smartBASIC

Core Funtionality

User Manual

Version 2.0-r1

global solutions: local support

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

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

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

REVISION HISTORY

Version	Revisions Date	Change History
2.0-r1	1 Feb 2013	Created by splitting from BL600 user manual

CONTENTS

1. INTRODUCTION

This user manual provides detailed information on the core aspects of Laird Technologies' *smart* BASIC language which is embedded inside Laird modules. This manual is designed to make handling BLE-enabled end products a straightforward process and it includes the following:

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

An appropriate specific user manual provides detailed information on module specific smartBASIC extensions relating to Bluetooth, BLE etc. This second manual is supplied along with this one in the Firmware zip file.

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 the 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 some module platforms which have limited code flash space, the parsing from source code to bytecode is done on a Windows PC using a free cross-compiler supplied by Laird. Other platforms with more firmware code space also offer on-board compiling capabilities in addition to the external cross-compilation utility.

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

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

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

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

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

Why Do We Need *smart*BASIC?

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

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

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

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

Why Write Applications?

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

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

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

What does a BLE Module Contain?

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

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

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

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

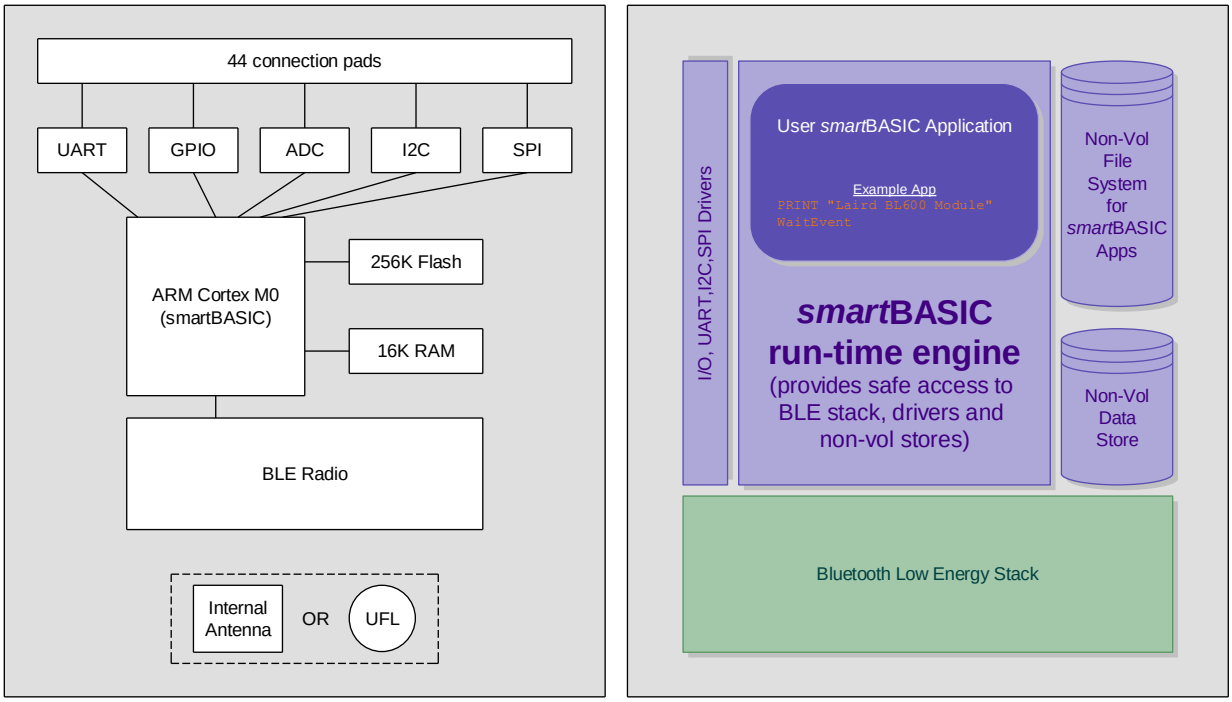


Figure 1: BLE smartBASIC module block diagram

*smart*BASIC Essentials

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

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

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

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

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

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

Developing with *smart*BASIC

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

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

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

*smart*BASIC Operating Modes

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

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

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

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

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

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

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

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

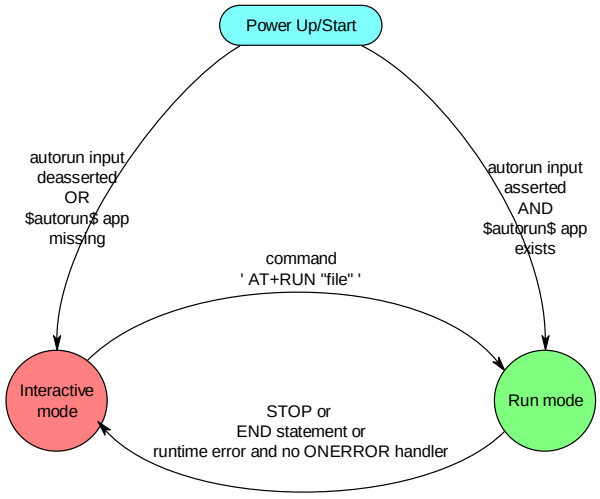


Figure 2: Module modes & transitions

Types of Applications

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

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

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

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

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

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

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

Non Volatile Memory

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

The flash memory is available for three purposes:

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

Using the Module's Flash File System

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

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

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

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

2. GETTING STARTED

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

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

Requirements

To replicate this example, you need the following items:

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

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

Connecting Things Up

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

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

UWTerminal

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

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

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

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

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

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

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

Figure 3: UWTerminal opening screen

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

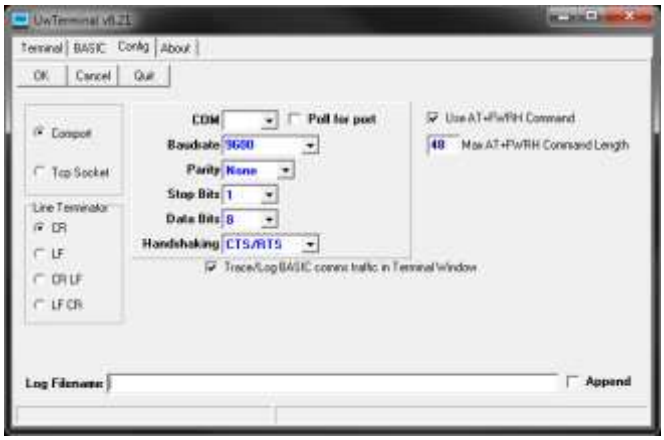


Figure 4: UWTerminal Configuration screen

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

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

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

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

Getting Around UWTerminal

Figure 5: UWTerminal tabs and status lights

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

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

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

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

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

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

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

- **RI** – Ring Indicate.

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

Figure 6: White lights

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

Figure 7: Control options

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

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

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

- **BREAK** – Used to assert a break condition over the Rx line at the module. It must be deasserted after use. A Tx pin is normally at logic high (> 3v for RS232 voltage levels) when idle; a BREAK condition is where the Tx output pin is held low for more than the time it takes to transmit 10 bits.
  If the BREAK checkbox is ticked then the Tx output is at non-idle state and no communication is possible with the UART device connected to the serial port.
- **LocalEcho –** Enables local echoing of any characters typed at the terminal. In default operation, this option box should be selected because modules do not reflect back commands entered in the terminal emulator.

- **LineMode** – Delays transmission of characters entered into UWTerminal until you press **Enter**. Enabling LineMode means that Backspace can be used to correct mistakes. We recommend that you select this option.
- **Clear** – Removes all characters from the terminal screen.
- **ClosePort** – Closes the serial port. This is useful when a USB to serial adaptor is being used to drive the development board which has been briefly disconnected from the PC.
- **OpenPort** – Re-opens the serial port after it has been manually closed.

Useful Shortcuts

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

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

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

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

   accept com=n baud=bbb linemode

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

Figure 8: Shortcut properties

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

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

Figure 9: Terminal screen status bar

Using UWTerminal

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

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

Figure 10: Interactive command access

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

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

The next chapter lists all of the Interactive commands.

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

```
at+dir

06        $factory$
00
```

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

Your First *smart*BASIC Application

Create 'Hello World' App

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

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

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

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

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

Note the following:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 11: Example of a smartBASIC code fragment in TextPad

Download 'Hello World' App

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

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

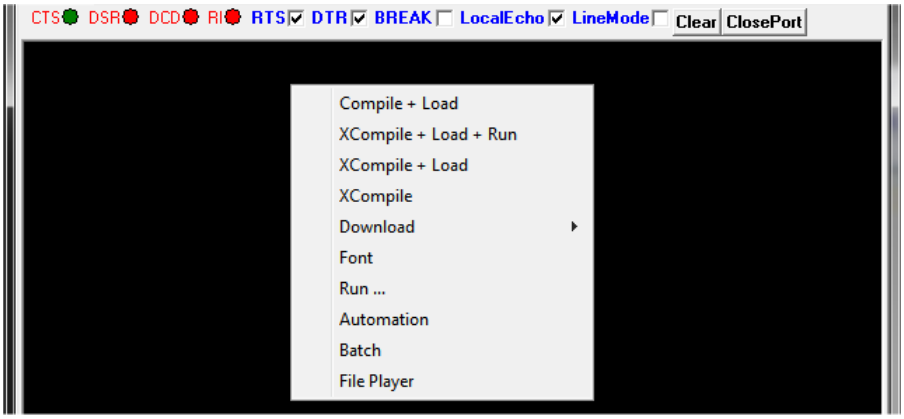


Figure 12: Right-click UWTerminal screen

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

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

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

```
AT I 0
10     0      Bl600Med
AT I 13
10     13     9E56 5F81
<<Cross Compiling [test1.sb]>>

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

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

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

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

```
AT I 0
10     0      Bl600Med
AT I 13
10     13     9E56 5F81
??? Cross Compiler [XComp_Bl600Med_9E56_5F81.exe] not found ???
??? Please save a copy to the same folder as UwTerminal.exe ???
??? If you cannot locate the file, please contact the supplier ???
```

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

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

