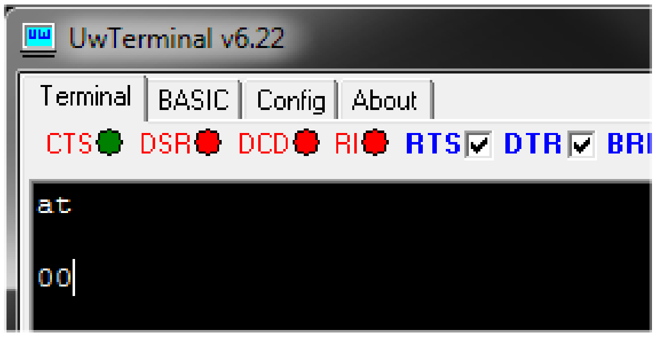
```
[COM5:9600,N,8,1]{cr}                    Tx 5     Rx 20
```

Figure 8: 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 'led' is green (DSR,DCD,RI should be red).
2. Type '**at**' (without the quotation marks).
3. Press **Enter**. You should get a 00 response as per the following screenshot :-



```
at

00
```

Figure 9: 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, then it is even 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**:

```
00
at+dir

06      $factory$
00
```

If the module has not been used before then you should not see any lines starting with the 2 digit '06' sequence.

Writing a *smart* BASIC Application

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 'Hello World' *smart* BASIC application, follow these steps:

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

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

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

Note: *smart* BASIC files can have any extension, as UWTerminal, which is used to download an application to the module, will strip the extension when the file is downloaded to the module.
Laird recommends always using the extension '.sb' as this makes it easy to distinguish between *smart* BASIC files and other files. You can also associate this extension with your favourite editor and enable appropriate syntax highlighting.

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: If you use TextPad and mark files with .sb extensions as C/C++ files (via Configure | Preferences) then you should be able to see your application with syntax colour highlighting. It is planned in the future to supply a configuration file for TextPad which will contain syntax highlighting information specifically for *smart* BASIC.

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

To manage file downloads, right click on any part of the black UWTerminal screen to display the drop-down menu (Figure 10).

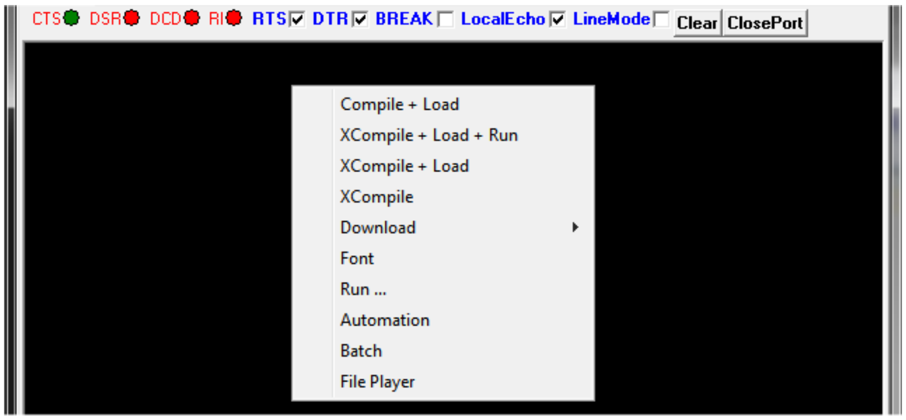


Figure 10: Right-click UWTerminal screen

4. Click **XCompile+Load** and navigate to the directory where you've stored your *test1.sb* file. Note: do not select **Compile+Load**
5. 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]>>**.

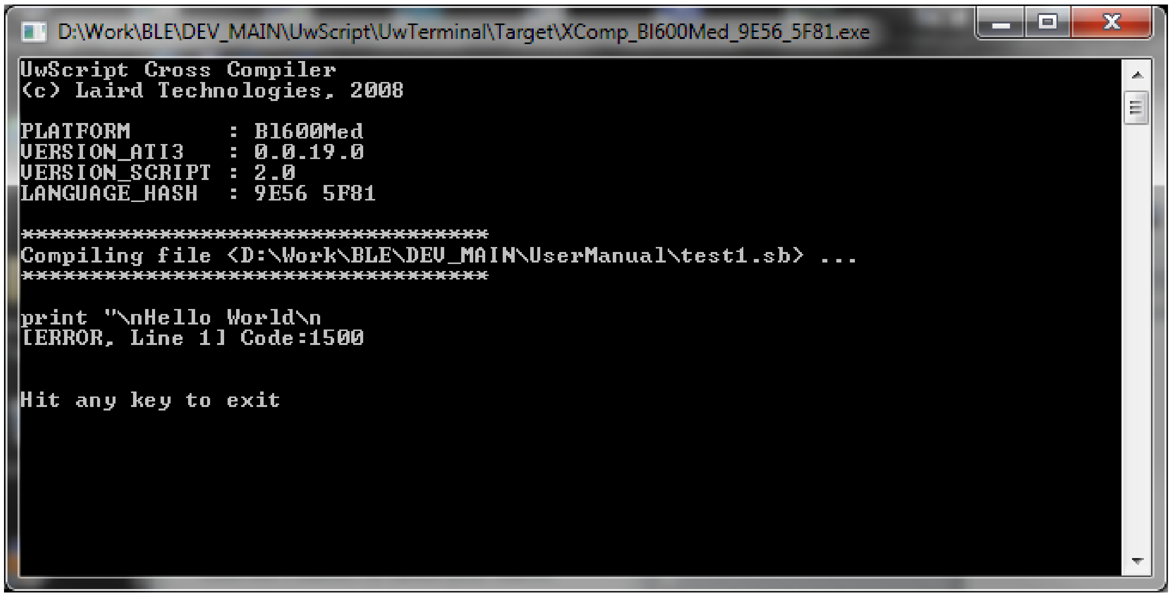
In this example, since the compilation is successful, the binary file generated needs to be downloaded and the **AT+DEL "filename" +** deletes any previous file with the same name that might already be on the module. The new file is downloaded using he **AT+FOW**, **AT+FWRH** and **AT+FCL** commands.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

A possible failure in this process is if the cross compiler cannot be located. In this case you should see the following display in the same window:

```
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 ???
```

The solution is to 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

Another cause of failure is if there is a compilation error. Say, for example, the print statement contained an error in the form of a missing " delimiter, then you should see the following display in a separate window:-



```
D:\Work\BLE\DEV_MAIN\UwScript\UwTerminal\Target\XComp_BI600Med_9E56_5F81.exe
UwScript Cross Compiler
(c) Laird Technologies, 2008

PLATFORM        : BI600Med
VERSION_ATI3    : 0.0.19.0
VERSION_SCRIPT  : 2.0
LANGUAGE_HASH   : 9E56 5F81

***********************************
Compiling file <D:\Work\BLE\DEV_MAIN\UserManual\test1.sb> ...
***********************************

print "\nHello World\n
[ERROR, Line 1] Code:1500


Hit any key to exit
```

Now that the application has been downloaded into the module, run it by issuing one of the following commands:

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 result (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:

```
00
at+dir

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 flavour 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 }
```

Where anything in *{ }* is mandatory and < > is optional and 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 which is 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 behaviour, e.g. with #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 depending on their content and its operating mode) and does ***one*** of the following:

- Act on them immediately (such as with AT commands).
- Optionally, if the build includes the compiler, generate 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 need to 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 Script "helloworld"

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                        'Increment the value of i
```

We have introduced a few new things, the first being comments. Any line that starts with an apostrophe ' is ignored by the compiler from the token onwards and treated as a comment, so the opening line is ignored. You can also add comments 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”
```

You'll see the following screen output:

```
at+run "helloworld"

Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World

00
```

If you now change the print statement in the application to

```
print "Hello World “;I;\n"      'The \n forces a new line each time
```

You'll see the following screen output:

```
at+run "helloworld"

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

00
```

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**.

```
at+dir

06    test1
06    HelloWorld
00
```

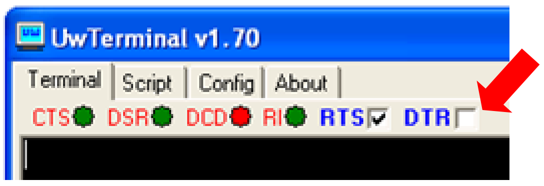
Note: All responses to interactive commands are of the format **\nNN\tOptionalText1\tOptionalText2…\r** where ***NN*** is always a two digit number and **\t** is the tab character and is terminated by **\r**.
This format has been provided to assist with developing host algorithms that can parse these 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 last 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 pinout for the BL600 module) and, if it is asserted (that is, 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 ticked by default, hence asserted state, which will translate to a 0v at the nAutoRUN input of the module. This means if an autorun application exists in the module's file system it will be 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 if you enter the AT+DIR command you should see:-

```
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.

Now clear the UWTerminal screen by clicking the ‘Clear’ button on the toolbar and then enter the command **ATZ** which forces the module to reset itself. You could also hit the ‘reset’ button on the development kit to achieve the same.

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

You’ll see the following screen output:

```
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
```

Next, in UWTerminal clear the screen using the ‘Clear’ button and then untick the checkbox labelled DTR so that the nAutoRUN input of the module is not asserted, and you will see that 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 to suppress the launching of the autorun application is purely to ensure that if your autorun application has the WAITEVENT as the last statement then you can still 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 for the die-hard senior coders). 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 the IDs are unique. The compiler will not check for repeated values.

Each breakpoint statement has 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:-

```
DIM i as integer

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

Then when you launch the application using AT+RUN you will see the following:-

```
at+run "test2"

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

00
```

And if you launch the application using AT+DBG you will see the following:-

```
at+dbg "test2"

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** will rerun 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:-

```
at+dbg "test2"

Hello World 1
Hello World 2
Hello World 3
21    BREAKPOINT 3333
= I 8
resume
Hello World 8
Hello World 9
Hello World 10
00
```

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 has been 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 will forget 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 which monitors button presses and reflects them to leds on the BLE development kit is as follows:-

```
'//******************************************************************************
'// Laird Technologies (c) 2013
'//
'// 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
'//
'//******************************************************************************

'//******************************************************************************
'// Global Variable Declarations
'//******************************************************************************
```

```
dim rc '// declare rc as integer variable


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

'//=============================================================================
'// This handler is called when button 0 is released
'//=============================================================================
function button0release()
gpiowrite(18,0) '// turns LED 0 off
print "Button 0 has been released \n"
print "LED 0 should now go out \n\n"
endfunc 1

'//=============================================================================
'// This handler is called when button 0 is pressed
'//=============================================================================
function button0press()
gpiowrite(18,1)
print "Button 0 has been pressed \n"
print "LED 0 will light while the button is pressed \n"
endfunc 1

'//=============================================================================
'// This handler is called when button 1 is released
'//=============================================================================
function button1release()
gpiowrite(19,0)
print "Button 1 has been released \n"
print "LED 1 should now go out \n\n"
endfunc 1

'//=============================================================================
'// This handler is called when button 1 is pressed
'//=============================================================================
function button1press()
gpiowrite(19,1)
print "Button 1 has been pressed \n"
print "LED 1 will light while the button is pressed \n"
endfunc 1


'//*****************************************************************************
'// Startup code : equivalent to main() in C
'//*****************************************************************************
rc = gpiosetfunc(18,2,2) '//sets sio18 (LED0) as a digital out with a weak pull up
rc = gpiosetfunc(19,2,2) '//sets sio19 (LED1) as a digital out with a weak pull up
rc = gpiobindevent(0,16,0) '//binds a gpio high event to an event. sio16 (button 0)
rc = gpiobindevent(1,16,1) '//binds a gpio low event to an event. sio16 (button 0)
rc = gpiobindevent(2,17,0) '//binds a gpio high event to an event. sio17 (button 1)
rc = gpiobindevent(3,17,1) '//binds a gpio low event to an event. sio17 (button 1)

'//=============================================================================
'//Bind events to handler functions
'//=============================================================================
onevent evgpiochan0 call button0release '//handler for button 0 release
onevent evgpiochan1 call button0press   '//handler for button 0 press
onevent evgpiochan2 call button1release '//handler for button 1 release
onevent evgpiochan3 call button1press   '//handler for button 1 press
```

```
print "Ready to begin button and LED test \n"
print "Please press button 0 or button 1 \n\n"

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:-

```
AT+RUN "sampleapp"

Ready to begin button and LED test
Please press button 0 or button 1

Button 0 has been pressed
LED 0 will light while the button is pressed
Button 0 has been released
LED 0 should now go out

Button 1 has been pressed
LED 1 will light while the button is pressed
Button 1 has been released
LED 1 should now go out
```

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:

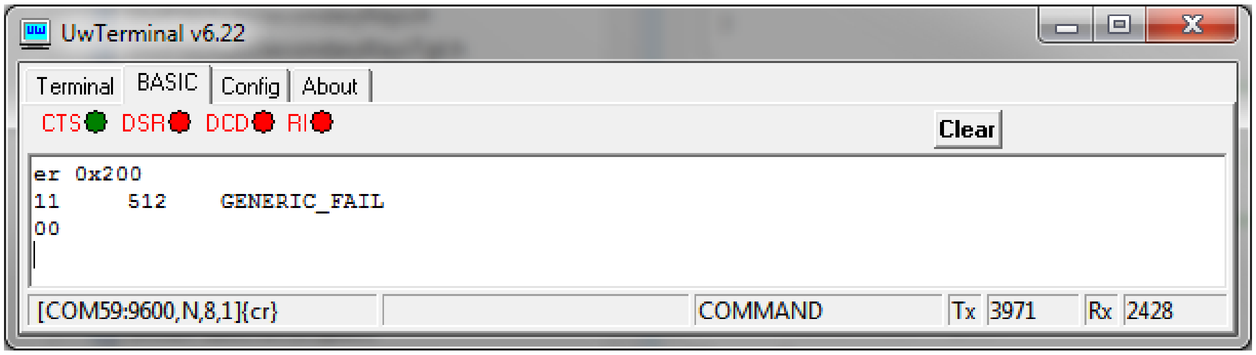
<lf>00<cr> for a successful outcome, or

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

Note that in the case of the 01 response the "<tab>optional_verbose_explanation" will be missing in resource constrained platforms like the BL600 modules. The 'verbose explanation' is a constant string and since there are over 1000 error codes, these verbose strings can occupy more than 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, in UWTerminal, click on the BASIC tab and then if the error value is hhhh, enter the command "ER 0xhhhh" and note the 0x prefix to 'hhhh'. This is illustrated in the following screenshot.



```
er 0x200
11      512     GENERIC_FAIL
00
```

[COM59:9600,N,8,1]{cr} COMMAND Tx 3971 Rx 2428

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

AT

An Interactive Mode command. 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
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
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)
03	Random Private Non-Resolvable (used for reconnections)

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

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

AT+DIR <"string">

Command

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

This command is used to delete a file from the module's flash file system.

When the file is deleted the space it used to occupy does not get marked as free for use again, Hence eventually after many deletions the file system will not have any free space for new files. When that happens the module will respond 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 will respond as follows:-

```
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 that after AT&F 1 has been processed, the file system manager context is unstable so there will be an automatic self-reboot.

AT+DEL "filename" (+)

Command

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

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"

Command

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 will be 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 just 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

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"

Command

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

AT+SET is used to set a run-time configuration key. Configuration keys are user definable, non-volatile memory storage areas, which are analogous to S registers in modems. Their values are kept over a power cycle but will be deleted if the AT&F* command is used to clear the file system.

AT+SET num = string

Command

Returns	If the config key is successfully set or updated, 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.
String	String_constant The entire value array is written to the configuration ID and is specified in a single command (in contrast to the returned values of AT+GET). The new value array is specified as fixed format 4 digit hex numbers (with optional H' prefixes).

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 command 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 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 will be forced by the property of the TX characteristic of the service.
102	This is the advert interval in milliseconds when advertising for connections in command mode. Valid values are :- 20 to 10240 milliseconds
103	This is the advert timeout in milliseconds when advertising for connections in command mode. Valid values are :- 0 to 16383 seconds, and 0 disables the timer hence continuous
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 will allow 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 command mode submitting the command ATi 2014
105	When in command mode and connected for virtual serial port services this will be the minimum connection interval in milliseconds to be negotiated with the master. Valid value is 0 to 4000ms and if a value of less than 8 is specified then the minimum value of 7.5 will be selected.
106	When in command mode and connected for virtual serial port services this will be the maximum connection interval in milliseconds to be negotiated with the master. Valid value is 0 to 4000ms and if a value of less the minimum specified in 105 then it will be forced to the value in 105 + 2ms
107	When in command mode and connected for virtual serial port services this will be 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 will be used.
108	When in command mode and connected for virtual serial port services this will be the slave latency to be negotiated with the master. An adjusted value will be used if this value times the value in 106 is greater than the supervision timeout in 107
109	When in command mode and connected for virtual serial port services this will be the tx power used for adverts and connections. Main reason setting a low value is to ensure that in production, if smartBASIC applications are downloaded over the air then limited range will allow many stations to be used to program devices.
110	If Virtual Serial Port Service is enabled in command 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 command 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.

Interactive Command: YES

```
'Example:
AT+SET 40 = "0x0040"
AT+SET 40 = "H'0040"
```

AT+SET 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+GET

AT+GET is used to read a run-time configuration key. Configuration keys are user definable, non-volatile memory storage areas, which are analogous to S registers in modems. Their values are kept over a power cycle.

AT+GET num

Command

Returns The response to this command is

\n07\tiiii oooo hhhh hhhh hhhh hhhh\r
\n00\r

Where each line starting with 07 will have up to 8 words. If the configuration key contains more data words, then more of these 07 lines are displayed.

In each 07 line the **oooo** value (hexadecimal) specifies the start offset of the data in the key. The value **iiii** (hexadecimal) is an echo of the config key ID specified in the command line. The config key data is **hhhh** again in hexadecimal.

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.

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

See the definition of the AT+SET command for a list of all the predefined configuration keys.

Interactive Command: YES

```
'Example:

AT+GET 40
07    0028 0000 0014
00
```

AT+GET is a core command.

AT+FOW

AT+FOW opens a file to allow it to be written to 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"

Command

Returns If the filename is valid, AT+FOW will respond 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

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"

Command

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 will get 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

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"

Command

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

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

Command

Returns If the filename exists, AT+FCL will respond 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)

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

? var <[index]>

Command

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)

When an application encounters a STOP, BPnnn, or END statement, it will fall into the Interactive Mode of operation and will 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 will contain the new value. It is useful when debugging applications.

= var<[n]> value

Command

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

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

Command

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

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

Command

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 command mode)

AT+RUN "someapp"
ABORT
```

ABORT is a core command.

AT+REN

Renames an existing file.

AT+REN "oldname" "newname"

Command

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

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

AT&F integermask

Command

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+GET and AT+SET for examples of config keys)
2	Not applicable to current modules
4	Not applicable to current modules
8	Not applicable to current modules
16	Erases the User config keys only
32	Not applicable to current modules
*	Erases all data segments

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

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.

Interactive Command: YES

```
'Examples:

AT Z
```

AT Z is a core command.

AT + BTD *

Deletes the bonded device database from the flash.

AT + BTD*

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 that the module will self-reboot so that the bonding manager context is reset too.

Interactive Command: YES

```
'Examples:

AT+BTD*
```

AT+BTD* is an extension command

AT + MAC "12 hex digit mac address"

This is a command that will be successful one time only 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"

Command

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 that the module will self-reboot if the write is successful. Subsequent invocations of this command will generate an error.

Interactive Command: YES

```
'Examples:

AT+MAC "008098010203"
```

AT+MAC is an extension command

AT + BLX

This command is used to stop all radio activity whether it be adverts or connections when in command mode. Particularly useful when the virtual serial port is enabled while in command 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 that the module will self-reboot so that the bonding manager context is reset too.

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, users need to take care of the syntax of these commands.

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 hand 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 Arguments 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, on exit, any changes are lost.

Passing Arguments as byRef

If an argument is passed as byRef, then the function or subroutine can modify the variable and, on 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.

A characteristic of functions, as opposed to subroutines, is that they 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 0. 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 mitigated.

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 will abort to Interactive Mode. This WILL be 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 will reset 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

Description of the command.

COMMAND (<byRef | byval> *arg1 <AS type>*,..)
FUNCTION / SUBROUTINE / STATEMENT

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.

  Interactive Command: NO

```
'Example:

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
```

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

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

- **Complex Variables** – 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:

DIM i$ as STRING
DIM a$ as STRING
a$ = "Laird"
i$ = a$ + "Rocks!"
```

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:

DIM nCmds AS INTEGER
DIM stCmds[20] AS STRING   'declare an array as a string with 20 elements
stCmds[0]="ATS0=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

DIM i AS INTEGER
for i 0 to nCmds step 1
  SendData(stCmds[i])
  WaitForOkResp()
Next
```

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, i.e *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 or 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
Linefeed	\n
Carriage return	\r
Horizontal Tab	\t
\	\5C
"	\22 or ""
A	\41
B	\42
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 =

CmdID	MinVal	MaxVal	Default	Comments
				ByRef, 1=ByVal
2	0	1	1	Default Complex Arguments type for routines. 0 = ByRef, 1=ByVal
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 (e.g INTEGER).

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

Variable types should not be mixed.

```
'Examples:

DIM Sum1,bit1,bit2
DIM Volume,height,area

Sum1 = bit1 + bit2
Volume = height * area
```

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 the '(' and ')' brackets 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 that *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+SET command.

DO / UNTIL

This DO / UNTIL construct allows a block of statements, consisting 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 as defined in the section 'Arithmetic Expressions'.

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

```
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
```

DO / UNTIL is a core function.

DO / DOWHILE

This DO / DOWHILE construct allows a block of statements, consisting 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 as defined in the section 'Arithmetic Expressions'.

For DO / DOWHILE, if the arithmetic expression does not evaluate 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

```
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
```

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. Use of 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 4 variants as follows:

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

```
DIM A
FOR A=1 TO 2                    'output  -HelloHello
  PRINT "Hello"
NEXT
FOR A=2 DOWNTO 1                'output  -HelloHello
  PRINT "Hello"
NEXT
FOR A=1 TO 4 STEP 2             'output  -HelloHello
  PRINT "Hello"
NEXT
```

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

```
DIM N
N=1
IF N>0 THEN
  PRINT "Laird Rocks"
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
```

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

```
DIM N
N=0

'now print "Hello" ten times
```

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

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 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 will always be processed. When the first CASE statement is encountered, execution will move 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

```
DIM A,B,C

A=3 : B=4
```

```
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
```

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

```
DIM N
N=0

'now print "Hello" ten times

WHILE N<10
  PRINT "Hello " ;N
  N=N+1
  IF N==5 THEN
    BREAK                     'Only 5 Hello will be printed
  ENDIF
ENDWHILE
```

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

```
WHILE N<10
  N=N+1
  IF N==5 THEN
    CONTINUE                'The 5th Hello will not get printed
  ENDIF
  PRINT "Hello " ;N
ENDWHILE
```

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 itself 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 and they are 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

```
DIM A,B,C
SUB HandlerOnErr()
  PRINT "Divide by 0 error"
```

```
ENDSUB
A=100 : B=0
ONERROR NEXT HandlerOnErr
C=A/B
```

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.

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 ONFATALERROR command being compiled.

Interactive Command: NO

```
DIM A,B,C
SUB HandlerOnErr()
  PRINT "Divide by 0 error"
ENDSUB
A=100 : B=0
ONFATALERROR NEXT HandlerOnErr
C=A/B
```

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 will continue 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 EV0 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 will only be 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 will continue to wait for more events. A zero value will force 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

```
FUNCTION Func0()
  PRINT "\nTimer 0"
ENDFUNC 1

FUNCTION Func1()
  PRINT "\nTimer 1"
ENDFUNC 1

ONEVENT EVTMR0 CALL Func0
ONEVENT EVTMR1 CALL Func1

TIMERSTART(0,500,0)
TIMERSTART(1,1500,0)

WAITEVENT                        'wait  for an event  to occur
```

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 in most platforms a UART interface.

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 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 will result 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 tab before next variable
;	print next variable without any intervening whitespace

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 will be 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

```
PRINT  "Hello"
DIM A
A=100
PRINT A             'print as decimal
PRINT h'A           'print as hex
PRINT o'A           'print as octal
PRINT b'A           'print as binary
```

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 will result 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 tab before next variable
;	print next variable without any intervening whitespace

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 will be 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

```
DIM A, S$
A=100
SPRINT #S$,A                'S$ var will contain 100
PRINT S$
SPRINT #S$,h'A              'S$ var will contain 64
SPRINT #S$,o'A              'S$ var will contain 144
SPRINT #S$,b'A              'S$ var will contain 1100100
```

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

```
'Examples:

STOP
```

STOP is a core function.

BP

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 just a developer supplied identifier which will get 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 will then be 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 that the next state will be the BP statement itself, hence multiple SO commands may need to be issued.

Command

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 64Kword 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

```
'Examples:

PRINT "hello"
BP 1234
PRINT "world"
BP 5678
```

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.

Information Routines

GETLASTERROR

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.

GETLASTERROR ()

Function

Returns	INTEGER Last error that was generated.
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	None

Interactive Command: NO

```
DIM err
err = GETLASTERROR()
print "\nerror  = 0x" ; h'err            'print it as a hex value
```

GETLASTERROR is a core function.

RESETLASTERROR

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

RESETLASTERROR ()

Subroutine

Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	None

Interactive Command: NO

```
RESETLASTERROR()
```

RESETLASTERROR is a core function.

SYSINFO

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

SYSINFO(varId)

Function

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
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
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
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 command mode submit the command AT I 0x8010.

Interactive Command: No

```
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
```

SYSINFO is a core language function.

Event & Messaging Routines

SENDMSGAPP

This function is used to send a 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)

Function

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

```
DIM rc
```

```
FUNCTION HandlerMsgApp(BYVAL nMsgId AS INTEGER, BYVAL nMsgCtx AS INTEGER) AS INTEGER
  PRINT "\nId=";nMsgId;" Ctx=";nMsgCtx  '//output will be 100,200
ENDFUNC 1

ONEVENT EVMSGAPP CALL HandlerMsgApp

rc = SendMsgApp(100,200)

WAITEVENT
```

SENDMSGAPP is a core function.

Arithmetic Routines

ABS

Returns the absolute value of its INTEGER argument.

ABS (var)

Function

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 will cause 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

```
DIM s1 as INTEGER,s2 as INTEGER
S1 = -2 : s2 = 4
PRINT S1, ABS(S1);"\n";s2, ANS(s2)
```

ABS is a core language function.

MAX

Returns the maximum of two integer values.

MAX (var1, var2)

Function

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

```
DIM s1 as INTEGER,s2 as INTEGER
S1 = -2 : s2 = 4
PRINT s1, MAX(s1,s2)
```

MAX is a core language function.

MIN

Returns the minimum of two integer values.

MIN (var1, var2)

Function

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

```
DIM s1 as INTEGER,s2 as INTEGER
S1 = -2 : s2 = 4
PRINT s1, MIN(s1,s2)
```

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 Bluetooth Low Energy 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

```
DIM newstring$                          ' declare strings
DIM ss as STRING                        ' should really append a $ to the name

ss="Arsenic"
newstring$ = left$(ss,4)                'get the four leftmost characters
print newstring$; "\n"
```

LEFT$ is a core language function.

MID$

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)

Function

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 will return 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

```
DIM newstring$ AS STRING
DIM ss$

Ss$="Arsenic"
Newstring$ = mid$(ss$,0,4)  'get the four leftmost characters
print newstring; "\n"

DIM longstring$ AS STRING
DIM len AS INTEGER  'the Length variable must be an integer
DIM pos AS INTEGER

Longstring$ = "abcdefghijkl"
pos=0 : len = 6
newstring$ = mid$(longstring$,pos,len)
      '//newstring$ will be – abcdef

pos = 2 : len = 5
newstring$ = mid$(longstring$,pos,len)
      '//newstring$ will be - cdefg

pos = -5 : len = 3
newstring$ = mid$(longstring$,pos,len)
      '//newstring$ will be - hij
```

MID$ is a core language function.

RIGHT$

Retrieves the rightmost n characters from a string.

RIGHT$(string, len)

Function

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

```
DIM newstring$
DIM ss$ as STRING

ss$="Parse"
newstring$ = right$(ss$,4)              : 'get the four rightmost characters
print newstring$; "\n"
```

RIGHT$ is a core function.

STRLEN

STRLEN returns the number of characters within a string.

STRLEN (string)

Function

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

```
DIM s$
S$="HelloWorld"
PRINT "\n";S$;" is ";STRLEN(S$);" bytes long"
```

STRLEN is a core function.

STRPOS

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)

Function

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

```
DIM s1$,s2$
S1$="Are you there"
S2$="there"
PRINT "\nIn ";S1$;" the word ";S2$;" occurs at position ";STRPOS(S1$,S2$,0)
```

STRPOS is a core function.

STRSETCHR

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)

Function

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

```
DIM s$
S$="Hello"
PRINT strsetchr(s$,64,0)         'output will be @ello
PRINT strsetchr(s$,64,5)         'output will be Hello@
PRINT strsetchr(s$,64,8)         'output will be Hello@@@@
```

STRSETCHR is a core function.

STRGETCHR

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

STRGETCHR (string, nIndex)

Function

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

```
 DIM s$
 S$="Hello"
 PRINT strgetchr(s$,0)     'output will be  72 which is the ascii value for 'H'
 PRINT strgetchr(s$,1)     'output will be  101 which is the ascii value for 'e'
 PRINT strgetchr(s$,-100)  'output will be  -1 because index is negative
 PRINT strgetchr(s$,6)     'output will be  -1 because index is larger than the string
length
```

STRGETCHR is a core function.

STRSETBLOCK

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 values 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

```
DIM s$
S$="HelloWorld"
PRINT strsetblock(S$,64,4,2)     'output will be 0
PRINT S$                      'output will be Hell@@orld
PRINT strsetblock(S$,64,4,200)   'output will be -1
```

STRSETBLOCK is a core function.

STRFILL

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

STRFILL (string, nChr, nCount)

Function

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

```
DIM s$
S$="hello"
PRINT strfill(s$,64,7)          'will output 0
PRINT s$                        'will output @@@@@@@
PRINT strfill(s$,-23,7)         'will output -1
```

STRFILL is a core function.

STRSHIFTLEFT

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)

SUBROUTINE

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

```
DIM s$
S$="123456789"
Strshiftleft(s$,4)        'drop leftmost 4 characters
PRINT s$             'output will be 56789
```

STRSHIFTLEFT is a core function.

STRCMP

Compares two string variables.

STRCMP(string1, string2)

Function

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

```
DIM s1$,s2$
s1$="hello"
s2$="world"
print strcmp(s1$,s2$)     'outputs   -1
print strcmp(s2$,s1$)     'outputs    1
print strcmp(s1$,s1$)     'outputs    0
```

STRCMP is a core function.

STRHEXIZE$

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)

Function

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

```
DIM S$,T$
S$="\01\02\03\04\05"
T$=strhexize$(S$)
PRINT strlen(S$)                    'outputs   5
PRINT strlen(T$)                    'outputs   10  and will contain "0102030405"
```

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 is encountered a nonfatal error is generated which needs to be handled otherwise 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

```
DIM S$,T$
S$="40414243"
```

```
PRINT strlen(S$)                'outputs 8
T$ = strdehexize$(S$)
PRINT strlen(T$)                'outputs 4  S$="@ABC"
S$="4041hello4243"
PRINT strlen(S$)                'outputs 13
T$ = strdehexize$(S$)
PRINT strlen(T$)                'outputs 2  S$="@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

```
DIM S$,B
S$="0102030405"
B=strhex2bin(S$,4)
PRINT B                         'outputs   3
```

STRHEX2BIN is a core function.

STRESCAPE$

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)

Function

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

```
DIM S$,T$
S$="Hello\00world"
T$=strescape$(S$)
PRINT strlen(S$)                'outputs   11
PRINT strlen(T$)                'outputs   13
```

STRESCAPE$ is a core function.

STRDEESCAPE

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 will never be a string longer than the original.

STRDEESCAPE (string)

SUBROUTINE

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 is encountered a nonfatal error is generated which needs to be handled otherwise 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

```
DIM S$,T$
S$="Hello\5C40world"
PRINT strlen(S$)                'outputs 15
strdeescape(S$)
PRINT strlen(S$)                'outputs 13 S$="Hello\40world"
strdeescape(S$)
PRINT strlen(S$)                'outputs 11 S$="Hello@world"
```

STRDEESCAPE is a core function.

STRVALDEC

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

```
DIM S$
S$="  1234"
PRINT "\n";strvaldec(S$)         'outputs    1234
S$=" -1234"
PRINT "\n";strvaldec(S$)         'outputs    -1234
S$=" +1234"
PRINT "\n";strvaldec(S$)         'outputs    1234
S$="  2345hello"
PRINT "\n";strvaldec(S$)         'outputs    2345
S$="  hello"
PRINT "\n";strvaldec(S$)         'outputs    0
```

STRVALDEC is a core function.

STRSPLITLEFT$

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)

Function

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

```
DIM OriginalString$, OriginalString$
OriginalString$ = "12345678"
NewString$ = stringsplitleft$ (OrigianlString$, 3)
print NewString$                                    ' The printed value will be 123
print "\n"
print OriginalString$                               ' The printed value will be 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

```
DIM Number$, Result1, Result2
Number$="01234"
Result1 = strsum(number$,0,5,0)
print Result1                       ' The printed result will be 250
Result2 = strsum(number$,0,5,10)
print Result2                       ' The printed result will be 260
```

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

```
DIM Number$, Result1, Result2
Number$="01234"
Result1 = strxor(number$,0,5,0)
print Result1                         ' The printed result will be 52
Result2 = strxor(number$,0,5,10)
print Result2                         ' The printed result will be 62
Result2 = strxor(number$,0,5,1000)
print Result2                         ' The printed result will be 988
```

STRXOR 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

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)

Function

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**

```
DIM T$
T$="Hello"
PRINT "\n";"[";T$;"]"            'output will be [Hello]
PRINT "\n";TABLEINIT(T$)         'output will be   0
PRINT "\n";"[";T$;"]"              'output will be []
```

TABLEINIT is a core function.

TABLEADD

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)

Function

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**

```
DIM T$
DIM resCode
resCode = TABLEINIT(T$)
PRINT TABLEADD(T$,"Hello",1)           'outputs   0
PRINT TABLEADD(T$,"world",2)           'outputs   0
PRINT TABLEADD(T$,"to",300)            'outputs   1
PRINT TABLEADD(T$,"",3)                'outputs   4
```

TABLEADD is a core function.

TABLELOOKUP

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)

Function

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**

```
DIM T$
DIM resCode
resCode = TABLEINIT(T$)
PRINT TABLEADD(T$,"Hello",100)   'outputs   0
PRINT TABLEADD(T$,"world",2)     'outputs   0
PRINT TABLEADD(T$,"to",3)        'outputs   0
PRINT TABLEADD(T$,"you",4)       'outputs   0

PRINT TABLELOOKUP(T$"to")        'outputs   3
PRINT TABLELOOKUP(T$"Hello")     'outputs   100
```

TABLELOOKUP is a core function

Miscellaneous Routines

This section describes all miscellaneous functions and subroutines

RESET

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

RESET (nType)

Subroutine

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

```
RESET(0)     'force a reset of the module
```

RESET is a core subroutine.

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

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 ()

Function

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

```
PRINT "\nRandom number is ";RAND()
```

RAND is a core language function.

RANDEX

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)

Function

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

```
PRINT "\nRandom number is ";RAND()
```

RAND is a core language function.

RANDSEED

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)

SUBROUTINE

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)
```

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)

```
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"
```

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

```
SUB HandlerOnErr()
  PRINT "Timer Error ";getlasterror()
ENDSUB

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

ONERROR NEXT HandlerOnErr

ONEVENT EVTMR0 CALL handlerTimer0
ONEVENT EVTMR1 CALL handlerTimer1

TIMERSTART(0,-500,1)   //start a -500 millisecond recurring timer
PRINT "\nStarted Timer 0 with invalid inerval"

TIMERSTART(0,500,1)   //start a 500 millisecond recurring timer
PRINT "\nWaiting for Timer 0"

TIMERSTART(0,1000,0) //start a 1000 millisecond timer
PRINT "\nWaiting for Timer 1"

WAITEVENT
PRINT "\nGot here because TIMER 1 expired and handler returned 0"
```

TIMERSTART is a core subroutine.

TimerRunning

This function is used to determine 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 that 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

```
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(0,2000,0) //start a 1000 millisecond timer
PRINT "\nWaiting for Timer 1"

WAITEVENT
```

TIMERRUNNING is a core function.

TimerCancel

This subroutine stops one of the built-in timers so that it will not generate a timeout event.

TIMERCANCEL (number)

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.

Interactive Command: NO

Related Commands: ONEVENT, TIMERCANCEL,TIMERRUNNING

```
FUNCTION handlerTimer0()
  PRINT "\nTimer 0 has expired"
  PRINT "\nCancelling Timer 1"
  TIMERCANCEL(1)
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(0,1000,0) //start a 1000 millisecond timer
PRINT "\nWaiting for Timer 1, but will never happen because cancelled in 0"

WAITEVENT
```

TIMERCANCEL is a core subroutine.

GetTickCount

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 ()

Function

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

```
DIM startTick,endTick,elapseMs

startTick = GETTICKCOUNT()
… do something
endTick = GETTICKCOUNT()

'Following code is an illustration - more efficient to use GETTICKSINCE() function
IF endTick > startTick THEN
  elapseMs = endTick - startTick
ELSE
  elapseMs = (0x7FFFFFF - startTick) + endTick
ENDIF

PRINT "\nsomthing took ";elapseMS; "msec to process"
```

GETTICKCOUNT is a core subroutine.

GetTickSince

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)

Function

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

```
DIM startTick, elapseMs
startTick = GETTICKCOUNT()
… do something

elapseMs = GETTICKSINCE(startTick)
PRINT "\nsomthing took ";elapseMS; "msec to process"
```

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 inherit 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

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)

FUNCTION

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()

dim rc
dim circHandle

// create a circular that can store a maximum of 16 integers
rc = CircBufCreate(16,circHandle)
if rc != 0 then
  print "The cicular buffer was not created"
endif
```

CIRCBUFCREATE is an extension function.

CircBufDestroy

This function is used to destroy a circular buffer previously created using CircBufCreate.

CIRCBUFDESTROY (circHandle)

SUBROUTINE

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()

dim rc
dim circHandle
dim i
dim nItems

// create a circular that can store a maximum of 16 integers
rc = CircBufCreate(16,circHandle)
if rc != 0 then
  print "The cicular buffer was not created"
endif

//
// Read and write from buffer
//

// No longer need the circular buffer
CircBufDestroy(circHandle)
```

CIRCBUFDESTROY is an extension function.

CircBufWrite

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)

FUNCTION

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()

dim rc
dim circHandle
dim i

// create a circular that can store a maximum of 16 integers
rc = CircBufCreate(16,circHandle)
if rc != 0 then
  print "The cicular buffer was not created"
endif

//write 3 values into the circular buffer
for i = 1 to 3
 rc = CircBufWrite(circHandle,i)
  if rc != 0 then
    print "Failed to write into the circular buffer"
  endif
next
```

CIRCBUFWRITE is an extension function.

CircBufOverWrite

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)

FUNCTION

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 will always be written into the buffer. Oldest is discarded to make space for this.

Interactive Command: NO

```
// Example :: CircBufOverWrite()

dim rc
dim circHandle
dim i

// create a circular that can store a maximum of 16 integers
rc = CircBufCreate(16,circHandle)
if rc != 0 then
  print "The cicular buffer was not created"
endif

//write 3 values into the circular buffer
for i = 1 to 3
 rc = CircBufOverWrite(circHandle,i)
  if rc == 0x5103 then
    print "Oldest value was discarded to save this one"
  elseif rc != 0
    print "Failed to write into the circular buffer"
  endif
next
```

CIRCBUFOVERWRITE is an extension function.

CircBufRead

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)

FUNCTION

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()

dim rc
dim circHandle
dim i
dim nData

// create a circular that can store a maximum of 16 integers
```

```
rc = CircBufCreate(16,circHandle)
if rc != 0 then
  print "The cicular buffer was not created"
endif

//write 3 values into the circular buffer
for i = 1 to 3
  rc = CircBufOverWrite(circHandle,i)
  if rc == 0x5103 then
    print "Oldest value was discarded to save this one"
  elseif rc != 0
    print "Failed to write into the circular buffer"
  endif
next

//read 4 values from the circular buffer
for i = 1 to 4
 rc = CircBufRead(circHandle,nData)
  if rc == 0x5102 then
    print "The buffer was empty"
  elseif rc != 0
    print "Failed to read from the circular buffer"
  else
    print "Read value = ";nData
  endif
next
```

CIRCBUFREAD is an extension function.

CircBufItems

This function is used to determine the number of integer items held in the circular buffer.

CIRCBUFITEMS(circHandle, nItems)

FUNCTION

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 which needs to be queried.

nItems **byRef *nItems* AS INTEGER**
This will return the number of items waiting to be read in the circular buffer

Interactive Command: NO

```
// Example :: CircBufItems()

dim rc
dim circHandle
dim i
dim nItems

// create a circular that can store a maximum of 16 integers
rc = CircBufCreate(16,circHandle)
if rc != 0 then
  print "The cicular buffer was not created"
endif

//write 3 values into the circular buffer
for i = 1 to 3
  rc = CircBufOverWrite(circHandle,i)
  if rc == 0x5103 then
    print "Oldest value was discarded to save this one"
  elseif rc != 0
    print "Failed to write into the circular buffer"
  endif

 rc = CircBufItems(circHandle,nItems)
  if rc==0 then
    print nItems;" items in the circular buffer"
  endif
next
```

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.

```
FUNCTION hdlrUartRx()
  PRINT "\nData has arrived"
ENDFUNC 1 //remain blocked in WAITEVENT

FUNCTION hdlrUartTxEty()
  PRINT "\nTx buffer is empty"
ENDFUNC 0 //exit from WAITEVENT


ONEVENT EVUARTRX      CALL hdlrUartRx
ONEVENT EVUARTTXEMPTY CALL hdlrUartTxEty

PRINT "\nSend this via uart"

WAITEVENT     //wait for rx, tx and modem status events
```

UartOpen

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 command mode, then those settings will be inherited by the command mode parser. Hence this is the only way to alter the communications parameters for Command 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)

Function

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

Please 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

```
DIM rc

//--- Handler to process receive data

FUNCTION hdlrUartRx()
  PRINT "\nData has arrived"
ENDFUNC 1 //remain blocked in WAITEVENT

//--- Register event handler for receive data

ONEVENT EVUARTRX      CALL hdlrUartRx

//--- Open comport so that DCD and RI are inputs

rc=UartOpen(9600,0,0,"CN81H")   //open as DCE at 9600 baudrate, no parity
                                //8 databits, 1 stopbits, cts/rts flow control
if rc!= 0 then
  print "\nFailed to open UART interface with error code ";interger.h' rc
else
  print "\nUART open success"
endif

WAITEVENT     //wait for rx, events
```

UARTOPEN is a core function.

UartClose

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 is safe to call if it is already closed.

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()

Subroutine

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

```
DIM rv
DIM mdm

//--- Open comport so that DCD and RI are outputs

rv=UartOpen(300,0,0,"TO81H")     //open as DTE at 300 baudrate, odd parity
                                 //8 databits, 1 stopbits, cts/rts flow control
UartClose()  //close the port
UartClose()  //no harm done doing it again
```

UARTCLOSE is a core subroutine.

UartInfo

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

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

```
DIM rv

//--- Handler to process receive data

FUNCTION hdlrUartRx()
  PRINT "\nThis many bytes in rx buffer ";uartinfo(3)
ENDFUNC 1 //remain blocked in WAITEVENT

//--- Register event handler for receive data
ONEVENT EVUARTRX       CALL hdlrUartRx

//--- Open comport so that DCD and RI are outputs

UartClose()

PRINT "\nUart State ";uartinfo(0) //will print 0

rv=UartOpenDce(300,1,8,1,1)      //open as DTE at 300 baudrate, odd parity
                                //8 databits, 1 stopbits, cts/rts flow control

PRINT "\nUart State ";uartinfo(0) //will print 1

WAITEVENT     //wait for rx, events
```

UARTINFO is a core subroutine.

UartWrite

This function is used to transmit a string of characters.

UARTWRITE (strMsg)

Function

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

Arguments:

strMsg ***byRef strMsg AS STRING***
The array of bytes to be sent. STRLEN(strMsg) bytes will be written to the local transmit ring buffer. If STRLEN(strMsg) and the return value are not the same then 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, 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

```
DIM rv,cnt
DIM str$

//--- Handler to process receive data

FUNCTION hdlrUartRx()
  PRINT "\nData has arrived"
ENDFUNC 1 //remain blocked in WAITEVENT

FUNCTION hdlrUartTxEty()
  PRINT "\nTx buffer is empty"
ENDFUNC 0 //exit from WAITEVENT

//--- Register event handler for tx buffer empty

ONEVENT EVUARTTXEMPTY CALL hdlrUartTxEty

//--- Register event handler for receive data

ONEVENT EVUARTRX      CALL hdlrUartRx

//--- Open comport so that DCD and RI are outputs

rv=UartOpen(300,0,0,"TO81H")     //open as DTE at 300 baudrate, odd parity
                                //8 databits, 1 stopbits, cts/rts flow control
IF rv==0 THEN
  str$="Hello World"
  cnt = UartWrite(str$)
  if cnt > 0 then
    strshiftleft(str$,cnt)
  endif
ENDIF

WAITEVENT     //wait for rx and txempty events
```

UARTWRITE is a core subroutine.

UartRead

This function is used to read the content of the receive buffer and **append** it to the string variable supplied.

UARTREAD(strMsg)

Function

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 **appended** 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

```
DIM rv,cnt
DIM str$

//--- Handler to process receive data

FUNCTION hdlrUartRx()
  cnt = UartRead(str$)
  PRINT "\nData is ";str$
ENDFUNC 1 //remain blocked in WAITEVENT

FUNCTION hdlrUartTxEty()
  PRINT "\nTx buffer is empty"
ENDFUNC 0 //exit from WAITEVENT

//--- Register event handler for tx buffer empty

ONEVENT EVUARTTXEMPTY CALL hdlrUartTxEty

//--- Register event handler for receive data

ONEVENT EVUARTRX      CALL hdlrUartRx

//--- Open comport so that DCD and RI are outputs

rv=UartOpen(300,0,0,"TO81H")     //open as DTE at 300 baudrate, odd parity
                                 //8 databits, 1 stopbits, cts/rts flow control

//--- Can read from rx buffer anytime, even outside handler

cnt = UartRead(str$)
PRINT "\nData is ";str$

WAITEVENT     //wait for rx and txempty events
```

UARTREAD is a core subroutine.

UartReadMatch

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)

Function

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

```
DIM rv,cnt
DIM str$

//--- Handler to process receive data

FUNCTION hdlrUartRx()
  cnt = UartReadMatch(str$,13) //read up to and including CR
  PRINT "\nData is ";str$
ENDFUNC 1 //remain blocked in WAITEVENT

FUNCTION hdlrUartTxEty()
  PRINT "\nTx buffer is empty"
ENDFUNC 0 //exit from WAITEVENT

//--- Register event handler for tx buffer empty

ONEVENT EVUARTTXEMPTY CALL hdlrUartTxEty

//--- Register event handler for receive data

ONEVENT EVUARTRX      CALL hdlrUartRx

//--- Open comport so that DCD and RI are outputs

rv=UartOpen(300,0,0,"TO81H")     //open as DTE at 300 baudrate, odd parity
                                 //8 databits, 1 stopbits, cts/rts flow control

//--- Can read from rx buffer anytime, even outside handler

cnt = UartRead(str$)
PRINT "\nData is ";str$

WAITEVENT     //wait for rx and txempty events
```

UARTREADMATCH is a core subroutine.

UartFlush

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.

UARTFLUSH(bitMask)

Subroutine

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPEN

Arguments:

bitMask	***byVal bitMask AS INTEGER*** Bit 0 is set to flush the rx buffer and Bit 1 to flush the tx buffer.

Interactive Command: No

Related Commands: UARTOPEN,UARTINFO, UARTCLOSE, UARTWRITE, UARTREAD, UARTREADMATCH, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTGETDSR, UARTSETRTS, UARTSETDCD, UARTBREAK, UARTFLUSH

```
DIM rv

//--- Open comport so that DCD and RI are inputs

rv=UartOpen(300,0,0,"TO81H")     //open as DTE at 300 baudrate, odd parity
                                 //8 databits, 1 stopbits, cts/rts flow control

If rv==0 then
  UartFLUSH(1)    //flush the receive buffer
endif
```

UARTFLUSH is a core subroutine.

UartGetCTS

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()

Function

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

```
DIM rv
DIM mdm

//--- Open comport so that DCD and RI are outputs

rv=UartOpen(300,0,0,"TO81H")     //open as DTE at 300 baudrate, odd parity
                                 //8 databits, 1 stopbits, cts/rts flow control

If rv==0 then
  mdm = UartGetCts()
  PRINT "\nCTS is ";mdm
endif
```

UARTGETCTS is a core subroutine.

UartSetRTS

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)

Subroutine

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

```
DIM rv

//--- Open comport so that DCD and RI are outputs

rv=UartOpen(300,0,0,"TO81H")     //open as DTE at 300 baudrate, odd parity
                                 //8 databits, 1 stopbits, cts/rts flow control
```

```
// RTS output has automatically been asserted

If rv==0 then
  UartSetRts(0)   //has no effect because flow control was enabled
  UartSetRts(1)   //has no effect because flow control was enabled
endif

UartClose()

rv=UartOpenDce(300,1,8,1,0)      //open as DTE at 300 baudrate, odd parity
                                 //8 databits, 1 stopbits, no cts/rts flow control

// RTS output has automatically been asserted

If rv==0 then
  UartSetRts(0)   //RTS will be deasserted
  UartSetRts(1)   //RTS will be asserted
endif
```

UARTSETRTS is a core subroutine.

UartBREAK

This function 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)

Subroutine

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

```
DIM rv

//--- Open comport so that DCD and RI are outputs

rv=UartOpen(300,0,0,"TO81H")     //open as DTE at 300 baudrate, odd parity
                                //8 databits, 1 stopbits, cts/rts flow control

// RI output has automatically been de-asserted

If rv==0 then
  UartBREAK(1)
  PRINT "\nBREAK has been asserted"
  UartBREAK(0)
  PRINT "\nBREAK has been deasserted"
endif
```

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.

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

This function is used to open the main I2C peripheral using the parameters specified.

I2COPEN (nSclSigNo, nSdaSigNo, nClockHz, nCfgFlags, nHande)

Function

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:

nSclSigNo ***byVal nSclSigNo AS INTEGER***
This is the signal number, as detailed in the module pinout table, that must be used as the I2C clock line – SCL.

nSdaSigNo ***byVal nSdaSigNo AS INTEGER***
This is the signal number, as detailed in the module pinout table, that must be used as the I2C data line – SDA.

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

```
DIM rc
DIM handle

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
```

```
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif
```

I2COPEN is a core function.

I2cClose

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)

Subroutine

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

```
DIM rc
DIM handle

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif

I2cClose(handle)  //close the port
I2cClose(handle)  //no harm done doing it again
```

I2CCLOSE is a core subroutine.

I2cWriteREG8

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)

Subroutine

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

```
DIM rc
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif

nSlaveAddr=0x68 :  nRegAddr = 0x34 : nRegVal = 0x42
rc = I2cWriteReg8(nSlaveAddr, nRegAddr, nRegVal)
if rc!= 0 then
  print "\nFailed to Write to slave/register"
endif

I2cClose(handle)  //close the port
```

I2CWRITEREG8 is a core function.

I2cReadREG8

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)

Subroutine

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

```
DIM rc
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif

nSlaveAddr=0x68 :  nRegAddr = 0x34
rc = I2cReadReg8(nSlaveAddr, nRegAddr, nRegVal)
if rc!= 0 then
  print "\nFailed to Write to slave/register"
else
  print "\nValue read from register is "; integer.h' nRegVal
endif

I2cClose(handle)  //close the port
```

I2CREADREG8 is a core function.

I2cWriteREG16

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)

Subroutine

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

```
DIM rc
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif

nSlaveAddr=0x68 :  nRegAddr = 0x34 : nRegVal = 0x4210
rc = I2cWriteReg16(nSlaveAddr, nRegAddr, nRegVal)
if rc!= 0 then
  print "\nFailed to Write to slave/register"
endif

I2cClose(handle)  //close the port
```

I2CWRITEREG16 is a core function.

I2cReadREG16

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)

Subroutine

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

```
DIM rc
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif

nSlaveAddr=0x68 :  nRegAddr = 0x34
rc = I2cReadReg16(nSlaveAddr, nRegAddr, nRegVal)
if rc!= 0 then
  print "\nFailed to Write to slave/register"
else
  print "\nValue read from register is "; integer.h' nRegVal
endif

I2cClose(handle)  //close the port
```

I2CREADREG16 is a core function.

I2cWriteREG32

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)

Subroutine

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

```
DIM rc
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif

nSlaveAddr=0x68 :  nRegAddr = 0x34 : nRegVal = 0x4210FEDC
rc = I2cWriteReg32(nSlaveAddr, nRegAddr, nRegVal)
if rc!= 0 then
  print "\nFailed to Write to slave/register"
endif

I2cClose(handle)  //close the port
```

I2CWRITEREG32 is a core function.

I2cReadREG32

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)

Subroutine

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

```
DIM rc
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif

nSlaveAddr=0x68 :  nRegAddr = 0x34
rc = I2cReadReg32(nSlaveAddr, nRegAddr, nRegVal)
if rc!= 0 then
  print "\nFailed to Write to slave/register"
else
  print "\nValue read from register is "; integer.h' nRegVal
endif

I2cClose(handle)  //close the port
```

I2CREADREG16 is a core function.

I2cWriteRead

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)

Subroutine

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

```
DIM rc
DIM handle
DIM nSlaveAddr
DIM stWrite$, stRead$, nReadLen

rc=I2cOpen(9,8,100000,0,handle)
if rc!= 0 then
  print "\nFailed to open I2C interface with error code ";interger.h' rc
else
  print "\nI2C open success"
endif

//Write 2 bytes and read 0
nSlaveAddr=0x68 :  stWrite = "\34\35" : stRead$="" : nReadLen = 0
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
if rc!= 0 then
  print "\nFailed to WriteRead"
else
  print "\nWrite = ";strhexize$(stWrite$);" Read = ";strhexize$(stRead$)
endif

//Write 3 bytes and read 4
nSlaveAddr=0x68 :  stWrite = "\34\35\43" : stRead$="" : nReadLen = 4
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
if rc!= 0 then
  print "\nFailed to WriteRead"
else
  print "\nWrite = ";strhexize$(stWrite$);" Read = ";strhexize$(stRead$)
endif

//Write 0 bytes and read 8
nSlaveAddr=0x68 :  stWrite = "" : stRead$="" : nReadLen = 8
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
if rc!= 0 then
  print "\nFailed to WriteRead"
else
  print "\nWrite = ";strhexize$(stWrite$);" Read = ";strhexize$(stRead$)
```

```
endif

I2cClose(handle)   //close the port
```

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

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

SPIOPEN (nMode, nClockHz, nCfgFlags, nHande)

Function

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
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

```
DIM rc
DIM handle

rc=SpiOpen(0,1000000,0,handle)
if rc!= 0 then
  print "\nFailed to open SPI interface with error code ";interger.h' rc
else
  print "\nSPI open success"
endif
```

SPIOPEN is a core function.

SpiClose

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)

Subroutine

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

```
DIM rc
DIM handle

rc=SpiOpen(0,1000000,0,handle)
if rc!= 0 then
  print "\nFailed to open SSPI interface with error code ";interger.h' rc
else
  print "\nSPI open success"
endif

SpiClose(handle)  //close the port
SpiClose(handle)  //no harm done doing it again
```

SPICLOSE is a core subroutine.

SpiReadWrite

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$, stRead$)

Subroutine

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

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

```
DIM rc
DIM handle
DIM stWrite$, stRead$
DIM cs_pin
Cs_pin = 14
```

```
rc=SpiOpen(0,1000000,0,handle)
if rc!= 0 then
  print "\nFailed to open SPI interface with error code ";interger.h' rc
else
  print "\nSPI open success"
endif

//enable the chip select to the slave
Gpiowrite(cs_pin,0)

//Write 2 bytes and read 2 at the same time
stWrite = "\34\35" : stRead$=""
rc = SpiReadWrite(stWrite$, stRead$)
if rc!= 0 then
  print "\nFailed to ReadWrite"
else
  print "\nWrite = ";strhexize$(stWrite$);" Read = ";strhexize$(stRead$)
endif

//disable the chip select to the slave
Gpiowrite(cs_pin,1)



SpiClose(handle)  //close the port
```

SPIWRITEREAD is a core function.

SpiWrite

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$)

Subroutine

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

stWrite$	***byRef stWrite$ AS STRING*** This string contains the data that must be written.

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
DIM rc
DIM handle
DIM stWrite$
DIM cs_pin
Cs_pin = 14
```

```
rc=SpiOpen(0,1000000,0,handle)
if rc!= 0 then
  print "\nFailed to open SPI interface with error code ";interger.h' rc
else
  print "\nSPI open success"
endif

//enable the chip select to the slave
Gpiowrite(cs_pin,0)

//Write 2 bytes
stWrite = "\34\35"
rc = SpiWrite(stWrite$)
if rc!= 0 then
  print "\nFailed to Write"
else
  print "\nWrite = ";strhexize$(stWrite$)
endif


//disable the chip select to the slave
Gpiowrite(cs_pin,1)


SpiClose(handle)  //close the port
```

SPIWRITE is a core function.

SpiRead

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)

Subroutine

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

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

```
DIM rc
DIM handle
DIM stRead$
DIM cs_pin
cs_pin = 14

rc=SpiOpen(0,1000000,0,handle)
if rc!= 0 then
  print "\nFailed to open SPI interface with error code ";interger.h' rc
else
  print "\nSPI open success"
endif

//enable the chip select to the slave
Gpiowrite(cs_pin,0)

//Read 2 bytes
rc = SpiRead(stRead$)
if rc!= 0 then
  print "\nFailed to Write"
else
  print "\nRead = ";strhexize$(stRead$)
endif

//disable the chip select to the slave
Gpiowrite(cs_pin,1)


SpiClose(handle)  //close the port
```

SPIREAD is a core function.

Non-Volatile Memory Management Routines

These commands provide access to the non-volatile memory of the module, as well as providing the ability to use non-volatile storage for individual records.

NvRecordGet

NVRECORDGET is used to read the value of a user record as a string from non-volatile memory.

NVRECORDGET (*recnum, strvar*)

FUNCTION

Returns: INTEGER Returns the number of bytes that were read into ***strvar***.

A negative value is returned if an error was encountered as follows:-

-1	***recnum*** is not in valid range or unrecognised
-2	failed to determine the size of the record
-3	The raw record is less than 2 bytes long – suspect flash corruption
-4	insuffucient RAM memory
-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 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 read from the record.

Interactive Command: NO

```
DIM R$
PRINT NVRECORDGET(100,R$)        'print result of operation
PRINT R$                         'print content of record
```

NVRECORDGET is a module function.

NvRecordGetEx

NVRECORDGETX is used to read 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*)

FUNCTION

Returns: INTEGER Returns 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 depends on 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

```
DIM R$
PRINT NVRECORDGETEX(100,R$,"hello")     'print result of operation
PRINT R$                                'print content of record
```

NVRECORDGETEX is a module function.

NvRecordSet

NVRECORDSET is used to write a value to a user record in non-volatile memory.

NVRECORDSET (*recnum, strvar*)

FUNCTION

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: Programmers should minimise the number of writes as each time a record is changed, flash is used up. The flash filing system does not overwrite previously used locations. At some point there will be no more free flash memory and an automatic defragment operation will occur and this operation will take 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

```
DIM W$,R$
DIM RC
W$="HelloWorld"
RC=NVRECORDET(500,W$)
PRINT NVRECORDGETEX(500,R$,"hello")     'print result of operation
```

NVRECORDSET 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

This routine is used to set the function of the gpio pin identified by the nSigNum argument.

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.

GPIOSETFUNC (*nSigNum, nFunction, nSubFunc*)

FUNCTION

Returns: INTEGER Returns a result code. The most typical value is 0x0000, which indicates 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
5 := ANALOG_OUT (not available in the BL600 module)

nSubFunc **byVal nSubFunc INTEGER.**
Configures the pin as follows:

If nFunction := DIGITAL_IN then it consists of 2 bit fields as follows:-
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..7
1 :- When in deep sleep mode, awake when this pin is LOW
2 :- 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
0 := Init output to LOW

1 := Init output to HIGH

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

Interactive Command: NO

```
DIM number

number = gpiosetfunc(3,1,2)     //remove the pull resistor the DIGITAL_IN pin3
number = gpiosetfunc(4,3,0)     //set gpio pin4 as analog in
number = gpiosetfunc(5,1,0x12)  //internal pull up on gpio5 and wake from deep sleep
                                //when there is transition from high to low
```

GPIOSETFUNC is a Module functio

GpioRead

This routine is used to read 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)*

FUNCTION

Returns: INTEGER

Returns the value from the signal. If the signal number is invalid, then it will still return a value and it will be 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

```
DIM signal

signal = gpioread(5)
print signal                      ' the value on gpio pin 3 will be printed.
```

GPIOREAD is a Module function.

GpioWrite

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

GPIOWRITE (*nSigNum, nNewValue*)

SUBROUTINE

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.

Interactive Command: NO

```
DIM signal

gpiowrite(5,1)
signal = gpioread(5)
print signal                          ' the value on gpio pin 3 will be printed.
```

GPIOWRITE is a Module function.

GPIO Events

EVGPIOCHANn	where n=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 and in the case of the BL600 module, N can be 0,1,2 or 3.

GpioBindEvent

This routine is used to bind 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

GPIOBINDEVENT (*nEventNum, nSigNum, nPolarity*)

FUNCTION

Returns: INTEGER

Returns a result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nEventNum	**byVal *nEventNum* INTEGER**. The GPIO event number (in the range of 0 - N) which will result in the event EVGPIOCHANn being thrown to the *smart* BASIC runtime engine.

nSigNum	**byVal *nSigNum* INTEGER**. The signal number as stated in the pinout table of the module.
nPolarity	**byVal *nPolarity* INTEGER**. States the transition as follows:

0	Low to high transition
1	High to low transition
2	Either a low to high or high to low transition

Interactive Command: NO

```
DIM RC

RC = GpioBindEvent(0,20,0)
```

GPIOBINDEVENT is a Module function.

GpioUnbindEvent

This routine is used to unbind the runtime engine event from a level transition.

GPIOUNBINDEVENT (*nEventNum)*

FUNCTION

Returns: INTEGER

Returns a result code. The most typical value is 0x0000, which indicates 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

```
DIM RC

RC = GpioUnBindEvent(0)
```

GPIOUNBINDEVENT 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 typically used to perform frequently repeated tasks within an application and to write event & message handler functions. An application with user routines has optimal modularity enabling reuse of 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 marks the end of 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 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

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.

Events and Messages

EVBLE_ADV_TIMEOUT

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

```
// Example :: EVBLE_ADV_TIMEOUT

//start adverts
rc = BleAdvertStart(adv,addr$,advInt,advTmout,advFilPol)

//handler to service an advert timeout
function HandlerBlrAdvTimOut() as integer
  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 1

OnEvent  EVBLE_ADV_TIMEOUT  call   HandlerBlrAdvTimOut

//wait for events and messages
waitevent
```

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) which contains 2 parameters. The first parameter, to be called **msgID** subsequently, identifies what event got triggered and the second parameter, to be called **msgCtx** subsequestly, 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 all triggers and the associated context parameter is as 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)
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

An example of how this message can be used is as follows:-

```
DIM connHndl  '//global variable to store connection handle
DIM addr$

addr$=""

'//==============================================================================
'// This handler is called when there is a BLE message
'//==============================================================================
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
  select nMsgId
  case 0
    '//print "\nBle Connection ";integer.h' nCtx
    rc = BleAuthenticate(nCtx)
    connHndl = nCtx
  case 1
    '//print "\nBle Disonnection ";integer.h' nCtx
    inconn = 0
    '// restart advertising
    rc = BleAdvertStart(ADV_IND,addr$,ADV_INTERVAL_MS,ADV_TIMEOUT_MS,0)
  case else
    print "\nUnknown Ble Msg"
  endselect
endfunc 1

'//==============================================================================
'// This handler is called when data has arrived at the serial port
'//==============================================================================
function HandlerBlrAdvTimOut() as integer
   print "\nAdvert stoped via timeout"
   '//------------------------------------------------------------
   '//  Switch off the system - requires a power cycle to recover
   '//------------------------------------------------------------
```

```
    rc = SystemStateSet(0)
  endfunc 1

 '// register the handler for all BLE messages
 OnEvent  EVBLEMSG call HandlerBleMsg
 '// register the handler for adv timeouts
 OnEvent  EVBLE_ADV_TIMEOUT  call HandlerBlrAdvTimOut

 '// start adverts
     rc = BleAdvertStart(ADV_IND,addr$,ADV_INTERVAL_MS,ADV_TIMEOUT_MS,0)

 '//wait for event and messages
 WaitEvent
```

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 from where the 'reason of disconnection' can be determined in provided in this document here.

```
// Example :: EVDISCON event

function HandlerDiscon(BYVAL hConn AS INTEGER, BYVAL nRsn AS INTEGER) as integer
  if nRsn == 0x08 then
    print "There was a link supervision timeout"
  endif
endfunc 1

OnEvent  EVDISCON call HandlerDiscon

//wait for events
WaitEvent
```

EVCHARVAL

This event is thrown when a characteristic has been written to by a remote GATT client. It comes with one parameter which is the characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()

```
// Example :: EVCHARVAL charHandle

dim rc
dim hMyChar
dim attr$
dim hSvc

attr$="hello\00worl\64"
rc = BleCharCommit(hSvc,attr$,hMyChar)

function HandlerCharVal(BYVAL charHandle AS INTEGER) as integer
  dim at$
  if charHandle == hMyChar then
      rc = BleCharValueRead(hMyChar,at$)
      print "My characteristic had been written to :";StrHexize$(at$)
  endif
endfunc 1

OnEvent  EVCHARVAL call HandlerCharVal

//wait for events
WaitEvent
```

EVCHARHVC

This event is thrown when an 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 that was returned when the characteristic was registered using the function BleCharCommit() and the second is the new 16 bit value in the updated CCCD attribute.

```
// Example :: EVCHARHVC charHandle
//            EVCHARCCCD charHandle newCccdValue

dim rc
dim hMyChar
dim attr$
dim hSvc

attr$="hello\00worl\64"
rc = BleCharCommit(hSvc,attr$,hMyChar)

//handler to service characteristic value confirmation from gatt client
function HandlerCharHvc(BYVAL hChar AS INTEGER) as integer
  if hChar == hMyChar then
    print "\nGot confirmation to recent indication"
  else
    print "\nGot confirmation to some other indication"; hChar
  endif
endfunc 1

//handler to service writes to CCCD which tells us we can indicate or not
function HandlerCharCccd(BYVAL hChar AS INTEGER, BYVAL nVal AS INTEGER) as integer
  if hChar == hMyChar then
    //The following if statment to convert to 1 is only so that we can submit
    //this app to the regression test
    if nVal & 0x02 then
      print "\nIndications have been enabled by client"
      attr$="hello"
      rc = BleCharValueIndicate(hMyChar,attr$)
      if rc != 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

//install a handler for writes to characteristic values
OnEvent  EVCHARCCCD      call HandlerCharCccd
OnEvent  EVCHARHVC       call HandlerCharHvc

//wait for events and messages
waitevent
```

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 charHandle newSccdValue

dim rc
dim hMyChar
dim attr$
dim hSvc

attr$="hello\00worl\64"
rc = BleCharCommit(hSvc,attr$,hMyChar)

//handler to service writes to CCCD which tells us we can indicate or not
function HandlerCharSccd(BYVAL hChar AS INTEGER, BYVAL nVal AS INTEGER) as integer
  if hChar == hMyChar then
    //The following if statment to convert to 1 is only so that we can submit
    //this app to the regression test
    if nVal & 0x01 then
      print "Broadcasts have been enabled by client"
    else
      print "Broadcasts have been disabled by client"
    endif
  else
    print "This is for some other characteristic"
  endif
endfunc 1

//install a handler for writes to characteristic values
OnEvent  EVCHARCCCD       call      HandlerCharSccd

//wait for events and messages
waitevent
```

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 a composite value which consists of the 16 bit UUID of the descriptor that got updated and the instance. The instance is to cater for the fact that some descriptors can exist multiple times in a characteristic

```
// Example :: EVCHARDESC charHandle compValue

dim rc
dim hMyChar
dim attr$
dim hSvc

attr$="hello\00worl\64"
rc = BleCharCommit(hSvc,attr$,hMyChar)

//handler to service writes to descriptors by a gatt client
function HandlerCharDesc(BYVAL hChar AS INTEGER, BYVAL nComp AS INTEGER) as integer
  dim instnc,nUuid
  instnc = nComp >> 16
  nUuid  = nComp & 0xFFFF
  if hChar == hMyChar then
    rc = BleCharDescRead(hChar,nUuid,instnc,attr$)
    if rc ==0 then
      print "\nNew value for desriptor "; nUuid; " is ";StrHexize(attr$)
    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

//wait for events and messages
waitevent
```

EVVSPRX

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

```
// Example :: EVVSPRX
//   echos incoming data back to the peer

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,0,hndl)

//handler to service data arriving on VSP
function HandlerBleVSpRx() as integer
  //echo the data back
  dim n,rx$
  n = BleVSpRead(rx$,20)
  n = BleVSpWrite(rx$)
endfunc 1

OnEvent  EVVSPRX  call HandlerBleVSpRx

//wait for events and messages
waitevent
```

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.

```
// Example :: EVVSPTXEMPTY

dim tx$

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,0,hndl)

//handler to service VSP tx buffer is empty
function HandlerVSpTxEmpty() as integer
  //echo the data back
  tx$="this is more data"
  n = BleVSpWrite(tx$)
endfunc 1

OnEvent  EVVSPTXEMPTY  call HandlerVSpTxEmpty

tx$="send this data"
n = BleVSpWrite(tx$)

//wait for events and messages
waitevent
```

EVNOTIFYBUF

When in a connection and attribute data is sent to the GATT Client using a notify procedure (for example using the function BleCharValueNotify() 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 will fail with a resultcode 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.

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.

```
// Example :: EVNOTIFYBUF

dim dt$
dim rc
dim ntfyEnabled   //will get set to not0 when the appropriate CCCD is updated

//Routine to send data to client using notifies
sub SendData()
  if ntfyEnabled then
    //send as many notifies as possible
    do
      tx$="SomeData"
      rc = BleCharValueNotify(tx$)
    until (rc != 0) //rc==0x6803 when no transmission buffers left
  endif
endsub

//handler to send more notifies as there are more transmission buffers
function HandlerNotifyBuf() as integer
    //Send as much data as possible
    SendData()
endfunc 1

//handler for generic BLE messages
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
  select nMsgId
  case BLE_EVBLEMSGID_CONNECT
    //Send as much data as possible
    SendData()
  case else
    //ignore this message
  endselect
endfunc 1

OnEvent  EVBLEMSG     call HandlerBleMsg
OnEvent  EVNOTIFYBUF  call HandlerNotifyBuf

//wait for events and messages
waitevent
```

Miscellaneous Functions

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

BleTxPowerSet

This function is used to set the power of all packets that are transmitted subsequently, it is advisable to recreate the advert packet if this new tx power is to be reflected in advertisement packets.

This function is also very useful to temporarily set the power to the lowest value possible so that a pairing is expedited in the smallest bubble of space.

BLETXPOWERSET(nTxPower)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nTxPower **byVal *nTxPower* AS INTEGER.**
Specifies the new transmit power in dBm units to be used for all subsequent tx packets and valid values are :-

4	Maximum
0	
-4	
-8	
-12	
-16	
-20	
-30 or -40	*depends on software stack from vendor*
<= -50	Whisper Mode

Interactive Command: NO

```
DIM RC

RC = bletxpowerset(0)                        'The transmitted power is set to 0 dBm
```

BLETXPOWERSET is an extension function.

BleConfigDcDc

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

BLECONFIGDCDC(nNewState)

SUBROUTINE

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 be able to get access to this information. It is available at https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=245130

BleAdvertStart

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 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.

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

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

If a 0x6A01 resultcode is received it implies whitelist has been enabled but the Flags AD in the advertising report is set for Limited and/or General Discoverability. The solution is to resubmit a new advert report which is made up so that the nFlags argument to BleAdvRptInit() function is 0.
The BT 4.0 spec disallows discoverability when a whitelist is enabled during advertisement see Volume 3, Sections 9.2.3.2 and 9.2.4.2.

Arguments:

nAdvType **byVal *nAdvType* AS INTEGER.**
Specifies the advertisement type as follows:

0	ADV_IND	invites connection requests
1	ADV_DIRECT_IND	invites connection from addressed device
2	ADV_SCAN_IND	invites scan request for more advert data
3	ADV_NONCONN_IND	will not accept connections or active scans

***peerAddr$*** **byRef *peerAddr$* AS STRING**
It can be an empty string that is omitted if the advertisement type is not ADV_DITRECT_IND.
This parameter is only required when nAdvType == 1

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.

nFilterPolicy **byVal *nFilterPolicy* AS INTEGER.**

Specifies the filter policy as follows:

0	Filter Policy	Any
1	Filter Policy	Filter Scan Request
2	Filter Policy	Filter Connection Request
3	Filter Policy	Both

If the filter policy is not 0, then the whitelist is filled with all the addresses of all the devices in the trusted device database.

Interactive Command: NO

```
DIM ReturnCode
DIM Adr$

Adr$=""
ReturnCode = bleadvertstart(0,Adr$,25,60000,0) 'The advertising interval is set to 25
                                               'milliseconds. The module will stop
                                               'advertising after 60000 ms (1 minute)
```

BLEADVERTSTART is an extension function.

BleAdvertStop

This function causes the BLE module to stop advertising.

BLEADVERTSTOP ()

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments None

Interactive Command: NO

```
DIM ReturnCode

ReturnCode = bleadvertstop()           'Causes the BLE module to stop advertising
```

BLEADVERTSTOP is an extension function.

BleAdvRptInit

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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

advRpt$	byRef ***advRpt$*** AS STRING. This will contain an advertisement report.
nFlagsAD	byVal ***nFlagsAD*** AS INTEGER. Specifies the flags AD bits where bit 0 is set for limited discoverability and bit 1 is set for general discoverability. Bit 2 will be forced to 1 and bits 3 & 4 will be forced to 0. Bits 3 to 7 are reserved for future use by the BT SIG and must be set to 0. **Note that 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

```
DIM RC,advRpt$,scnRpt$,discoverableMode, advAppearance,MaxDevName

ad$=""
scnRpt$=""
discoverableMode = 0
advAppearance = 1
nMaxDevName = 10

RC = bleadvrptinit(advRpt$, discoverableMode, advAppearance, MaxDevName)
RC = bleadvrptscommit(advRpt$,scnRpt$)
```

BLEADVRPTINIT is an extension function.

BleScanRptInit

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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

scanRpt	**byRef *scanRpt* *ASSTRING*.** This will contain a scan report.

Interactive Command: NO

```
DIM RC,advRpt$,scnRpt$,discoverableMode, advAppearance,MaxDevName

ad$=""
scnRpt$=""
discoverableMode = 0
advAppearance = 1
nMaxDevName = 10

RC = bleadvrptinit(advRpt$, discoverableMode, advAppearance, MaxDevName)
RC = blescanrptinit(scnRpt$)
RC = bleadvrptscommit(advRpt$,scnRpt$)
```

BLESCANRPTINIT is an extension function.

BleAdvRptAddUuid16

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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

AdvRpt	**byRef *AdvRpt AS* STRING**. The advert report onto which the 16 bit uuids AD record is added.
Uuid1	***byVal uuid1 AS INTEGER*** Uuid in the range 0 to FFFF, if 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

```
DIM RC,advRpt$,discoverableMode, advAppearance,MaxDevName

discoverableMode = 0
advAppearance = 1
nMaxDevName = 10

RC = bleadvrptinit(advRpt$, discoverableMode, advAppearance, MaxDevName)

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

RC = bleadvrptadduuid(advRpt$,0x180F,0x180A, -1, -1, -1, -1)

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

BLEADVRPTADDUUID16 is an extension function.

BleAdvRptAddUuid128

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

BLEADVRPTADDUUID128 (advRpt, nUuidHandle)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

advRpt	**byRef *AdvRpt AS* STRING**. The advert report into which the 128 bit uuid AD record is to be added.
nUuidHandle	***byVal nUuidHandle AS INTEGER*** This is handle to a 128 bit uuid which was obtained using say the function BleHandleUuid128() or some other function which returns one, like BleVSpOpen()

Interactive Command: NO

```
// Example :: BLEADVRPTADDUUID128

dim tx$
dim scRpt$,adRpt$,addr$

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,0,hndl)

//Advertise the service in a scan report
rc = BleAdvRptAddUuid128(scRpt$,hndl)
adRpt$=""
rc = BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""  //because we are not doing a DIRECT advert
rc = BleAdvertStart(0,addr$,100000,300000,0)
//Now advertising so can be connectable

//wait for events and messages
waitevent
```

BLEADVRPTADDUUID128 is an extension function.

BleAdvRptAppendAD

This function is used to add 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 in the range 0 to 255 and DATA is a sequence of octets.

BLEADVRPTAPPENDAD (advRpt, nTag, stData$)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates 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 which can be a maximum of 31 bytes long.

Interactive Command: NO

```
DIM RC,advRpt$,ad$

RC = BleScanRptInit(advRpt$)

ad$="\01\02\03\04"

RC = BleAdvRptAppendAD(advRpt$,0x31,ad$) '6 bytes will be used up in the report
```

BLEADVRPTAPPENDAD is an extension function.

BleAdvRptsCommit

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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

advRpt	**byRef *advRpt* AS STRING.** The most recent advert report.

scanRpt	**byRef *scanRpt* AS STRING.** The most recent scan report.

Note: If any one of the two strings is not valid then the call will be aborted without updating the other report even if this other report is valid.

Interactive Command: NO

```
DIM RC,advRpt$,discoverableMode,advAppearance,MaxDevName
DIM UuidBatteryService, UuidDeviceInfoService

ad$=""
scRpt$=""
discoverableMode = 0
advAppearance = 1
nMaxDevName = 10
UuidBatteryService = 0x180F
UuidDeviceInfoService = 0x180A


RC = bleadvrptinit(advRpt$, discoverableMode, advAppearance, MaxDevName)

RC = bleadvrptadduuid(UuidBatteryService,UuidDeviceInfoService, -1, -1, -1, -1)

RC = bleadvrptscommit(ad$, scRpt$)

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

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 message IDs that are relevant are (0), (1), (14), (15), (16), (17), (18) and (20) as follows:-

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

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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nConnHandle **byVal *nConnHandle* AS INTEGER.**
Specifies the handle of the connection that needs to be disconnected.

Interactive Command: NO

```
DIM RC,connHandle

RC = bledisconnect(connHandle)
```

BLEDISCONNECT is an extension function.

BleSetCurConnParms

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 then an EVBLEMSG message with msgid = 15 will be thrown to the *smart* BASIC runtime engine

BLESETCURCONNPARMS (nConnHandle, nMinInt, nMaxInt, nSuprTout, nSlaveLatency)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nConnHandle	**byVal *nConnHandle* AS INTEGER.** Specifies the handle of the connection that needs to have the connection parameters changed
nMinInt	**byVal *nMinInt* AS INTEGER.** The minimum acceptable connection interval in microseconds
nMaxInt	**byVal *nMaxInt* AS INTEGER.** The maximum acceptable connection interval in microseconds
nSuprTout	**byVal *nSuprTout* AS INTEGER.** The link supervision timeout for the connection. It should be a value which is more than the slave latency times the actual granted connection interval.
nSlaveLatency	**byVal *nSlaveLatency* AS INTEGER.** This is 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 used to reduce power usage in a peripheral device and yet have a short latency. Generally a slave will reduce power usage by having as large a connection interval as possible. But this means the latency is equivalent to that connection interval. To mitigate this, the peripheral can reduce the connection interval to a much reduced value and then have a non-zero slave latency.
For example, a keyboard could set the connection interval to 1000msec and slave latency to 0. In this case key presses will get reported to the central device once a second which will lead to a bad user experience. So instead, the connection interval can be set to say 50msec and slave latency to 19. This means that in the event there are no key presses the power usage is the same as the first example because (19+1) times 50 is 1000. When a key is pressed, the peripheral knows that the central device will poll within 50msec and so it will be able to send that keypress with a latency of 50msec.
A connection interval of 50 and slave latency of 19 means the slave is allowed to NOT ack a poll for up to 19 poll messages from the central device.

Interactive Command: NO

```
DIM RC,connHandle


function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
 dim intrvl,sprvto,slat

 select nMsgId
  case 0 //BLE_EVBLEMSGID_CONNECT
    print " --- Hrs Connect : ",nCtx
    RC=BleGetCurConnParms(nCtx,intrvl,sprvto,slat)
    if rc==0 then
      print "Conn Interval",intrvl
      print "Conn Supervision Timeout",sprvto
      print "Conn Slave Latency",slat
```

```
     Endif

   case 1 //BLE_EVBLEMSGID_DISCONNECT
     print " --- Hrs Disconnect : ",nCtx

    case 14 //BLE_EVBLEMSGID_CONN_PARMS_UPDATE
     RC=BleGetCurConnParms(nCtx,intrvl,sprvto,slat)
     if rc==0 then
       print "Conn Interval",intrvl
       print "Conn Supervision Timeout",sprvto
       print "Conn Slave Latency",slat
     Endif

    case 15 //BLE_EVBLEMSGID_CONN_PARMS_UPDATE_FAIL
      print " ??? Conn Parm Negotiation FAILED"

    case else
      print "Unknown Ble Msg"
    endselect
  endfunc 1

  OnEvent  EVBLEMSG           call HandlerBleMsg

 //request connection interval in range 50ms to 75ms and link supervision timeout of
 //4seconds with a slave latency of 19

 RC = bleSetCurConnParms(connHandle, 50000,75000,4000000,19)

 waitevent
```

BLESETCURCONNPARMS is an extension function.

BleGetCurConnParms

This function is used to get 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, nInterval, nSuprTout, nSlaveLatency)

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nConnHandle	**byVal *nConnHandle* AS INTEGER.** Specifies the handle of the connection that needs to have the connection parameters changed
nInterval	**byRef *nMinInt* AS INTEGER.** The current connection interval in microseconds
nSuprTout	**byRef *nSuprTout* AS INTEGER.** The current link supervision timeout for the connection.

nSlaveLatency	**byRef *nSlaveLatency* AS INTEGER.** This is the current number of connection interval polls that the peripheral may ignore. This times the connection interval will not be greater than the link supervision timeout.

Note:
Slave latency is a mechanism used to reduce power usage in a peripheral device and yet have a short latency. Generally a slave will reduce power usage by having as large a connection interval as possible. But this means the latency is equivalent to the connection interval. To mitigate this, the peripheral can reduce the connection interval to a much reduced value and then have a non-zero slave latency.
For example, a keyboard could set the connection interval to 1000msec and slave latency to 0. In this case key presses will get reported to the central device once a second which will lead to a bad user experience. So instead, the connection interval can be set to say 50msec and slave latency to 19. This means that in the event there are no key presses the power usage is the same as the first example because (19+1) times 50 is 1000. When a key is pressed, the peripheral knows that the central device will poll within 50msec and so it will be able to send that keypress with a latency of 50msec.
A connection interval of 50 and slave latency of 19 means the slave is allowed to NOT ack a poll for up to 19 poll messages from the central device.

Interactive Command: NO

```
DIM RC,connHandle


function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
 dim intrvl,sprvto,slat

 select nMsgId
  case 0 //BLE_EVBLEMSGID_CONNECT
    print " --- Hrs Connect : ",nCtx
    RC=BleGetCurConnParms(nCtx,intrvl,sprvto,slat)
    if rc==0 then
      print "Conn Interval",intrvl
      print "Conn Supervision Timeout",sprvto
      print "Conn Slave Latency",slat
    Endif

  case 1 //BLE_EVBLEMSGID_DISCONNECT
    print " --- Hrs Disconnect : ",nCtx

   case 14 //BLE_EVBLEMSGID_CONN_PARMS_UPDATE
    RC=BleGetCurConnParms(nCtx,intrvl,sprvto,slat)
    if rc==0 then
      print "Conn Interval",intrvl
      print "Conn Supervision Timeout",sprvto
      print "Conn Slave Latency",slat
    Endif

   case 15 //BLE_EVBLEMSGID_CONN_PARMS_UPDATE_FAIL
     print " ??? Conn Parm Negotiation FAILED"

   case else
     print "Unknown Ble Msg"
   endselect
 endfunc 1

 OnEvent  EVBLEMSG           call HandlerBleMsg

//request connection interval in range 50ms to 75ms and link supervision timeout of
//4seconds with a slave latency of 19

RC = bleSetCurConnParms(connHandle, 50000,75000,4000000,19)

waitevent
```

BLEGETCURCONNPARMS is an extension function.

Security Manager Functions

This section describes routines which are used to manage all aspects related to 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 the msgID :-

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.

When msgId 9 is sent the msgCtx parameter contains the passkey to display which will be a value in the range 0 to 999999. It is advisable to display the number with leading 0's so that the passkey is always displayed as a 6 digit decimal number.

To submit a passkey, use the function BLESECMNGRPASSKEY.

BleSecMngrPasskey

This function is used to submit 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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

connHandle **byVal connHandle AS INTEGER**.
This is the connection handle as received via the EVBLEMSG event with msgId set to 0.

nPassKey **byVal *nPassKey* AS INTEGER.**
This is the passkey to submit to the stack. Submit a value outside the range 0 to 999999 to reject the pairing.

Interactive Command: NO

```
DIM rc
DIM connHandle
```

```
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
  select nMsgId
  case BLE_EVBLEMSGID_CONNECT
    connHandle =nCtx
    DbgMsgVal("Ble Connection ",nCtx)


  case BLE_EVBLEMSGID_AUTH_KEY_REQUEST
    DbgMsgVal(" +++ Auth Key Request, type=",nCtx)
    rc=BleSecMngrPassKey(connHandle,123456)  '//key is 123456

  case else
    DbgMsg("Unknown Ble Msg" )
  endselect
endfunc 1

OnEvent  EVBLEMSG           call HandlerBleMsg

waitevent
```

BLESECMNGRPASSKEY is an extension function.

BleSecMngrKeySizes

This function is used to set the minimum and maximum long term encryption key size requirement for subsequent pairings.

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

BLESECMNGRKEYSIZES(nMinKeysize, nMaxKeysize)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nMinKeysiz	**byVal *nMinKeysiz* AS INTEGER**. The minimum key size. The range of this value is between 7 and 16.
nMaxKeysize	**byVal *nMaxKeysize* AS INTEGER.** The maximum key size. The range of this value is between nMinKeysize and 16.

Interactive Command: NO

```
DIM RC

RC = blemngrkeysizes(8,15)                  'The key size requirement is set between
                                            '8 and 15 seconds
```

BLESECMNGRKEYSIZES is an extension function.

BleSecMngrIoCap

This function is used to set 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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nIoCap **byVal *nIoCap AS* INTEGER**.
The user i/o capability for all subsequent pairings.

0	None also known as 'Just Works' (unauthenticated pairing)
1	Display with Yes/No input capability (authenticated pairing)
2	Keyboard Only (authenticated pairing)
3	Display Only (authenticated pairing – if other end has input cap)
4	Keyboard only (authenticated pairing)

Interactive Command: NO

```
DIM RC
RC = blesecmngriocap(0)     'Select 'just works' pairing
```

BLESECMNGRIOCAP is an extension function.

BleSecMngrBondReq

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)**

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nBondReq	**byVal *nBondReq AS* INTEGER**. 0 Disable 1 Enable

Interactive Command: NO

```
DIM RC,ConnHndl

RC = BleSecMngrBondReq(0)     'Disable
RC = BleAuthenticate(ConnHndl)
```

BLESECMNGRBONDREQ is an extension function.

BleAuthenticate

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

BLEAUTHENTICATE (nConnCtx)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nConnCtx	byVal ***nConnCtx*** AS INTEGER. This is the context value provided in the BLEMSG(0) message which informed the stack that a connection had been established.

Interactive Command: NO

```
DIM RC, conhndl

RC = bleAuthenticate(conhndl)     'Request the master to initiate a pairing.
```

BLEAUTHENTICATE is an extension function.

Bespoke 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 and hopefully attain a quicker comprehension.

To help understand the terms Service and Characteristic better, think of a Characteristic as a container (or a pot) of data were the container (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 . You should note that 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 128bit 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, and also further notes :-

> *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.

> 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 16bit (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 16bit 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.
>
> SmartBASIC 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, as by definition, interoperability is restricted to just the provider and implementor.

A Service is an abstraction of some collectivised functionality which if broken down further into smaller components would cease to provide the behaviour that it claims to want to describe. A couple of examples in the BLE domain that have been adopted by the Bluetooth SIG are Blood Pressure Service and Heart Rate Service and each have sub-companents 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 can be viewed online at:-
http://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx and Laird recommend 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.

You should note that these Service are also assigned 16 bit UUIDs (value 0x18xx) and are referenced in some of the smartBASIC API functions described in this section.

If the developer does intend to create a custom Service, 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 16bit value which is not going to coincide with any adopted values to minimise confusion. Selecting a similar value is possible and legal given the base UUID is different. The recommendation is just for ease of maintenance.

Services, as has been described above, are a collection of one or more Characteristics.
A list of all adopted charactersitics can be viewed online on the Bluetooth SIG website 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.

If the developer does intend to create a custom Characteristic, and adopt the recommendation, stated above, of a single long 16 byte base UUID, so that the characteristic 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 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 smartBASIC 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.

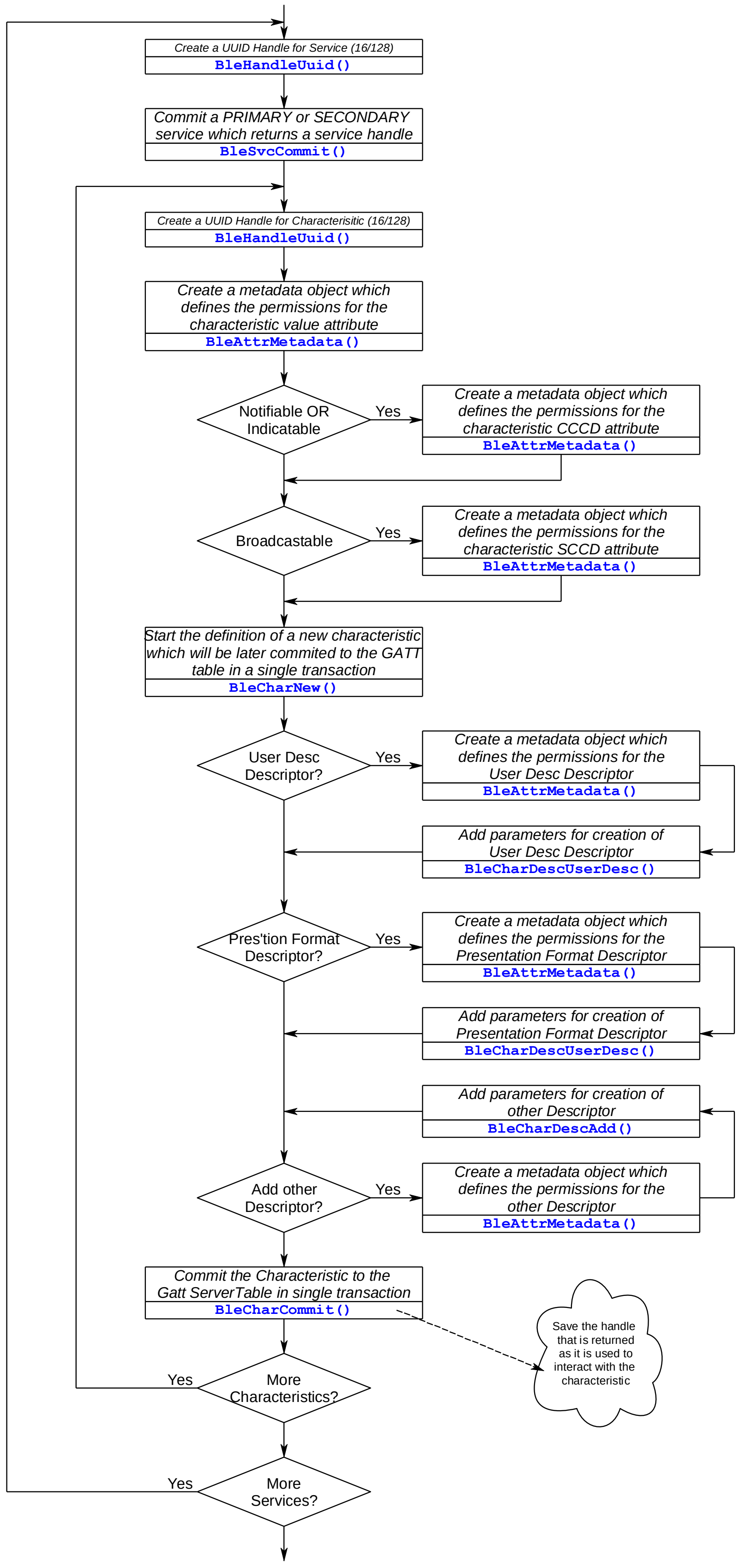
The GATT Client (remote end of the wireless connection) needs to see those carrier bags to determine the groupings and once it has identified the pots it will only need to keep a list of references to the pots it is interested in. Once that list is 'created' at the client end it can 'throw away the carrier 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 he scenes in smartBASIC does not care.

Therefore from the smartBASIC apps developer's **logical** perspective a GATT table looks nothing like the table that is presented in various BLE literatures. Instead the GATT table is <u>purely and simply</u> 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 smartBASIC 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 smartBASIC 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 smartBASIC functions are used to register Services and Characteristics is illustrated in the flowchart on the right and sample code follows on the next page.


Create a UUID Handle for Service (16/128)
BleHandleUuid()
Commit a PRIMARY or SECONDARY service which returns a service handle
BleSvcCommit()
Create a UUID Handle for Characterisitic (16/128)
BleHandleUuid()
Create a metadata object which defines the permissions for the characteristic value attribute
BleAttrMetadata()
Notifiable OR Indicatable
Yes
Create a metadata object which defines the permissions for the characteristic CCCD attribute
BleAttrMetadata()
Broadcastable
Yes
Create a metadata object which defines the permissions for the characteristic SCCD attribute
BleAttrMetadata()
Start the definition of a new characteristic which will be later commited to the GATT table in a single transaction
BleCharNew()
User Desc Descriptor?
Yes
Create a metadata object which defines the permissions for the User Desc Descriptor
BleAttrMetadata()
Add parameters for creation of User Desc Descriptor
BleCharDescUserDesc()
Pres'tion Format Descriptor?
Yes
Create a metadata object which defines the permissions for the Presentation Format Descriptor
BleAttrMetadata()
Add parameters for creation of Presentation Format Descriptor
BleCharDescUserDesc()
Add parameters for creation of other Descriptor
BleCharDescAdd()
Add other Descriptor?
Yes
Create a metadata object which defines the permissions for the other Descriptor
BleAttrMetadata()
Commit the Characteristic to the Gatt ServerTable in single transaction
BleCharCommit()
Save the handle that is returned as it is used to interact with the characteristic
More Characteristics?
Yes
More Services?
Yes

```
//------------------------------------------------------------------------------
//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

//---Register Service 1
rc = BleSvcCommit(BLE_SERVICE_PRIMARY,BleHandleUuid16(0x180D),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
rc = BleCharNew(chProp, BleHandleUuid16(0x2A37),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
rc = BleCharNew(chProp, BleHandleUuid16(0x2A37),mdAttr,mdCccd,mdSccd)
attr$="\00\00"
rc = BleCharCommit(hSvc,attr$,hChar21)


//---Register Service 2  (can now reuse the service handle)
rc = BleSvcCommit(BLE_SERVICE_PRIMARY, BleHandleUuid16(0x1856),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
rc = BleCharNew(chProp, BleHandleUuid16(0x2A54),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 remot 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 = BleCharReadValue(hChar11,attr$)
    print "Svc1/Char1 has been writen with = ";attr$

  endif
endfunc 1

//enable characteristic value write handler
OnEvent  EVCHARVAL          call HandlerCharVal

//wait for events
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 CHARHVC 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 will now describe in detail, all the smartBASIC 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

BleHandleUuid16

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.

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)

FUNCTION

Returns:	INTEGER A handle representing the shorthand UUID and shall never be zero which is an invalid UUID handle.
Arguments:	
nUuid16	**byVal *nUuid16* AS INTEGER** A uuid value in the range 0 t0 65535 which will be treated as an offset into the Bluetooth SIG 128bit base UUID.

Interactive Command: NO

```
// Example :: BleHandleUuid16()

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"
endif
```

BLEHANDLEUUID16 is an extension function.

BleHandleUuid128

This function takes a 16 byte string and converts it into a 32 bit integer handle which consists of a 16 bit (2 byte) offset into a new 128 bit base UUID created from the 16 byte string that was supplied.

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, apart from all 0's which represents an invalid uuid handle.

Please ensure that you use a 16 byte UUID that has been generated using a random number gernerator with suffienct 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$)

FUNCTION

Returns: INTEGER

A handle representing the shorthand UUID and coud be zero which is an invalid UUID handle and that will be returned in the event that there is no spare RAM memory to save the 16 byte base or if more than 253 cusom base uuid's 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()

dim uuid$
dim hUuidCustom

//creat 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"
endif
// hUuidCustom now references an object which points to
// a base uuid = ced9d91366924a1287d56f2700002b2a  (note 0's in byte position 13/14)
// and an offset = 0x6476
```

BLEHANDLEUUID128 is an extension function.

BleHandleUuidSibling

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 which 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)

FUNCTION

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 128bit base UUID referenced by nUuidHandle.

Interactive Command: NO

```
// Example :: BleHandleUuidSibling()

dim uuid$
dim hUuidCustom1
dim hUuidCustom2  //will have the same base uuid as hUuidCustom1

//creat a custom uuid for my ble widget
uuid$ = "ced9d91366924a1287d56f2764762b2a"
uuid$ = StrDehexize$(uuid$)
hUuidCustom1 = BleHandleUuid128(uuid$)
if hUuidCustom1 == 0 then
  print "\nFailed to create a handle"
endif
// hUuidCustom1 now references an object which points to
// a base uuid = ced9d91366924a1287d56f2700002b2a  (note 0's in byte position 13/14)
// and an offset = 0x6476
hUuidCustom2 = BleHandleUuidSibling(hUuidCustom1,0x1234)
if hUuidCustom2 == 0 then
  print "\nFailed to create new sibling handle"
endif
// hUuidCustom2 now references an object which also points to
// the base uuid = ced9d91366924a1287d56f2700002b2a  (note 0's in byte position 13/14)
// and has the offset = 0x1234
```

BLEHANDLEUUIDSIBLING is an extension function.

BleSvcCommit

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.

BLESVCCOMMIT (nSvcType, nUuidHandle, hService)

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.
Arguments:	
nSvcType	**byVal *nSvcType* AS INTEGER** This will be 0 for a SECONDARY service and 1 for a PRIMARY service and all other values are reserved for use and will result in this function failing with an appropriate resultcode.
nUuidHandle	**byVal *nUuidHandle* AS INTEGER** This is 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 precreated 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 which is then subsequently used when adding Characteristics to the GATT table. If the function fails to install the Service attribure for any reason this variable will contain 0 and the returned resultcode will be non-zero.

Interactive Command: NO

```
// Example :: BleSvcCommit()
#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

rc = BleSvcCommit(BLE_SERVICE_PRIMARY,BleHandleUuid16(0x1809),hHtsSvc)
// Add one or more characteristics
dim hHtsMeas
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
atr$="some value"
rc = BleCharCommit(hHtsSvc,attr$,hHtsMeas)

//----------------------------------------------------------------------------
//Create a Battery PRIMARY service attribute which has a uuid of 0x180F
//----------------------------------------------------------------------------
dim hBatSvc    //composite handle for batteru primary service
               //or we could have reused nHtsSvc

rc = BleSvcCommit(BLE_SERVICE_PRIMARY,BleHandleUuid16(0x180F),hBatSvc)
// Add one or more characteristics
dim hHtsMeas
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
atr$="\50"  //80% battery level
rc = BleCharCommit(hBatSvc,attr$,hHtsMeas)
```

BLESVCCOMMIT is an extension function.

BleSvcAddIncludeSvc

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 of Characteristics to be shared by multiple primary services. This is especially 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 to be part of many PRIMARY services, will install that Characteristic in a SECONDARY service (see BleSvcCommit()) and then use the function defined in this section to add it to all the PRIMARY services that want to have that characteristic as part of it's group.

Note it is possible to include a service which itself is also a PRIMARY or SECONDARY service, which in turn can include further PRIMARY or SECONDARY services. There is no logical restriction to the nesting of includes as long as there is no 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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

hService **byVal hService AS INTEGER**
This argument will contain a handle that was previously created using the function BleSvcCommit().

Interactive Command: NO

```
// Example :: BleSvcAddIncludeSvc()

#define BLE_SERVICE_SECONDARY                          0
#define BLE_SERVICE_PRIMARY                            1

//-----------------------------------------------------------------------------
//Create a Battery PRIMARY secondary service attribure which has a uuid of 0x180F
//-----------------------------------------------------------------------------
dim hBatSvc    //composite handle for batteru primary service
               //or we could have reused nHtsSvc

rc = BleSvcCommit(BLE_SERVICE_SECONDARY,BleHandleUuid16(0x180F),hBatSvc)
// Add one or more characteristics
dim hHtsMeas
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
atr$="\50"  //80% battery level
rc = BleCharCommit(hBatSvc,attr$,hHtsMeas)
//-----------------------------------------------------------------------------
//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
rc = BleSvcAddIncludeSvc(hHtsSvc)

// Add one or more characteristics
dim hHtsMeas
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
atr$="some value"
rc = BleCharCommit(hHtsSvc,attr$,hHtsMeas)
```

XXXXX is an extension function.

BleAttrMetadata

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 that is then used to submit 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 <u>value attribute</u> have to be specified. Note that this is properties for the attribute and not properties for the Characteristic container as a whole, which also exist and needs to also be specified but that is done in a different manner as explained later.

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

If the Characteristic is capable of notification and indication, then it implies that the client needs to be able to enable or disable that. This is done through a Characteristic Descriptor which is also another attribute and so that 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 and 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 it's value data in advertisements and for the gatt client to be able to control that, there is yet another type of Characteristc Descriptor and that will also also need 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 and 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 then a seperate 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 is where to get that 'note' written.

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

FUNCTION

Returns: INTEGER

A 32 bit integer which is an opaque data object to be used in subsequent calls when adding Characteristics to a GATT table.

Arguments:

nReadRights **byVal *nReadRights* AS INTEGER**
This specified 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)

Please note in early releases of the firmware 4 & 5 is not available

nWriteRights **byVal *nWriteRights* AS INTEGER**
This specified 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)

Please note in early releases of the firmware 4 & 5 is not available

nMaxDataLen **byVal *nMaxDataLen* AS INTEGER**
This specified the maximum data length of the VALUE attribute which can be anything from 1 to 512 according to the BT specification but 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**
This can be 0 to force the GATT client to write all bytes atomically in one operation or non-zero to allow the client to write into one or more bytes at any offset.

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()

#define BLE_ATTR_ACCESS_NONE                            0
#define BLE_ATTR_ACCESS_OPEN                            1
#define BLE_ATTR_ACCESS_ENC_NO_MITM                     2
#define BLE_ATTR_ACCESS_ENC_WITH_MITM                   3
#define BLE_ATTR_ACCESS_SIGNED_NO_MITM                  4
```

```
#define BLE_ATTR_ACCESS_SIGNED_WITH_MITM                5

#define BLE_CHAR_METADATA_ATTR_NOT_PRESENT              0

#define MAX_HTM_LEN                                    20

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 chProp   //properties for this Characteristic

//++++
// 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(BLE_ATTR_ACCESS_NONE,BLE_ATTR_ACCESS_NONE,MAX_HTM_LEN,0,rc)
//There is a CCCD in this characteristic
mdCccd=BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_ENC_NO_MITM,2,0,rc)
//There is no SCCD in this characteristic
mdSccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
//Create the Characteristic object
chProp = BLE_CHAR_PROPERTIES_INDICATE
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdVal,mdCccd,mdSccd)
```

BLEATTRMETADATA is an extension function.

BleCharNew

When a Characteristic needs to be added to a GATT table, multiple attribute 'objects' need to be precreated and then 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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates 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
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 singned writes
7	Reliable writes

nUuidHandle **byVal *nUuidHandle* AS INTEGER**
This is used to specify the UUID that will be allocated to the Characteristic and can be either 16 or 128 bits. This variable is a handle that will have been pre-created using one of the functions BleHandleUuid16() or BleHandleUuid128() or 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. Note that if nCharProps specifies that the Characteristic is notifiable or indicatable and this value contains 0 then this function will abort with an appropriate resultcode.

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 CCCD is not to be created. Note that if nCharProps specifies that the Characteristic is broadcastable and this value contains 0 then this function will abort with an appropriate resultcode.

Interactive Command: NO

```
// Example :: BleCharNew()

#define BLE_ATTR_ACCESS_NONE                            0
#define BLE_ATTR_ACCESS_OPEN                            1
#define BLE_ATTR_ACCESS_ENC_NO_MITM                     2
#define BLE_ATTR_ACCESS_ENC_WITH_MITM                   3
#define BLE_ATTR_ACCESS_SIGNED_NO_MITM                  4
#define BLE_ATTR_ACCESS_SIGNED_WITH_MITM                5

#define BLE_CHAR_METADATA_ATTR_NOT_PRESENT              0

#define MAX_HTM_LEN                                     20

#define BLE_CHAR_PROPERTIES_BROADCAST                   0x01
#define BLE_CHAR_PROPERTIES_READ                        0x02
#define BLE_CHAR_PROPERTIES_WRITE_WO_RESPONSE           0x04
#define BLE_CHAR_PROPERTIES_WRITE                       0x08
#define BLE_CHAR_PROPERTIES_NOTIFY                      0x10
#define BLE_CHAR_PROPERTIES_INDICATE                    0x20
#define BLE_CHAR_PROPERTIES_AUTH_SIGNED_WR              0x40
```

```
#define BLE_CHAR_PROPERTIES_RELIABLE_WRITE              0x80

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 chProp   //properties for this Characteristic

//++++
// 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(BLE_ATTR_ACCESS_NONE,BLE_ATTR_ACCESS_NONE,MAX_HTM_LEN,0,rc)
//There is a CCCD in this characteristic
mdCccd=BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_ENC_NO_MITM,2,0,rc)
//There is no SCCD in this characteristic
mdSccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
//Create the Characteristic object
chProp = BLE_CHAR_PROPERTIES_INDICATE
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdVal,mdCccd,mdSccd)
```

BLECHARNEW is an extension function.

BleCharDescUserDesc

This function is used to add 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 this Descriptor as ".. that defines a UTF-8 string of variable size that is a textual description of the characteristic value". It further stipulates that this attribute is optinally writable and so a metadata argument exists to configure it to be so. The metadata is used to automatically update the "Writable Auxilliaries" properties flag for the Characteristic and that is the reason that flag bit is NOT specified for the nCharProps argument to the BleCharNew() function.

BLECHARDESCUSERDESC(userDesc$, mdUser)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

***userDesc$*** **byRef *userDesc$* AS STRING**
The user description string to initiliase the Descriptor with. If the length of the string exceeds the maximum length of an attribute then this function will abort with an error result code.

mdUser **byVal *dUser* AS INTEGER**
This is a mandatory metadata that is used to define the properties of the User Description 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 is set to 1 or greater, then the attribute will be marked as

writable and so the client will be able to provide a user description that will overwrite the one provided in this call.

Interactive Command: NO

```
// Example :: BleCharDescUserDesc()

// ~ ~ ~
// <see code leading up to this in example for BleCharNew()
// ~ ~ ~

//Create the Characteristic object
chProp = BLE_CHAR_PROPERTIES_INDICATE
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
AssertResCode(rc,9206)

if fIsWritable then
  //++++
  //Add the USER_DESCRIPTION Descriptor that will be writable
  //++++
  mdAttr=BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,20,1,rc)
else
  //++++
  //Add the USER_DESCRIPTION Descriptor that will be read only
  //++++
  mdAttr=BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_NONE,20,0,rc)
endif
attr$="user_desc"
rc = BleCharDescUserDesc(attr$,mdAttr)
```

BLECHARDESCUSERDESC is an extension function.

BleCharDescPrstnFrmt

This function is used to add an optional Presentation Format 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 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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nFormat **byVal *nFormat* AS INTEGER**
Valid range 0 to 255.
The format specifies how the data in the Value attribute is structured. The full list of valid values for this argument can be obtained on line at http://developer.bluetooth.org/gatt/Pages/FormatTypes.aspx and the enumeration is as described in section 3.3.3.5.2 of the 4.0 Bluetooth spec.
At the time of writing this user manual the enumeration list was 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 an enumeration to specify the units which are listed in the Assigned Numbers document published by the Bluetooth SIG and also availalable at http://developer.bluetooth.org/gatt/units/Pages/default.aspx

nNameSpace **byVal *nNameSpace* AS INTEGER**
Valid range 0 to 255
The value identifies the orgnaisation as defined in the Assigned Numbers document published by the Bluetooth SIG.

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()

// ~ ~ ~
// <see code leading up to this in example for BleCharNew()
// ~ ~ ~

//Create the Characteristic object
chProp = BLE_CHAR_PROPERTIES_INDICATE
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
AssertResCode(rc,9206)

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

// 16 bit signed integer = 0x05
// exponent = 2
// unit = 0x271A ( amount concentration (mole per cubic metre) )
// namespace = 0x01 == Bluetooth SIG
// description = 0x0000
rc = BleCharDescPrstnFrmt(0x0E,2,0x271A,0x01,0x0000)
```

BLECHARDESCPRSTNFRMT is an extension function.

BleCharDescAdd

This function is used to add any Characteristic Descriptor as long as it's 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().

Given that this function allows any existing or future defined Descriptor to be added which may or may not have write access and may or may not require security requirements, a metadata object has to be supplied to allow that to be configured for it.

BLECHARDESCADD (nUuid16, attr$, mdDesc)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nUuid16 **byVal *nUuid16* AS INTEGER**
This will be a value in the range 0x2905 to 0x2999, and the highest value is at the tiem of writing 0x2908 which is defined for the Report Reference Descriptor.

Please see http://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorsHomePage.aspx for the full list of Descriptors that are currently 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 a 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 is set to 1 or greater, then the attribute will be marked as writable and so the client will be able to modify the attribute value.

Interactive Command: NO

```
// Example :: BleCharDescAdd()

// ~ ~ ~
// <see code leading up to this in example for BleCharNew()
// ~ ~ ~

//Create the Characteristic object
chProp = BLE_CHAR_PROPERTIES_INDICATE
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
AssertResCode(rc,9206)

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

//++++
//Add the other Descriptor 0x29XX  -- first one
//++++
mdAttr=BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_NONE,MAXLEN,isVARLEN,rc)
attr$="some_value1"
rc = BleCharDescAdd(0x2905,attr$,mdAttr)

//++++
//Add the other Descriptor 0x29XX  -- second one
//++++
mdAttr=BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,MAXLEN,isVARLEN,rc)
attr$="some_value2"
rc = BleCharDescAdd(0x2906,attr$,mdAttr)

//++++
//Add the other Descriptor 0x29XX  -- last one
//++++
mdAttr=BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_NONE,MAXLEN,isVARLEN,rc)
attr$="some_value3"
rc = BleCharDescAdd(0x2907,attr$,mdAttr)
```

BLECHARDESCADD is an extension function.

BleCharCommit

This function is used to commit 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 then 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 resultcode 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 Decleration and the other for the Value. However there is no need for the smartBASIC 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 it's content – which never changes once created.

BLECHARCOMMIT (hService,attr$,charHandle)

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.
Arguments:	
hService	**byVal hService AS INTEGER** This is the handle of the service that this Characteristic shall belong to which in turn will have been 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 and will have the special value 0 if the function fails with a non-zero resultcode. This handle is then subsequently used as an argument in subsequent function calls to perform read/write actions so it is essential to have around in a global smartBASIC variable. When a significant event occurs which is as a result of action by a remote client, then an event message will be sent to the application which can be serviced using a handler. That message will contain a handle field which corresponds to this composite characteristic handle and the 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()

#define BLE_SERVICE_SECONDARY                          0
#define BLE_SERVICE_PRIMARY                            1

dim hHtsSvc    //composite handle for hts primary service

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

// ~ ~ ~
// <see code leading up to this in example for BleCharNew()
// ~ ~ ~

dim hHtsMeas    //composite handle for htsMeas characteristic
dim hHtsLoc     //composite handle for htsLoc characteristic

//Create the Characteristic object
chProp = BLE_CHAR_PROPERTIES_INDICATE
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
AssertResCode(rc,9206)

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

//++++
//Commit the characteristic with some initial data
//++++
attr$="hello\00worl\64"
rc = BleCharCommit(hHtsSvc,attr$,hHtsMeas)
//the characteristic will now be visible in the GATT table
//and is refrenced by 'hHtsMeas'for subsequent calls
```

BLECHARCOMMIT is an extension function.

BleCharValueRead

This function is used to read 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 will be presented asynchronously to the smartBASIC 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$)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

charHandle **byVal *charHandle* AS INTEGER**
This is the handle to the characteristic whose value needs to 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 1 :: BleCharValueRead()

//------------------------------------------------------------------------------
//Create a Battery PRIMARY service attribure which has a uuid of 0x180F
//------------------------------------------------------------------------------
dim hBatSvc    //composite handle for batteru primary service
rc = BleSvcCommit(BLE_SERVICE_SECONDARY,BleHandleUuid16(0x180F),hBatSvc)
// Add one or more characteristics
dim hBatLvl
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
atr$="\50"  //80% battery level
rc = BleCharCommit(hBatSvc,attr$,hBatLvl)

// ~ ~ ~ ~ ~ ~ ~ ~
// some time latter, the following code fragment to change the battery level
// to 50% which is 0x32 in hex.
// ~ ~ ~ ~ ~ ~ ~ ~

rc = BleCharValueRead(hBatLvl,attr$)
if rc != 0 then
  print "\nFailed to read value from characteristic"
endif
```

```
// Example 2 :: BleCharValueRead()  -- event driven usage

//------------------------------------------------------------------------------
//Create an Immediate Alert PRIMARY service attribure which has a uuid of 0x1802
//------------------------------------------------------------------------------
dim hIasSvc    //composite handle for batteru primary service
rc = BleSvcCommit(BLE_SERVICE_SECONDARY,BleHandleUuid16(0x1802),hIasSvc)
// Add one or more characteristics
dim hAlert
rc = BleCharNew(chProp,BleHandleUuid16(0x2A06),mdAttr,mdCccd,mdSccd)
atr$="\00"  //no initial alert
rc = BleCharCommit(hIasSvc,attr$,hAlert)

//handler to service characteristic value writes from gatt clients
function HandlerCharVal(BYVAL hChar AS INTEGER) as integer
  dim lvl,len
  if hChar == hAlert then
  rc = BleCharValueRead(hAlert,attr$)
    if rc != 0 then
      print "\nFailed to read value from characteristic"
    else
      len = BleDecodeU8(attr$,lvl,0)
      print "\n --- Immediate Alert Service Alert ";lvl
    endif
  else
    DbgMsgVal(" +++ HVC : ",hChar)
  endif
endfunc 1

//install a handler for writes to characteristic values
OnEvent  EVCHARVAL  call HandlerCharVal

//wait for events and messages
waitevent
```

BLECHARVALUEREAD is an extension function.

BleCharValueWrite

This function is used to write new data into the VALUE attribute of a Characteristic which is in turn indentified by a composite handle that was returned by the function BleCharCommit().

BLECHARVALUEWRITE (charHandle,attr$)

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.
Arguments:	
charHandle	**byVal *charHandle* AS INTEGER** This is the handle to the characteristic whose value needs to be updated which was returned when BleCharCommit() was called.
attr$	**byRef attr$ AS STRING** This string variable contains the new value to be written to the characteristic.

Interactive Command: NO

```
// Example :: BleCharValueWrite()

//------------------------------------------------------------------------------
//Create a Battery PRIMARY service attribure which has a uuid of 0x180F
//------------------------------------------------------------------------------
dim hBatSvc    //composite handle for batteru primary service
rc = BleSvcCommit(BLE_SERVICE_SECONDARY,BleHandleUuid16(0x180F),hBatSvc)
// Add one or more characteristics
dim hBatLvl
rc = BleCharNew(chProp,BleHandleUuid16(0x2A1C),mdAttr,mdCccd,mdSccd)
atr$="\50"  //80% battery level
```

```
rc = BleCharCommit(hBatSvc,attr$,hBatLvl)

// ~ ~ ~ ~ ~ ~ ~ ~
// some time latter, the following code fragment to change the battery level
// to 50% which is 0x32 in hex.
// ~ ~ ~ ~ ~ ~ ~ ~

atr$="\32"  //50% battery level
rc = BleCharValueWrite(hBatLvl,attr$)
if rc != 0 then
  print "\nFailed to write new value"
endif
```

BLECHARVALUEWRITE is an extension function.

BleCharValueNotify

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 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 a acknowledgement from the client.

BLECHARVALUENOTIFY (charHandle,attr$)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

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

***attr$*** **byRef attr$ AS STRING**
This string variable contains the new value to be written to the characteristic and then to be sent as a notification to the client. If there is no connection then this function will fail with an appropriate resultcode.

Interactive Command: NO

```
// Example :: BleCharValueNotify()

//------------------------------------------------------------------------------
//Create some PRIMARY service attribure which has a uuid of 0x18FF
//------------------------------------------------------------------------------
dim hSvc    //composite handle for batteru primary service
rc = BleSvcCommit(BLE_SERVICE_PRIMARY,BleHandleUuid16(0x18FF),hSvc)
// Add one or more characteristics
dim hMyChar
rc = BleCharNew(chProp,BleHandleUuid16(0x2AFF),mdAttr,mdCccd,mdSccd)
atr$="\00"  //no initial alert
```

```
rc = BleCharCommit(hSvc,attr$,hMyChar)

//handler to service writes to CCCD which tells us we can indicate or not
function HandlerCharCccd(BYVAL hChar AS INTEGER, BYVAL nVal AS INTEGER) as integer
  if hChar == hMyChar then
    //The following if statment to convert to 1 is only so that we can submit
    //this app to the regression test
    if nVal & 0x01 then
      print "\nNotifications have been enabled by client"
      attr$="hello"
   rc = BleCharValueNotify(hMyChar,attr$)
      if rc != 0 then
        print "\nFailed to notify new value"
      endif
    else
      print "\nNotifications have been disabled by client"
    endif
  else
    print "\nThis is for some other characteristic"
  endif
endfunc 1

//install a handler for writes to characteristic values
OnEvent  EVCHARCCCD          call HandlerCharCccd

//wait for events and messages
waitevent
```

BLECHARVALUENOTIFY is an extension function.

BleCharValueIndicate

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 indicationto the GATT client. The characteristic is identified by a composite handle that was returned by the function BleCharCommit().

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

BLECHARVALUEINDICATE (charHandle,attr$)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

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

***attr$*** **byRef attr$ AS STRING**
This string variable contains the new value to be written to the characteristic and then to be sent as a notification to the client. If there is no connection then this function will fail with an appropriate resultcode.

Interactive Command: NO

```
// Example :: BleCharValueIndicate()

//----------------------------------------------------------------------------
//Create some PRIMARY service attribure which has a uuid of 0x18FF
//----------------------------------------------------------------------------
dim hSvc    //composite handle for batteru primary service
rc = BleSvcCommit(BLE_SERVICE_PRIMARY,BleHandleUuid16(0x18FF),hSvc)
// Add one or more characteristics
dim hMyChar
rc = BleCharNew(chProp,BleHandleUuid16(0x2AFF),mdAttr,mdCccd,mdSccd)
atr$="\00"  //no initial alert
rc = BleCharCommit(hSvc,attr$,hMyChar)

//handler to service characteristic value confirmation from gatt client
function HandlerCharHvc(BYVAL hChar AS INTEGER) as integer
  if hChar == hMyChar then
    print "\nGot confirmation to recent indication"
  else
    print "\nGot confirmation to some other indication"; hChar
  endif
endfunc 1

//handler to service writes to CCCD which tells us we can indicate or not
function HandlerCharCccd(BYVAL hChar AS INTEGER, BYVAL nVal AS INTEGER) as integer
  if hChar == hMyChar then
    //The following if statment to convert to 1 is only so that we can submit
    //this app to the regression test
    if nVal & 0x02 then
      print "\nIndications have been enabled by client"
      attr$="hello"
   rc = BleCharValueIndicate(hMyChar,attr$)
      if rc != 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

//install a handler for writes to characteristic values
OnEvent  EVCHARCCCD         call HandlerCharCccd
OnEvent  EVCHARHVC          call HandlerCharHvc

//wait for events and messages
waitevent
```

BLECHARVALUEINDICATE is an extension function.

BleCharDescRead

This function is used to read the current content of a writable Characteristic Descriptor identified by a composite handle that was previously returned by the function BleCharCommit()and a 16 bit UUID which identifies the type of Descriptor.

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

BLECHARDESCREAD (charHandle,nUuid,attr$))

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

charHandle **byVal *charHandle* AS INTEGER**
This is the handle to the characteristic whose descriptor needs to be read which was returned when BleCharCommit() was called.

nUuid **byVal nUuid AS INTEGER**
This is a 16 bit UUID that is a published number by the Bluetooth SIG. It will be a number starting with hex 0x29 usually. For it to start with any other value implies that over 100 types of descriptors have been defined by the Bluetooth SIG and so far in 2 years since the specification was published less than 10 have been defined.

***attr$*** **byRef attr$ AS STRING**
This string variable contains the new value from the characteristic descriptor.

Interactive Command: NO

```
// Example :: BleCharDescRead()

//------------------------------------------------------------------------------
//Create some PRIMARY service attribure which has a uuid of 0x18FF
//------------------------------------------------------------------------------
dim hSvc    //composite handle for batteru primary service
rc = BleSvcCommit(BLE_SERVICE_PRIMARY,BleHandleUuid16(0x18FF),hSvc)
// Add one or more characteristics
dim hMyChar
rc = BleCharNew(chProp,BleHandleUuid16(0x2AFF),mdAttr,mdCccd,mdSccd)
atr$="\00"  //no initial alert
rc = BleCharCommit(hSvc,attr$,hMyChar)

//handler to service writes to descriptors by a gatt client
function HandlerCharDesc(BYVAL hChar AS INTEGER, BYVAL nComp AS INTEGER) as integer
  dim instnc
  dim nUuid
  instnc = nComp >> 16
  nUuid  = nComp & 0xFFFF
  if hChar == hMyChar then
  rc = BleCharDescRead(hChar,nUuid,instnc,attr$)
    if rc ==0 then
      print "\nNew value for desriptor "; nUuid; " is ";StrHexize(attr$)
    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

//wait for events and messages
waitevent
```

BLECHARDESCREAD is an extension function.

Encoding Functions

Data for Characteristics are stored in Value attributes which is just an array of bytes. Similarly multibyte Characteristic Descriptors content are also stored in byte arrays. Those bytes of arrays are manipulated in smartBASIC applications using STRING variables.

The Bluetooth specification stipulates that mutlibyte 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 smartBASIC.

Please note that CCCD and SCCD Descritors are special cases as they are defined as having just 2 bytes which are treated as 16 bit integers. And this special treatment is reflected in smartBASIC applications so that INTEGER variables are used to manipulate those values instead of STRINGS.

BleEncode8

This function is used to overwrite 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, then this function will fail. 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.

BLEENCODE8 (attr$,nData, nIndex)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

attr$	**byRef *attr$* AS STRING** This argument is the string that will be written to an attribute
nData	**byVal nData AS INTEGER** The least signifant byte of this integer is saved and the rest is ignored.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

Interactive Command: NO

```
// Example :: BleEncode8()

dim rc
dim attr$
attr$=""

//write 'C' to index 2 -- will get extended
rc=BleEncode8(attr$,0x43,2)
//write 'A' to index 0 -- no need for extension
rc=BleEncode8(attr$,0x41,0)
//write 'B' to index 1 -- no need for extension
rc=BleEncode8(attr$,0x42,1)
//write 'D' to index 3 -- will get extended
rc=BleEncode8(attr$,0x44,3)

print "attr$=";attr$   //output will be 'attr$=ABCD'
```

BLEENCODE8 is an extension function.

BleEncode16

This function is used to overwrite two 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, then this function will fail. 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.

BLEENCODE16 (attr$,nData, nIndex)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

attr$	**byRef *attr$* AS STRING** This argument is the string that will be written to an attribute
nData	**byVal nData AS INTEGER** The two least signifant bytes of this integer is saved and the rest is ignored.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

Interactive Command: NO

```
// Example :: BleEncode16()

dim rc
dim attr$
attr$=""

//write 0x4443 to index 2 -- will get extended
rc=BleEncode16(attr$,0x4443,2) //attr$ will contain ??CD where ?? is anything
//write 0x12344241 to index 0 -- no need for extension
rc=BleEncode16(attr$,0x4241,0) //attr$ will contain ABCD (0x1234 got ignored)
//write 0x4645 to index 3 -- will get extended
rc=BleEncode16(attr$,0x4645,4)

print "attr$=";attr$   //output will be 'attr$=ABCDEF'
```

BLEENCODE16 is an extension function.

BleEncode24

This function is used to overwrite 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, then this function will fail. 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.

BLEENCODE24 (attr$,nData, nIndex)

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that will be written to an attribute
nData	**byVal nData AS INTEGER** The three least signifant bytes of this integer is saved and the rest is ignored.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

Interactive Command: NO

```
// Example :: BleEncode24()

dim rc
dim attr$
attr$=""

//write 0x444342 to index 1 -- will get extended
rc=BleEncode24(attr$,0x444342,1)
//write 0x41 to index 0 -- no need for extension
rc=BleEncode8(attr$,0x41,0)
//write 0x4645 to index 4 -- will get extended
rc=BleEncode16(attr$,0x4645,4)

print "attr$=";attr$   //output will be 'attr$=ABCDEF'
```

BLEENCODE24 is an extension function.

BleEncode32

This function is used to overwrite four 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, then this function will fail. 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.

BLEENCODE32(attr$,nData, nIndex)

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that will be written to an attribute
nData	**byVal nData AS INTEGER** The four bytes of this integer is saved and the rest is ignored.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

Interactive Command: NO

```
// Example :: BleEncode32()

dim rc
dim attr$
attr$=""

//write 0x45444342 to index 1 -- will get extended
rc=BleEncode32(attr$,0x54444342,1)
//write 0x41 to index 0 -- no need for extension
rc=BleEncode8(attr$,0x41,0)

print "attr$=";attr$   //output will be 'attr$=ABCDEF'
```

BLEENCODE32 is an extension function.

BleEncodeFLOAT

This function is used to overwrite four 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 byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, then this function will fail. 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.

BLEENCODEFLOAT (attr$, nMatissa, nExponent, nIndex)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

***attr$*** **byRef *attr$* AS STRING**
This argument is the string that will be written to an attribute

nMatissa **byVal nMantissa AS INTEGER**
This value must be in the range -8388600 to +8388600 otherwise the function will fail. The data is written in little endian so that the least significant byte is at the lower memory address.

Please 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, otherwise the function will fail.

nIndex **byVal *nIndex* AS INTEGER**
This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

Interactive Command: NO

```
// Example :: BleEncodeFLOAT()

dim rc
dim attr$
attr$=""

//write 0x1234 as 16 bite integer to index 0
rc=BleEncode16(attr$,0x1234,0)
```

```
//write 1234567 x 10^-54 as FLOAT to index 2
rc=BleEncodeFLOAT(attr$,1234567,-54,2)

//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
rc=BleEncodeFLOAT(attr$,1234567,1000,2)
if rc!=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
rc=BleEncodeFLOAT(attr$,10000000,0,2)
if rc!=0 then
  print "\nfailed to encode to FLOAT"
endif
```

BLEENCODEFLOAT is an extension function.

BleEncodeSFLOATEX

This function is used to overwrite two bytes in a string at a specified offset as short 16 bit float value. 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, then this function will fail. 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.

BLEENCODESFLOATEX(attr$,nData, nIndex)

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that will be written to an attribute
nData	**byVal nData AS INTEGER** The 32 bit value is converted into a 2 byte IEEE-11073 16 bit SFLOAT which consists of a 12 bit signed mantissa and a 4 bit signed exponent. This means a signed 32 bit value will always fit in such a FLOAT enitity, but there will be a loss in significance to 12 from 32.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be

extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

Interactive Command: NO

```
// Example :: BleEncodeSFLOAT()

dim rc
dim attr$
attr$=""

//write 0x1234 as 16 bite integer to index 0
rc=BleEncode16(attr$,0x1234,0)

//write 10,000,000 as SFLOAT to index 2
rc=BleEncodeSFLOAT(attr$,10000000,2)
```

BLEENCODESFLOAT is an extension function.

BleEncodeSFLOAT

This function is used to overwrite two bytes in a string at a specified offset as short 16 bit float value. 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, then this function will fail. 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.

BLEENCODESFLOAT(attr$, nMatissa, nExponent, nIndex)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

***attr$*** **byRef *attr$* AS STRING**
This argument is the string that will be written to an attribute

nMatissa **byVal n AS INTEGER**
This value must be in the range -2046 to +2046 otherwise the function will fail. The data is written in little endian so that the least significant byte is at the lower memory address.

Please 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, otherwise the function will fail.

nIndex **byVal *nIndex* AS INTEGER**
This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

Interactive Command: NO

```
// Example :: BleEncodeSFLOATEX()

dim rc
dim attr$
attr$=""

//write 0x1234 as 16 bite integer to index 0
rc=BleEncode16(attr$,0x1234,0)

//write 1234 x 10^-4 as FLOAT to index 2
rc=BleEncodeSFLOATEX(attr$,1234,-4,2)

//write 1234 x 10^10 as FLOAT to index 2 and it will fail
//because the exponent is too large, it has to be < 7
rc=BleEncodeSFLOATEX(attr$,1234,10,2)
if rc!=0 then
  print "\nfailed to encode to SFLOAT"
endif

//write 10000 x 10^0 as FLOAT to index 2 and it will fail
//because the mantissa is too large, it has to be < 2046
rc=BleEncodeSFLOATEX(attr$,10000,0,2)
if rc!=0 then
  print "\nfailed to encode to SFLOAT"
endif
```

BLEENCODESFLOATEX is an extension function.

BleEncodeTIMESTAMP

This function is used to overwrite a 7 byte string into the 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.

The 7 byte string consists of 1 byte century, 1 byte year,1 byte month, 1 byte day, 1 byte hour, 1 byte minute and 1 byte second. If (year * month) is zero then it will be taken as a don't care year and likewise all the other fields get be set to 0 for don't care.

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, then this function will fail. 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.

Please note that 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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

attr$	**byRef *attr$* AS STRING** This argument is the string that will be written to an attribute
timestamp$	**byRef *timestamp$* AS STRING** This is an exactly 7 byte string as described above. For example 5 May 2013 10:31:24 will be represented as "\14\0D\05\05\0A\1F\18"
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

Interactive Command: NO

```
// Example :: BleEncodeTIMESTAMP()

dim rc
dim attr$
dim ts$
attr$=""

//write 0x1234 as 16 bite integer to index 0
rc=BleEncode16(attr$,0x1234,0)

//write the timetampe 5 May 2013 10:31:24>
ts$="\14\0D\05\05\0A\1F\18"
rc=BleEncodeTIMESTAMP(attr$,ts$,2)
```

BLEENCODETIMESTAMP is an extension function.

BleEncodeSTRING

This function is used to overwrite a substring at a specified offset with data from another substring of a string. If the destination 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, then this function will fail. 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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

***attr$*** — **byRef *attr$* AS STRING**
This argument is the string that will be written to an attribute

nIndex1 — **byVal *nIndex1* AS INTEGER**
This is the zero based index into the string attr$ where the new fragment of data will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

***str$*** — **byRef *str$* AS STRING**
This contains the source data which is qualified by the nIndex1 and nLen arguments that follow.

nIndex2 — **byVal *nIndex2* AS INTEGER**
This is the zero based index into the string str$ from where the data will be copied. No data will be copied if this index is negative or greater than the string

nLen — **byVal *nLen* AS INTEGER**
This species the number of bytes from offset nIndex2 that need to be copied into the destination string. It will be clipped to the number of bytes left to copy after the index.

Interactive Command: NO

```
// Example :: BleEncodeSTRING()
```

```
dim rc
dim attr$
dim ts$
attr$=""

//write 0x1234 as 16 bite integer to index 0
rc=BleEncode16(attr$,0x1234,0)

//write "Wor" from "Hello World" to the attbute at index 2
ts$="Hello World"
rc=BleEncodeSTRING(attr$,2,ts$,6,3)
```

BLEENCODESTRING is an extension function.

BleEncodeBITS

This function is used to overwrite 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, then it will be extended with the new extended block uninitialized and then the bits specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, then this function will fail. 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)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

***attr$*** — **byRef *attr$* AS STRING**
This argument is the string that will be written to an attribute and is treated as a bit array.

nDstIdx — **byVal *nDstIdx* AS INTEGER**
This is the zero based bit index into the string attr$, which is treated as a bit array, where the new fragment of data bits will be written to. If the string attr$ is currently not long enough to accommodate the index plus the length of the fragment then it will be extended. If the new extended length will exceed the maximum allowable length of an attribute (see SYSINFO(2013)) then this function will fail.

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 will be copied if this index is negative or greater than 32

nBitLen	**byVal *nBitLen* AS INTEGER** This species the number of bits from offset nSrcIdx that need 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()

dim n,ba
dim attr$
dim ts$
attr$=""

//copy 5 bits from index 7 to string attr$
ba=b'1110100001111
rc=BleEncodeBITS(attr$,20,ba,7,5)
```

BLEENCODEBITS is an extension function.

Decoding Functions

Data in a Characteristic is stored in a Value attribute which is just an array of bytes. Similarly multibyte Characteristic Descriptors content are also stored in byte arrays. Those bytes of arrays are manipulated in smartBASIC applications using STRING variables.

Attibute data is stored in little endian format.

This section describes all the decoding functions which allow attribute strings to be read from smaller bytewise subfields in a more efficient manner compared to the generic STRXXXX functions that are made available in smartBASIC.

Please note that CCCD and SCCD Descritors 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

This function is used to read 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 will fail and returns 0.

BLEDECODES8 (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$* AS STRING** This argument is a reference to the attribute string that will be read from
nData	**byRef nData AS INTEGER** This argument is a reference to an integer that will be updated with the 8 bit data from attr$, after sign extension.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeS8()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim v1

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read signed byte from index 2
rc=BleDecodeS8(attr$,v1,2)
print "\ndata=";integer.h' v1
//output will be 00000002

//read signed byte from index 6
rc=BleDecodeS8(attr$,v1,6)
print "\ndata=";integer.h' v1
//output will be FFFFFF86
```

BLEDECODES8 is an extension function.

BleDecodeU8

This function is used to read 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 then this function will fail.

BLEDECODEU8 (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$* AS STRING** This argument is a reference to the attribute string that will be read from
nData	**byRef nData AS INTEGER** This argument is a reference to an integer that will be updated with the 8 bit data from attr$, without sign extension.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeU8()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim v1

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read unsigned byte from index 2
rc=BleDecodeU8(attr$,v1,2)
print "\ndata=";integer.h' v1
//output will be 00000002

//read unsigned byte from index 6
rc=BleDecodeU8(attr$,v1,6)
print "\ndata=";integer.h' v1
//output will be 00000086
```

BLEDECODEU8 is an extension function.

BleDecodeS16

This function is used to read 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 will fail.

BLEDECODES16 (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$* AS STRING** This argument is a reference to the attribute string that will be read from
nData	**byRef nData AS INTEGER** This argument is a reference to an integer that will be updated with the 2 byte data from attr$, after sign extension.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeS16()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim v1

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read signed byte from index 2
rc=BleDecodeS16(attr$,v1,2)
print "\ndata=";integer.h' v1
//output will be 00000302

//read signed byte from index 6
rc=BleDecodeS16(attr$,v1,6)
print "\ndata=";integer.h' v1
//output will be FFFF8786
```

BLEDECODES16 is an extension function.

BleDecodeU16

This function is used to read 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 will fail.

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$* AS STRING** This argument is a reference to the attribute string that will be read from
nData	**byRef nData AS INTEGER** This argument is a reference to an integer that will be updated with the 2 byte data from attr$, without sign extension.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeU16()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim v1

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read unsigned byte from index 2
rc=BleDecodeU16(attr$,v1,2)
print "\ndata=";integer.h' v1
//output will be 00000302

//read unsigned byte from index 6
rc=BleDecodeU16(attr$,v1,6)
print "\ndata=";integer.h' v1
//output will be 00008786
```

BLEDECODEU16 is an extension function.

BleDecodeS24

This function is used to read 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 then this function will fail.

BLEDECODES24 (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 argument is a reference to the attribute string that will be read from

nData **byRef nData AS INTEGER**
This argument is a reference to an integer that will 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$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeS24()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim v1

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read signed byte from index 2
rc=BleDecodeS24(attr$,v1,2)
print "\ndata=";integer.h' v1
//output will be 00040302

//read signed byte from index 6
rc=BleDecodeS24(attr$,v1,6)
print "\ndata=";integer.h' v1
//output will be FF888786
```

BLEDECODES24 is an extension function.

BleDecodeU24

This function is used to read 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 will fail.

BLEDECODEU24 (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 argument is a reference to the attribute string that will be read from

nData **byRef nData AS INTEGER**
This argument is a reference to an integer that will 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$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeU24()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim v1

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read unsigned byte from index 2
rc=BleDecodeU24(attr$,v1,2)
print "\ndata=";integer.h' v1
//output will be 00040302

//read unsigned byte from index 6
rc=BleDecodeU24(attr$,v1,6)
print "\ndata=";integer.h' v1
//output will be 00888786
```

BLEDECODEU24 is an extension function.

BleDecode32

This function is used to read four bytes in a string at a specified offset into a 32bit integer variable.

If the offset points beyond the end of the string then this function will fail.

BLEDECODE32 (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$* AS STRING** This argument is a reference to the attribute string that will be read from
nData	**byRef nData AS INTEGER** This argument is a reference to an integer that will be updated with the 4 byte data from attr$, after sign extension.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecode32()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim v1

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read signed byte from index 2
rc=BleDecode32(attr$,v1,2)
print "\ndata=";integer.h' v1
//output will be 05040302

//read signed byte from index 6
rc=BleDecode32(attr$,v1,6)
print "\ndata=";integer.h' v1
//output will be 89888786
```

BLEDECODE32 is an extension function.

BleDecodeFLOAT

This function is used to read four bytes in a string at a specified offset into a couple of 32bit integer variables. The decoding will result in one variable that will be the 24 bit signed maintissa and the other will be the 8 bit signed exponent

If the offset points beyond the end of the string then this function will fail.

BLEDECODEFLOAT (attr$, nMatissa, nExponent, 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 argument is a reference to the attribute string that will be read from

nMantissa **byRef nMantissa AS INTEGER**
This variable will be updated with the 24 bit mantissa from the 4 byte object.

Note that if the nExponent is 0, then 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 variable will be updated with the 8 bit mantissa. If it is 0, then check the nMantissa for special cases as stated above.

nIndex **byVal *nIndex* AS INTEGER**
This is the zero based index into the string attr$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeFLOAT()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim mantissa
dim exponent

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read 4 bytes FLOAT from index 2 in the string
rc=BleDecodeFLOAT(attr$,mantissa,exponent,2)
print "\nmantissa=";integer.h' mantissa
//mantissa will be 00030201
print "\nexponent=";integer.h' exponent
//mantissa will be 00000004

//read 4 bytes FLOAT from index 6 in the string
rc=BleDecodeFLOAT(attr$,mantissa,exponent,6)
print "\nmantissa=";integer.h' mantissa
```

```
//mantissa will be FF888786
print "\nexponent=";integer.h' exponent
//mantissa will be FFFFFF89
```

BLEDECODEFLOAT is an extension function.

BleDecodeSFLOAT

This function is used to read two bytes in a string at a specified offset into a couple of 32bit integer variables. The decoding will result in one variable that will be the 12 bit signed maintissa and the other will be the 4 bit signed exponent

If the offset points beyond the end of the string then this function will fail.

BLEDECODESFLOAT (attr$, nMatissa, nExponent, 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 argument is a reference to the attribute string that will be read from

nMantissa **byRef nMantissa AS INTEGER**
This variable will be updated with the 12 bit mantissa from the 2 byte object.

Note that if the nExponent is 0, then 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 variable will be updated with the 4 bit mantissa. If it is 0, then check the nMantissa for special cases as stated above.

nIndex **byVal *nIndex* AS INTEGER**
This is the zero based index into the string attr$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeSFLOAT()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim mantissa
dim exponent

//assume a gatt attibure contains the hex data 00010203048586878889
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\03\04\85\86\87\88\89"

//read 2 bytes SFLOAT from index 2 in the string
rc=BleDecodeSFLOAT(attr$,mantissa,exponent,2)
print "\nmantissa=";integer.h' mantissa
//mantissa will be 00000201
print "\nexponent=";integer.h' exponent
//mantissa will be 00000000

//read 2 bytes FLOAT from index 6 in the string
rc=BleDecodeSFLOAT(attr$,mantissa,exponent,6)
print "\nmantissa=";integer.h' mantissa
//mantissa will be 00000786
print "\nexponent=";integer.h' exponent
//mantissa will be FFFFFFF8
```

BLEDECODESFLOAT is an extension function.

BleDecodeTIMESTAMP

This function is used to read 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 will fail.

The 7 byte source string consists of a 2 byte year,1 byte month, 1 byte day, 1 byte hour, 1 byte minute and 1 byte second. If year is zero then it will be taken as a don't care year and likewise all the other fields get be set to 0 for don't care.

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)

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 argument is a reference to the attribute string that will be read from

timestamp$	**byRef *timestamp$* AS STRING** On exit this will be an exactly 7 byte string as described above. For example 5 May 2013 10:31:24 will be represented as "\14\0D\05\05\0A\1F\18"
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the data is read from. If the string attr$ is currently not long enough to accommodate the index plus the number of bytes to read then this function will fail

Interactive Command: NO

```
// Example :: BleDecodeTIMESTAMP()

dim rc
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim ts$

//assume a gatt attibure contains the hex data 000102DD0705050A1F18
//which corresponds to the date/time 5 May 2103 @ 10:31:24
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="\00\01\02\DD\07\05\05\0A\1F\18"

//read 7 byte timestamp from index 2 in the string
rc=BleDecodeTIMESTAMP(attr$,ts$,2)
print "\nTimestamp=";StrHexize(ts$)
//nTimestamp will be 140D05050A1F18 where 140D == 2013 == 0x07DD
```

BLEENCODETIMESTAMP is an extension function.

BleDecodeSTRING

This function is used to read 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)

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 argument is a reference to the attribute string that will be read from

nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ where the data is read from.
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 argument is specifies the maximum number of bytes to read from attr$

Interactive Command: NO

```
// Example :: BleDecodeSTRING()

dim rc
dim charHandle       //handle to characteristic
dim attr$            //characteristic value will be read into thid variable
dim d$

//assume a gatt attibure contains the hex data 4142434445464748494A which
//is the string ABCDEFGHIJ
rc=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="ABCDEFGHIJ"

//read max 4 bytes from index 2 in the string
rc=BleDecodeSTRING(attr$,2,d$,4)
print "\nd$=";d$
//output will be CDEF

//read max 20 bytes from index 2 in the string
rc=BleDecodeSTRING(attr$,2,d$,20)
print "\nd$=";d$
//output will be CDEFGHIJ as it got truncated

//read max 4 bytes from index 14 in the string
rc=BleDecodeSTRING(attr$,14,d$,4)
print "\nd$=";d$
//output will be an empty string
```

BLEDECODESTRING is an extension function.

BleDecodeBITS

This function is used to read bits from an attribute string at a specified offset (which is treated as a bit array) into a destination integer object (which is 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)

FUNCTION

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$* AS STRING** This argument is a reference to the attribute string that will be read from which is treated as a bit array. Hence a string of 10 bytes will be an array of 80 bits.
nSrcIdx	**byVal *nSrcIdx* AS INTEGER** This is the zero based bit index into the string attr$ where the data is read from. For example, the third bit in the second byte will be index number 10.
dstBitArr	**byRef *dstBitArr* AS INTEGER** This argument is a reference to an integer which is treated as an array of 32 bits which is copied into. Only the bits that are written to get 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 is specifies the maximum number of bits to read from attr$ and due to the destination being an integer variable it cannot be greater than 32. Negative values are treated as 0.

Interactive Command: NO

```
// Example :: BleDecodeBITS()

dim n
dim charHandle      //handle to characteristic
dim attr$           //characteristic value will be read into thid variable
dim ba

//assume a gatt attibure contains the hex data 4142434445464748494A which
//is the string ABCDEFGHIJ
n=BleCharValueRead(charHandle,attr$)
//assuming successfully read so attr$="ABCDEFGHIJ"

//read max 14 bits from index 20 in the string to index 10
n=BleDecodeBITS(attr$,20,ba,10,14)
print "\nbit array=";integer'b ba

//read max 14 bits from index 20 in the string to index 10
ba=0x12345678
n=BleDecodeBITS(attr$,14000,ba,0,14)
print "\nbit array=";integer'b ba
//ba will not have been modified
```

BLEDECODEBITS is an extension function.

Predefined GATT Server Functions

This section describes all functions related to managing services and profiles from a GATT server perspective.

BleGapSvcInit

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 then default values will be exposed. Given this is a mandatory service, unlike other services which need to be registered, this one just needs to be initialised as the underlying BLE stack unconditionally registers it when starting up.

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

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

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

deviceName **byRef *deviceName* AS STRING**
The name of the device (e.g. Laird_Thermometer) that will be stored in the 'Device Name' characteristic of the GAP service.

Note: When an advert report is created using BLEADVRPTINIT() this field will be read from the service and an attempt will be made to append it in the Device Name AD. If this name is too long then that function to initialise the advert report will fail and so a default name will be transmitted. It is recommended that the device name submitted in this call be as short as possible.

nameWritable **byVal *nameWritable* AS INTEGER**
If this is non-zero, then the peer device is allowed to write the name of the device. Some profiles allow this to be optionally doable.

nAppearance **byVal *nAppearance* AS INTEGER**
The external appearance of the device and updates the Appearance characteristic of the GAP service. The full list of possible device appearances can be found at org.bluetooth.characteristic.gap.appearance.

nMinConnInterval	**byVal *nMinConnInterval* AS INTEGER** The minimum connection interval and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. The range of this value is between 7500 and 4000000 microseconds (rounded to the nearest multiple of 1250 microseconds). This value must be smaller than nMaxConnInterval.
nMaxConnInterval	**byVal *nMaxConnInterval* AS INTEGER** The maximum connection interval and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. The range of this value is between 7500 and 4000000 microseconds (rounded to the nearest multiple of 1250 microseconds). This value must be larger than nMinConnInterval.
nSupervisionTimeout	**byVal *nSupervisionTimeout* AS INTEGER** The link supervision timeout and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. The range of this value is between 100000 to 32000000 microseconds (rounded to the nearest multiple of 10000 microseconds).
nSlaveLatency	**byVal *nSlaveLatency* AS INTEGER** The 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

```
DIM rc,deviceName$, appearance, MinConnInt, MaxConnInt, ConnSupTimeout, SL

deviceName$ = "Laird_TS"
appearance = 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
ConnSupTimeout = 4000000      'Connection supervisory timeout is 4 seconds
SL = 0                        'Slave latency--number of conn events that can be missed

rc = blegapsvcinit(deviceName$,appearance,MinConnInt,MaxConnInt,ConnSupTimeout,SL)
```

BLEGAPSVCINIT is an extension function.

BleSvcRegDevInfo

This function is used to register the Device Information service with the GATT server.

The 'Device Information' service contains nine characteristics as listed at 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 An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

manfName$	**byVal *manfName$* AS STRING** The device's manufacturer name. It can be set as an empty string to omit submission.
modelNum$	**byVal *modelNum$* AS STRING** The device's model number. It can be set as an empty string to omit submission.
serialNum$	**byVal *serialNum$* AS STRING** The device's serial number. It can be set as an empty string to omit submission.
hwRev$	**byVal *hwRev$* AS STRING** The device's hardware revision string. It can be set as an empty string to omit submission.
swRev$	**byVal *swRev$* AS STRING** The device's software revision string. It can be set as an empty string to omit submission.
sysId$	**byVal *sysId$* AS STRING** The device's system ID as defined in the specifications. It can be set as an empty string to omit submission otherwise it shall be a string exactly 8 octets long, where:- Byte 0..4 := Manufacturer Identifier Byte 5..7 := Organisationally Unique Identifier Note: For the special case of the string being exactly 1 character long and containing "@" then the system ID will be created from the mac address if (and only if) an IEEE public address has been set. If the address is the random static variety then this characteristic will be omitted.
regDataList$	**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. It can be set as an empty string 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

```
DIM RC, manfName$


manfName$ = "Laird Technologies"
RC = blesvcregdevinfo(manfName$)
```

BLESVCREGDEVINFO is an extension function.

BleSvcRegBattery

This function is used to register a Battery service with the GATT server.

The 'Battery' service contains one characteristic as listed at http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.battery_service.xml which allows a battery level value as a percentage to be exposed.

The battery level value can be updated in the characteristic at any time using the BLESVCSETBATTLEVEL function after the service has been registered.

BLESVCREGBATTERY (nInitLevel, fEnableNotify)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments

nInitLevel	**byVal *nInitLevel AS* INTEGER** Specifies the initial value of the battery in percentage. The range of this value is between 0 and 100 which corresponds to 0 to 100%. Values outside this range will result in this function failing.
fEnableNotify	**byVal *fEnableNotify AS* INTEGER** If this is non-zero then the battery level characteristic will have READ and NOTIFY attributes.

Interactive Command: NO

```
DIM RC

RC = blesvcregbattery(80,0)  'the battery service is now registered with GATT with an
                             'initial battery level of 80% - No notification
```

BLESVCREGBATTERY is an extension function.

BleSvcSetBattLevel

This function is used to set the battery level in percentage as reported in the Battery service after it has been registered with the GATT server using the function BLESVCREGBATTERY.

BLESVCSETBATTLEVEL(nNewLevel)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nNewLevel	**byVal *nNewLevel AS* INTEGER** Specifies the value of the battery level in percentage. The range of this value is between 0 and 100 which corresponds to 0 to 100%

Interactive Command: NO

```
DIM RC

RC = blesvcregbattery()       'the battery service is now registered with GATT
RC = blesvcsetbattlevel(50)   'The battery value that will be reported
                              'in the battery service is 50 percent.
```

BLESVCSETBATTLEVEL is an extension function.

BleSvcRegHeartRate

This function is used to register a Heart Rate service with the GATT server.

The 'Heart Rate' service contains three characteristics as listed at http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.heart_rate.xml

The heart rate value can be updated in the characteristic at any time using the BLESVCSETHEARTRATE function after the service has been registered.

Events & Messages

See also Events & Messages for BLE related messages that are thrown to the application when the client configuration descriptor value is changed by a GATT client. The message id that is relevant is (12) as follows:-

MsgId	**Description**
12	Heart Rate Characteristic notification state has changed. 0 is off and 1 is on.

BLESVCREGHEARTRATE (nBodySensorLoc)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nBodySensorLoc **byVal *nBodySensorLoc* AS INTEGER**
Specifies the position of the heart rate sensor as follows:

0	Other
1	Chest
2	Wrist
3	Finger
4	Hand
5	Ear Lobe
6	Foot

Interactive Command: NO

```
DIM RC

RC = blesvcregheartrate(1)     'The position of the heart rate sensor is on the chest
```

BLESVCREGHEARTRATE is an extension function.

BleSvcSetHeartRate

This function is used to set the heart rate in beats per minute as reported in the Heart Rate service after the Heart Rate service has been registered using BLESVCREGHEARTRATE.

BLESVCSETHEARTRATE (nHeartRate)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nHeartRate **byVal *nHeartRate* AS INTEGER**
Specifies the value of the heart rate in beats per minute. The valid range of this parameter is between 0 and 1000.

Interactive Command: NO

```
DIM RC

RC = blesvcsetheartrate(99)                'The heart rate value that will be
reported
                                                'in the heart rate service is 99
```

BLESVCSETHEARTRATE is an extension function.

BleSvcAddHeartRateRR

This function is used to add an RR interval to an array in the heart rate context so that the array will be sent along with the heart rate next time BLESVSETHEARTRATE is called.

According to the specification at http://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.heart_rate_measurement.xml the units for RR interval shall be 1/1024 seconds which equates to slightly less than a millisecond.

BLESVCADDHEARTRATERR (rrInterval)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

rrINterval	**byVal *rrInterval* AS INTEGER** A value in the range 0 to 65535 in units of 1/1024 milliseconds.

Interactive Command: NO

```
DIM RC

RC = blesvcaddheartraterr(100)
RC = blesvcaddheartraterr(110)
RC = blesvcaddheartraterr(105)
RC = blesvcsetheartrate(99)    'send a heart rate of 99 and 3 RR intervals
```

BLESVCADDHEARTRATERR is an extension function.

BleSvcHeartRateContact

This function is used to modify the sensor contact status in the heart rate context so that the Boolean information will be sent along with the heart rate next time BLESVSETHEARTRATE is called.

BLESVCHEARTRATECONTACT (newStatus)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

newStatus	**byVal *newStatus* AS INTEGER** 0 for no contact and 1 for contact

Interactive Command: NO

```
DIM RC

RC = blesvcheartratecontact(1)
RC = blesvcsetheartrate(99)    'send a heart rate of 99 and 3 RR intervals
```

BLESVCHEARTRATECONTACT is an extension function.

BleSvcRegTherm

This function is used to register a Health Thermometer service with the GATT server.

The 'Health Thermometer' service contains four characteristics as listed at http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.health_thermometer.xml

The temperature value can be updated in the characteristic at any time using the BLESVCREGTHERMfunction after the service has been registered.

Events & Messages

See also Events & Messages for BLE related messages that are thrown to the application when the client configuration descriptor value is changed by a GATT client and when the characteristic value is acknowledged by the client. The message ID's that are relevant are (5) and (6) respectively as follows:-

MsgId	**Description**
5	Thermometer Client Characteristic Descriptor value has changed, msgCtx = 0 or 1
6	Thermometer measurement indication has been acknowledged

BLESVCREGTHERM(nTemperatureType)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments

nTemperatureType **byVal *nTemperatureType* AS INTEGER**
The value must be set between 0 and 255 and currently BT SIG allocated values, as of Feb 2013 are:-

1	Armpit
2	Body (General)
3	Ear (Usually ear lobe)
4	Finger
5	Gastro-intestinal Tract
6	Mouth
7	Rectum
8	Toe
9	Tympanum (ear drum)

See http://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.temperature_type.xml for the list of most current values.

Interactive Command: NO

```
DIM rc

rc = blesvcregtherm(2)   'the thermometer service is now registered with GATT
```

BLESVCREGTHERM is an extension function.

BleSvcSetTherm

This function is used to set the temperature, in centigrade, as reported in the Temperature service which has been registered in the GATT server using the function BLESVCREGTHERM.

The value is supplied as two integers, a mantissa and the exponent, which will be stored and transmitted as a 4 byte IEEE floating point value where the mantissa occupies 3 bytes and the exponent the last byte. The two integer values (mantissa and exponent) are interpreted so that the actual temperature value is **mantissa** times ten to the power of **exponent**. The following examples should make it clearer:-

Temperature	Mantissa	Exponent
37.3	373	-1
37300	373	2
37	37	0
1063	1063	0
1063.45	106345	-2

After this function is called wait for the EVBLEMSG message to arrive with msgId set to 6 which confirms that the measurement data has been confirmed by the GATT client.

BLESVCSETTHERM (nMantissa, nExponent, nUnits, dateTime$)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nMantissa	**byVal *nMantissa* AS INTEGER** The value must be set between -9,000,000 and +9,000,000
nExponent	**byVal *nExponent* AS INTEGER** The value must be set between -128 and +127
nUnits	**byVal *nUnits* AS INTEGER** Set to 0 for Centigrade and 1 for Fahrenheit
dateTime$	**byRef *dateTime$* AS STRING** The string contains a date and time stamp which can be optionally provided. It shall be presented to this function in a strict format and if the validation fails, then the information is omitted. To omit this information just provide an empty string. Otherwise the string SHALL consist of exactly 7 characters made up as follows:- Character 1: Century e.g 0x14 Character 2: Year e.g 0x0D Character 3: Month e.g 0x03 Character 4: Day e.g 0x10 Character 5: Hour in the range 0 to 23 Character 6: Minute in the range 0 to 59 Character 7: Seconds in the range 0 to 59 Note: Century/Year =00/00 will be accepted and treated as unknown Month=00 will be accepted and treated as unknown Day=00 will be accepted and treated as unknown For example, 15:36:18pm on 14 March 2013 shall be encoded as a string as follows:- "\14\0D\03\0E\10\24\12"

Interactive Command: NO

```
DIM rc


'//==============================================================================
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
  select nMsgId
  case 0
    DbgMsgVal("Ble Connection ",nCtx)
    inconn = 1

  case 1
    DbgMsgVal("Ble Disonnection ",nCtx)
    inconn = 0
    '// restart advertising
    StartAdverts()

  case 5
    DbgMsgVal(" +++ Indication State ",nCtx)
    indst = nCtx
```

```
   case 6
     DbgMsg(" === Indication Cnf")
     indcnt = indcnt + 1

   case else
     DbgMsg("Unknown Ble Msg" )
   endselect
endfunc 1

OnEvent  EVBLEMSG           call HandlerBleMsg

rc = blesvcregtherm(2)  'the thermometer service is now registered with GATT

rc = blesvcsettherm(364,-1)       'the temperature value that will be reported in
                                     'the temperature service is 36.4°C


'// when the gatt client acknowledges the data the application will get a EVBLEMSG
'// message with msgId set to (6)

'// Please refer to the Thermometer sample app provided
```

BleSvcSetTherm is an extension function.

BleSvcRegTxPower

This function is used to register a Tx Power service with the GATT server so that a client can determine the transmit power level.

The 'Tx Power' service contains a single characteristic as listed at http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.tx_power.xml

It is assumed that the tx power level value will not change while there is a connection and so the transmit level is supplied as a parameter to this function and before this service is registered with the underlying stack, the transmit power will be set to the value requested.

BLESVCREGTXPOWER(nTxLevel)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments

nTxLevel	**byVal *nTxLevel* AS INTEGER** The value must be set as one of the following :- +4, 0, -4, -8, -12, -16, -20 and -40

Interactive Command: NO

```
DIM rc
```

```
rc = blesvcregtxpower(-8)   'the tx power service is now registered with GATT and
                            'trasnmit power is set to -8dBm
```

BLESVCREGTXPOWER is an extension function.

BleSvcRegImmAlert

This function is used to register an Immediate Alert service with the GATT server to implement an optional service for the Proximity Profile.

The 'Immediate Alert' service contains a single characteristic as listed at http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.immediate_alert.xml

It contains a characteristic which can only be written to by a GATT client.

BLESVCREGIMMALERT()

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments None

Interactive Command: NO

```
DIM rc

rc = blesvcregimmalert()
```

BLESVCREGIMMALERT is an extension function.

BleSvcGetImmAlert

This function is used to read the current Alert Level in the Immediate Alert service within the GATT server when the optional service for the Proximity Profile has been registered.

When the value is changed by the GATT Client, an EVBLEMSG message is sent to the *smart* BASIC runtime engine which means this value does not need to be polled if a handler for EVBLEMSG is registered. This is shown in the example below.

BLESVCGETIMMALERT()

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments

nAlertLevel	**byRef *nAlertLevel* AS INTEGER** The value will be 0,1 or 2

Interactive Command: NO

```
DIM rc, alertlvl
Function HandlerBleMsg(byVal msgid as integer, byVal ctx as integer) as integer
 Select msgid
  Case 0
   Print "\n BLE Connection with handle "; integer.h' ctx
  Case 1
   Print "\n BLE Disconnection of handle "; integer.h' ctx
  Case 2
   Print "\n Immediate Alert Service Alert - new level = "; ctx
  Case 3
   Print "\n Link Loss Service Alert - new level = "; ctx
  Case 4
   Print "\n Service error = "; integer.h' ctx
  Case else
   Print "\n Unknown msg id"
 EndSelect
Endfunc 1
OnEvent  EVBLEMSG           call HandlerBleMsg
. . .
rc = blesvcregimmalert()
. . .
rc = blesvcgetimmalert(alertlvl)
. . .
rc = BleAdvertStart( . . . )
waitevent
```

BLESVCGETIMMALERT is an extension function.

BleSvcRegLinkLoss

This function is used to register a Link Loss service with the GATT server to implement an optional service for the Proximity Profile.

The 'Link Loss' service contains a single characteristic as listed at
http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.link_loss.xml

It contains a characteristic which can only be written to by a GATT client.

BLESVCREGLINKLOSS(nInitAlertLevel)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments None

Interactive Command: NO

```
DIM rc
```

```
rc = blesvcreglinkloss(1)   'register link loss service with initial medium alert
```

BLESVCREGLINKLOSS is an extension function.

BleSvcGetLLossAlert

This function is used to read the current Alert Level in the Link Loss Alert service within the GATT server after the service for the Proximity Profile has been registered.

When the value is changed by the GATT Client, an EVBLEMSG message is sent to the *smart* BASIC runtime engine which means this value does not need to be polled if a handler for EVBLEMSG is registered. This is shown in the example below.

BLESVCGETLLOSSALERT()

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments

nAlertLevel	**byRef *nAlertLevel* AS INTEGER** The value will be 0,1 or 2

Interactive Command: NO

```
DIM rc, alertlvl
Function HandlerBleMsg(byVal msgid as integer, byVal ctx as integer) as integer
 Select msgid
  Case 0
   Print "\n BLE Connection with handle "; integer.h' ctx
  Case 1
   Print "\n BLE Disconnection of handle "; integer.h' ctx
  Case 2
   Print "\n Immediate Alert Service Alert - new level = "; ctx
  Case 3
   Print "\n Link Loss Service Alert – new level = "; ctx
  Case 4
   Print "\n Service error = "; integer.h' ctx
  Case else
   Print "\n Unknown msg id"
 EndSelect
Endfunc 1
OnEvent  EVBLEMSG           call HandlerBleMsg
. . .
rc = blesvcregimmalert()
. . .
rc = blesvcgetimmalert(alertlvl)
. . .
rc = BleAdvertStart( . . . )
waitevent
```

BLESVCGETIMMALERT is an extension function.

BleSvcRegBloodPress

This function is used to register a Blood Pressure service with the GATT server.

The 'Blood Pressure' service contains two mandatory characteristics and a single optional characteristic for 'intermediate cuff pressure' as listed at http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.blood_pressure.xml

The optional 'intermediate cuff pressure' characteristic is not implemented but will be provided in a future release if there is a demand.

The blood pressure information can be updated in the characteristics at any time using the BLESVCSETBLOODPRESS function after the service has been registered.

Events & Messages

See also Events & Messages for BLE related messages that are thrown to the application when the client configuration descriptor value is changed by a GATT client and when the characteristic value is acknowledged by the client. The message id's that are relevant are (7) and (8) respectively as follows:-

MsgId	**Description**
7	Blood Pressure Client Characteristic Descriptor value has changed, msgCtx = 0 or 1
8	Blood Pressure measurement indication has been acknowledged

BLESVCREGBLOODPRESS(nFeature, nUserId, nUnits)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments

nFeature

byVal *nFeature* AS INTEGER
The value is made up of a bit mask and must be set between 0 and 0xFFFF and the BT SIG allocated bit masks, as of Mar 2013 are:-

0001	Body Movement Detection Support Bit
0002	Cuff Fit Detection Support Bit
0004	Irregular Pulse Detection Support Bit
0008	Pulse Rate Range Detection Support Bit
0010	Measurement Position Detection Support Bit
0020	Multiple Bond Support Bit

See

	http://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.blood_pressure_feature.xml for the list of most current values.
nUserId	**byVal *nUserId* AS INTEGER** The value shall be in the range 0 to 255, where 255 is 'Unknown User' and 0 to 254 is defined by the service specification. If a value outside this range is provided, then this field in the blood pressure measurement will be omitted.
nUnits	**byVal *nUnits* AS INTEGER** The value shall be 0 for mmHg and 1 for Pascal. Any other values will result in this function returning with an error code and the service will NOT get registered in the GATT table.

Interactive Command: NO

```
DIM rc

rc = blesvcregbloodpress(2,3,0) 'the thermometer service is now registered with GATT
```

BLESVCREGBLOODPRESS is an extension function.

BleSvcSetBloodPress

When connected to a master device and indications have been enabled, this function is used to send new blood pressure measurement data via the blood pressure service which has been registered in the GATT server using the function BLESVCREGBLOODPRESS.

The measurement data consists of many fields which map to arguments of this function. These parameters are simple integers or strings and the intermediate code translates those to appropriate formats as stipulated in the specification at http://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.blood_pressure_measurement.xml

After this function is called wait for the EVBLEMSG message to arrive with msgId set to 8 which confirms that the measurement data has been confirmed by the GATT client.

BLESVCSETBLOODPRESS (nSysPress, nDiasPress, nMeanArtPress, nPulseRate, nMeasStatus, dateTime$)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nSysPress	**byVal *nSysPress* AS INTEGER** The value is the systolic pressure in the units as specified using the nUnits parameter in the BLESVCREGBLOODPRESS function.
nDiasPress	**byVal *nDiasPress* AS INTEGER** The value is the diastolic pressure in the units as specified using the nUnits parameter in the BLESVCREGBLOODPRESS function.
nMeanArtPress	**byVal *nMeanArtPress* AS INTEGER** The value is the mean arterial pressure in the units as specified using the nUnits parameter in the BLESVCREGBLOODPRESS function.
nPulseRate	**byVal *nPulseRate* AS INTEGER** The value is the pulse rate in beats per minute and it can be omitted from the report to the peer by specifying a negative value.
nMeasStat	**byVal *nMeasStat* AS INTEGER** The value is made up of a bit mask and must be set between 0 and 0xFFFF and the BT SIG allocated bit masks, as of Mar 2013 are:- 0001 Body Movement Detection Flag 0002 Cuff Fit Detection Flag 0004 Irregular Pulse Detection Flag 0008 Pulse Rate Range Detection Flag: Exceeds Upper Limit 0010 Pulse Rate Range Detection Flag: Below Lower Limit 0010 Measurement Position Detection Flag Please refer to http://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.blood_pressure_measurement.xml for latest information.
dateTime$	**byRef *dateTime$* AS STRING** The string contains a date and time stamp which can be optionally provided. It shall be presented to this function in a strict format and if the validation fails, then the information is omitted. To omit this information just provide an empty string. Otherwise the string SHALL consist of exactly 7 characters made up as follows:- Character 1: Century e.g 0x14 Character 2: Year e.g 0x0D Character 3: Month e.g 0x03 Character 4: Day e.g 0x10 Character 5: Hour in the range 0 to 23 Character 6: Minute in the range 0 to 59 Character 7: Seconds in the range 0 to 59 Note: Century/Year =00/00 will be accepted and treated as unknown Month=00 will be accepted and treated as unknown Day=00 will be accepted and treated as unknown For example, 15:36:18pm on 14 March 2013 shall be encoded as a string as follows:- "\14\0D\03\0E\10\24\12"

Interactive Command: NO

```
DIM rc,dt$

'//==============================================================================
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
  select nMsgId
  case 0
    DbgMsgVal("Ble Connection ",nCtx)
    inconn = 1
    adv=0

  case 1
    DbgMsgVal("Ble Disonnection ",nCtx)
    inconn = 0
    '// restart advertising
    StartAdverts()

  case 5
    DbgMsgVal(" +++ Indication State ",nCtx)
    indst = nCtx

  case 6
    DbgMsg(" === Indication Cnf")
    indcnt = indcnt + 1

  case else
    DbgMsg("Unknown Ble Msg" )
  endselect
endfunc 1

OnEvent  EVBLEMSG           call HandlerBleMsg

dt$="\14\0D\03\0E\10\24\12"
rc = blesvcsetbloodpress(120,80,100,72,0,dt$)
  '//where systolic pressure = 120, diastolic pressure = 80
  '//means arterial pressure = 100, pulse rate = 72, measurement status = 0

'// when the gatt client acknowledges the data the application will get a EVBLEMSG
'// message with msgId set to (8)

'// Please refer to the blood pressure sample app provided
```

BLESVCSETBLOODPRESS 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)

BleBondMngrErase

This function is used to delete 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)

FUNCTION

Returns:	INTEGER An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nFutureUse	**byVal nFutureUse AS INTEGER** This shall be set to 0.

Interactive Command: NO

```
DIM rc
rc = BleBondMngrErase(0)   '//delete the entire trusted device database
```

BLEBONDMNGRERASE is an extension function.

BleBondMngrGetInfo

This function is used to retrieve the mac address and other information of a device from the trusted device database based on an index.

Please note that there should be no reliance on the same device in the database being retrieved by the same index because as new bonding occur if they get updated the position will change in the database.

BLEBONDMNGRGETINFO (nIndex, addr$, nExtraInfo)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nIndex	**byVal nIndex AS INTEGER** This is an index and shall be 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, then this variable will contain a mac address which will be exactly 7 bytes long. The first byte is identifies it as a public or private random address and the next 6 bytes the actual address.
nExtraInfo	**byRef nExtraInfo AS INTEGER** On exit if nIndex points to a valid entry in the database, then this variable will contain a composite integer value where the lower 16 bits are the eDIV, biut 16 will be set if the IRK (Identity Resolving Key) exists for the trusted device and bit 17 will be set if the CSRK (Connection Signing Resolving Key) exists for the trusted device.

Interactive Command: NO

```
DIM rc,exInfo,addr$

rc = BleBondMngrGetIngo(1,addr$,exInfo)   '//Extract info of device at index 1
```

BLEBONDMNGRGETINFO is an extension function.

Virtual Serial Port Service - Managed

This section describes all the events and routines used to interact with a managed virtual serial port service.

By 'managed' it means there is a driver consisting of transmit and receive ring buffers that isolate the BLE service from the *smart*BASIC application which presents easy to use api functions.

Please note: the driver makes the same assumption that the driver in a PC makes, that is, assuming that the on-air connection equates to the serial cable, then in the same manner that a PC cannot detect that after a cable disconnection and then a reconnection, there is no assumption that the cable is from the same source as prior to the disconnection.

The module has the capability of presenting a serial port service in the local GATT Table, consisting of two characteristics. One characteristic is the TX FIFO and the other is the RX FIFO, both consisting of an attribute taking up to 20 bytes.

By default, (can be changed using the AT+SET 112 command), 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`

If command AT+SET 112="0x0001" 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`

(Please note that 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. Which 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 and the following properties are enabled:-

WRITE
WRITE_NO_RESPONSE

The 'tx fifo characteristic' value attribute is with no authentication or encryption requirements and the following properties are enabled:-

NOTIFY (The CCCD descriptor also requires no authentication/encryption)

Given that the outgoing data is **notified** to the client, the 'tx fifo characteristic' has a Client Configuration Characteristic (CCCD) which will need to be set to 0x0001 to allow the module to send any data waiting to be sent in the transmit rign buffer. While the CCCD value is not set for notifications, any writes by the *smart*BASIC application will result in data being buffered. If the buffer is full the appropriate write routine will indicate how many bytes actually got absorbed by the driver. In the background the transmit ring buffer will be emptied with one or more indicate or notify messages to the client. When the last bytes from the ring buffer are sent an **EVVSPTXEMPTY** is thrown to the smartBASIC 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 then the data will be discarded. After the ring buffer is updated an event **EVVSPRX** is thrown to the smartBASIC runtime engine so that an application can read and process the data.

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 will allow the application developer to interact with any GATT client implementing a serial port service, currently deployed or for that matter with one that the Bluetooth SIG adopts.

Command Mode Operation

Just as the physical UART is used to interact with the module when it is not running a smartBASIC application, it is also possible to have **limited** interaction with the module in command mode. The limitation applies to NOT being able to launch smartBASIC applications using the AT+RUN command.

The main purpose of command mode operation is to facilitate the download of an autorun smartBASIC 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. Note, that it is the smartBASIC application that is downloaded over the air and 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+SET.

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 modifying the config key 100 using the AT+SET command.

Downloading of smartBASIC applications is facilitated using a Windows application which will be available for free from Laird. The PC will need to be BLE enabled using a Laird supplied adapter, please contact your local FAE for more details.

As most of the AT commands are functional it is possible to obtain information like version number by sending the command AT I 3 to the module over the air.

Finally note that the module enters command mode only if there is no autorun application or if the autorun application exits to command 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 subsequenct 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 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 detailed list of all events generated by the virtual serial port service manged 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.

```
// Example :: EVVSPRX & EVVSPTXEMPTY events

dim tx$

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,0,hndl)
//hndl can now be used to advert the 128 bit Service uuid
//using the function BleAdvRptAddUuid128()

//handler to data arrival
function HandlerBleVSpRx() as integer
  //just print the data that arrived
  dim n,rx$
  n = BleVSpRead(rx$,20)
  print "\nRX=";StrEscape$(rx$)
endfunc 1

//handler to service VSP tx buffer is empty
function HandlerVSpTxEmpty() as integer
  //echo the data back
  tx$="this is more data"
  n = BleVSpWrite(tx$)
endfunc 1

OnEvent  EVVSPRX  call HandlerBleVSpRx
OnEvent  EVVSPTXEMPTY  call HandlerVSpTxEmpty

tx$="send this data"
n = BleVSpWrite(tx$)

//wait for events and messages
waitevent
```

BleVSpOpen

This function is used to open the default virtual serial port service using the parameters specified. The service's UUID will be :- `6e40`***`0001`***`-b5a3-f393-e0a9-e50e24dcca9e`

If the virtual serial port is already open then this function will fail.

BLEVSPOPEN (txbuflen,rxbuflen,nFlags,svcUuid)

Function

Returns: INTEGER Indicates 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 then a default value will be 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 then a default value will be 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 which means use 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 will be updated with a handle to the service UUID which is 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 and 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

dim tx$
dim scRpt$,adRpt$,addr$

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,0,hndl)

//Advertise the service in a scan report
rc = BleAdvRptAddUuid128(scRpt$,hndl)
adRpt$=""
rc = BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""  //because we are not doing a DIRECT advert
 rc = BleAdvertStart(0,addr$,100000,300000,0)
```

```
//Now advertising so can be connectable

//wait for events and messages
waitevent
```

BLEVSPOPEN is an extension function.

BleVSpClose

This subroutine is used to close 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 any data in either of these buffers when the port is closed, it will be lost.

BLEVSPCLOSE()

Subroutine

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments: **None**

Interactive Command: No

Related Commands: BLEVSPINFO, BLEVSPOPEN, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH

```
// Example :: BLEVSPCLOSE

dim tx$

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,hndl)

//handler to service VSP tx buffer is empty
function HandlerVSpTxEmpty() as integer
  //data has been sent, so close the vsp
  BleVSpClose()
  BleVSpClose()   //safe to call when closed
endfunc 1

OnEvent  EVVSPTXEMPTY  call HandlerVSpTxEmpty

tx$="send this data and will close when sent"
n = BleVSpWrite(tx$)

//wait for events and messages
waitevent
```

BLEVSPCLOSE is a extension subroutine.

BleVSpInfo

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)

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 information 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

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,0,hndl)

//print the state
print "VSP state is ";BleVSpInfo(0)
//print the size of rx buffer
print "VSP Rx Buffer Size is ";BleVSpInfo(1)
//print the size of tx buffer
print "VSP Tx Buffer Size is ";BleVSpInfo(2)
//print the number of bytes waiting in the rx buffer
print "VSP bytes in rx buffer ";BleVSpInfo(3)
//print free space in tx buffer
print "VSP free space in tx buffer ";BleVSpInfo(4)

//close the VSP
rc = BleVSpClose()

//print the state
print "VSP state is ";BleVSpInfo(0)
```

```
//wait for events and messages
waitevent
```

BLEVSPINFO is a extension subroutine.

BleVSpWrite

This function is used to transmit a string of characters from the virtual serial port.

BLEVSPWRITE (strMsg)

Function

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

Arguments:

strMsg ***byRef strMsg AS STRING***
The array of bytes to be sent. STRLEN(strMsg) bytes will be written to the local transmit ring buffer. If STRLEN(strMsg) and the return value are not the same then 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, 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. Another strategy would be to wait for EVVSPTXEMPTY events to then resubmit the data

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: BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPREAD, BLEVSPFLUSH

```
// Example :: BLEVSPWRITE

dim tx$

//Open the VSP
dim hndl,cnt
rc = BleVSpOpen(128,128,hndl)

//handler to service VSP tx buffer is empty
function HandlerVSpTxEmpty() as integer
  cnt=cnt+1
  if cnt<= 2 then
    tx$="then this is sent"
    n = BleVSpWrite(tx$)
  endif
endfunc 1

OnEvent  EVVSPTXEMPTY  call HandlerVSpTxEmpty

cnt=1
tx$="send this data and "
n = BleVSpWrite(tx$)

//wait for events and messages
waitevent
```

BLEVSPWRITE is a extension subroutine.

BleVSpRead

This function is used to read the content of the receive buffer and **copy** it to the string variable supplied.

BLEVSPREAD(strMsg,nMaxRead)

Function

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 that need to be processed.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

strMsg ***byRef strMsg AS STRING***
The content of the receive buffer will get **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 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: BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPFLUSH

```
// Example :: BLEVSPREAD

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,hndl)
//hndl can now be used to advert the 128 bit Service uuid
//using the function BleAdvRptAddUuid128()
```

```
//handler to service an advert timeout
function HandlerBleVSpRx() as integer
  //just print the data that arrived
  dim n,rx$
  n = BleVSpRead(rx$,20)
  print "\nRX=";StrEscape$(rx$)
endfunc 1

OnEvent  EVVSPRX  call HandlerBleVSpRx

//wait for events and messages
waitevent
```

BLEVSPREAD is an extension subroutine.

BleVSpFlush

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. A flush of the receive buffer is the best approach to recover from that situation.

BLEVSPFLUSH(bitMask)

Subroutine

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

bitMask ***byVal bitMask AS INTEGER***
Bit 0 is set to flush the rx buffer and Bit 1 to flush the tx buffer.

Interactive Command: No

Related Commands: BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPREAD

```
// Example :: BLEVSPFLUSH

//Open the VSP
dim hndl
rc = BleVSpOpen(128,128,hndl)
//hndl can now be used to advert the 128 bit Service uuid
//using the function BleAdvRptAddUuid128()

//handler to service an advert timeout
function HandlerBleVSpRx() as integer
  //just flush the data
  BleVspFlush(1)
endfunc 1
```

```
OnEvent  EVVSPRX  call HandlerBleVSpRx

//wait for events and messages
waitevent
```

BLEVSPFLUSH is a 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

This function is used to alter the power state of the module as per the input parameter.

SYSTEMSTATESET (nNewState)

FUNCTION

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nNewState **byVal *nNewState* AS INTEGER**
New state of the module as follows:-
0 System OFF (Deep Sleep Mode)

Please note: This state can also be entered when the UART is open and a BREAK condition is asserted. Deasserting BREAK will make the module resume through reset as if it had been power cycled.

Interactive Command: NO

```
// Example :: SystemStateSet ()
dim rc

//put the module into deep sleep
rc = SystemStateSet(0)
```

SYSTEMSTATESET is an extension function.

Miscellaneous Routines

ReadPwrSupplyMv

This function is used to read the power supply voltage and the value will be returned in millivolts.

READPWRSUPPLYMV ()

FUNCTION

Returns: INTEGER

The power supply voltage in millivolts.

Arguments: None

Interactive Command: NO

```
// Example :: ReadPwrSupplyMv()
dim supplyMV

//read and print the supply voltage
supplyMV = ReadPwrSupplyMv()
print "\nSupply voltage is ";supplyMV;"mV"
```

READPWRSUPPLYMV is an extension function.

SetPwrSupplyThreshMv

This function is used to set 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 msd 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 and the context data will be the current reading of the supply voltage in millivolts

SETPWRSUPPLYTHRESHMV(nThresh)

FUNCTION

Returns: INTEGER

0 if the threshold was successfully set and 0x6605 if the threshold value cannot be implemented.

Arguments: None

nThreshMv **byVal *nThresMv* AS INTEGER**
The BLE_EVMSG event will be thrown to the runtime engine if the supply voltage drops below this value. Valid values are 2100, 2300, 2500 and 2700.

Interactive Command: NO

```
// Example :: SetPwrSupplyThreshMv()
dim rc
dim mv

//handler for generic BLE messages
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
  select nMsgId
  case BLE_EVBLEMSGID_POWER_FAILURE_WARNING
    print "\n --- Power Fail Warning ",nCtx
    mv=ReadPwrSupplyMv()
    print "\n --- Supply voltage is ";mv;"mV"
    print "\n --- count is ";count
  case else
    //ignore this message
  endselect
endfunc 1

OnEvent  EVBLEMSG           call HandlerBleMsg

mv=ReadPwrSupplyMv()
print "\nSupply voltage is ";mv;"mV\n"

mv=2700
rc=SetPwrSupplyThreshMv(mv)

print "\nWaiting for power supply to fall below ";mv;"mV"

//wait for events and messages
waitevent
```

SETPWRSUPPLYTHRESHMV is an extension function.

8. Events & Messages

smart BASIC has been designed so that it is event driven which makes it suitable for embedded platforms where it is normal to wait for something to happen and then respond to that.

To ensure that access to variables and resources ends up in race conditions, the event handling is done synchronously which means the *smart* BASIC runtime engine has to process a WAITEVENT statement for any events or messages to be processed. This mechanism guarantees that the code *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 command mode operation or how the 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 using the command AT&F* or AT&F 1

To write to these objects, which are identified by a positive number, the module has to be in command mode and the command AT+SET msut be used which is described in detail here.

To read current values of these objects use the command AT+GET described here.

Predefined configuration objects are as listed under details of the AT+SET command.

10. MISCELLEANEOUS

Bluetooth Result Codes

There are some operations and events that provide a single byte Bluetooth HCI resultcode, for example the EVDISCON message. The meaning of the resultcode is as per the list reproduced from the Bluetooth Specifications below and no guarantee is supplied as to its accuracy given the list was manually copied. If there is doubt the reader is encouraged to consult the actual specification.

Resultcodes presented in `grey` are not relevant to Bluetooth Low Energy operation and so 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**

INDEX

Module specific functions appear in the index with their prefixing underscore.

#SET 55
? (Read Variable) 41
= (Set Variable) 42
ABORT 44
ABS 75
Arrays 53
AT + BTD * 46, 47
AT + MAC 47
AT I 31
AT Z 46
AT&F 45
AT+DBG 35
AT+DEL 33
AT+DIR 33
AT+FCL 40
AT+FOW 39
AT+FWR 39
AT+FWRH 40
AT+GET 38
AT+REN 44
AT+RUN 34
AT+SET 36
ATI 31
ATZ 46
BASIC 6
BLEADVERTSTART 163
BLEADVERTSTOP 165
BLEADVRPTADDUUID16 ....167, 169
BLEADVRPTAPPENDAD 169
BLEADVRPTINIT 165
BLEADVRPTSCOMMIT 170
BLEATTRMETADATA 192
BLEAUTHENTICATE 180
BLEBONDMNGRERASE 254
BLEBONDMNGRGETINFO 254
BLECHARCOMMIT 201
BLECHARDESCADD 200
BLECHARDESCPRSTNFRMT 198
BLECHARDESCUSERDESC 197
BLECHARNEW 195
BLECHARVALUEINDICATE 207
BLECHARVALUENOTIFY 206
BLECHARVALUEREAD 203, 208
BLECHARVALUEWRITE 205
BLECONFIGDCDC 162
BLEDECODE32 228
BLEDECODEBITS 234
BLEDECODEFLOAT 229
BLEDECODES16 224
BLEDECODES24 226
BLEDECODES8 222
BLEDECODESFLOAT 231
BLEDECODESTRING 233
BLEDECODETIMESTAMP 232
BLEDECODEU16 225
BLEDECODEU24 227
BLEDECODEU8 223
BLEDISCONNECT 172
BLEENCODE16 211
BLEENCODE24 212
BLEENCODE32 213
BLEENCODE8 210
BLEENCODEBITS 221
BLEENCODEFLOAT 214
BLEENCODESFLOAT 217
BLEENCODESFLOATEX 216
BLEENCODESTRING 220
BLEENCODETIMESTAMP 218
BLEGAPSVCINIT 236
BLEGETCURCONNPARMS 174
BLEHANDLEUUID128 187
BLEHANDLEUUID16 187
BLEHANDLEUUIDSIBLING 188
BLESCANRPTINIT 167
BLESECMNGRBONDREQ 180
BLESECMNGRIOCAP 179
BLESECMNGRKEYSIZES . 171, 177, 178, 187, 240, 243, 250, 266
BLESECMNGRPASSKEY 177
BLESETCURCONNPARMS 172
BLESVCADDINCLUDESVC 191
BLESVCCOMMIT 189
BLESVCGETIMMALERT 247
BLESVCGETLLOSSLERT 249
BLESVCREGBATTERY 239
BLESVCREGBLOODPRESS 250
BLESVCREGDEVINFO 237
BLESVCREGHEARTRATE 240
BLESVCREGIMMALERT 247
BLESVCREGLINKLOSS 248
BLESVCREGTHERM 243
BLESVCREGTXPOWER 246
BLESVCSETBATTLEVEL 239
BLESVCSETBLOODPRESS 251
BLESVCSETHEARTRATE 241
BLESVCSETTHERM 244

BLETXPOWERSET 162
BleVSpClose 260
BleVSpFlush 264
BleVSpInfo 260
BleVSpOpen 258
BleVSpRead 263
BleVSpWrite 262
Bluetooth Result Codes 269
BP 70
BREAK 63
byRef 49
byVal 49
CIRCBUFCREATE 107
CIRCBUFDESTROY 107
CIRCBUFITEMS 111
CIRCBUFOVERWRITE 109
CIRCBUFREAD 110
CIRCBUFWRITE 108
Complex Variables 52
CONTINUE 63
Declaring Variables 53
Decoding Functions 222
DIM 51
DO / DOWHILE 58
DO / UNTIL 58
Encoding Functions 210
ENDFUNC 151
ENDSUB 150
EVBLE_ADV_TIMEOUT 153
EVBLEMSG 153
EVBLEMSG 153
EVCHARCCCD 156
EVCHARDESC 158
EVCHARHVC 156
EVCHARSCCD 157
EVCHARVAL 155
EVDISCON 155
EVNOTIFYBUF 160
EVVSPRX 159
EVVSPTXEMPTY 159
Exceptions 49
EXITFUNC 151
EXITSUB 150
FICR register 73
FOR / NEXT 59
FUNCTION 150
GETLASTERROR 71
GETTICKCOUNT 105
GETTICKSINCE 105
GPIO Events 147
GPIOBINDEVENT 147
GPIOREAD 146
GPIOSETFUNC 145
GPIOUNBINDEVENT 148
GPIOWRITE 147
I2C Events 125
I2CCLOSE 127
I2COPEN 125
I2CREADREG16 130
I2CREADREG32 132
I2CREADREG8 128
I2CWRITEREAD 133
I2CWRITEREG16 129
I2CWRITEREG32 131
I2CWRITEREG8 127
IF THEN / ELSEIF / ELSE / ENDIF 60
LEFT$ 77
MAX 75
MID$ 77
MIN 76
Numeric Constants 54
NVRECORDGET 142
NVRECORDGETEX 142
NVRECORDSET 143
ONERROR 64
ONEVENT 66
ONFATALERROR 65
PRINT 67
RAND 99
RANDEX 99
RANDSEED 100
READPWRSUPPLYMV 266
RESET 98
RESETLASTERROR 72
RESUME 43
RIGHT$ 79
SELECT / CASE / CASE ELSE / ENDSELECT 62
SENDMSGAPP 74
SETPWRSUPPLYTHRESHMV 266
Simple Variables 52
SO 43
SPI Events 136
SPICLOSE 137
SPIOPEN 136
SPIREAD 140
SPIREADWRITE 138
SPIWRITE 139
SPRINT 69
STOP 70
STRCMP 85
STRDEESCAPE 89
STRDEHEXIZE$ 87
STRESCAPE$ 88
STRFILL 84

STRGETCHR....82
STRHEX2BIN....88
STRHEXIZE....86
String Constants....54
STRLEN....79
STRPOS....80
STRSETBLOCK....83
STRSETCHR....81
STRSHIFTLEFT....85
STRSPLITLEFT$....91
STRSUM....92
STRVALDEC....90
STRXOR....93
SUB....149
Syntax....48
SYSINFO....72
SYSTEMSTATESET....265
TABLEADD....96
TABLEINIT....95
TABLELOOKUP....97
Timer Events....101
TIMERCANCEL....104
TIMERRUNNING....103
TIMERSTART....102
UART Events....113
UARTBREAK....124
UARTCLOSE....115
UARTFLUSH....121
UARTGETCTS....122
UARTINFO....116
UARTOPEN....113
UARTREAD....118
UARTREADMATCH....120
UARTSETRTS....123
UARTWRITE....117
Variables....50
VSP (Virtual Serial Port) Events....257
WAITEVENT....66
WHILE / ENDWHILE....61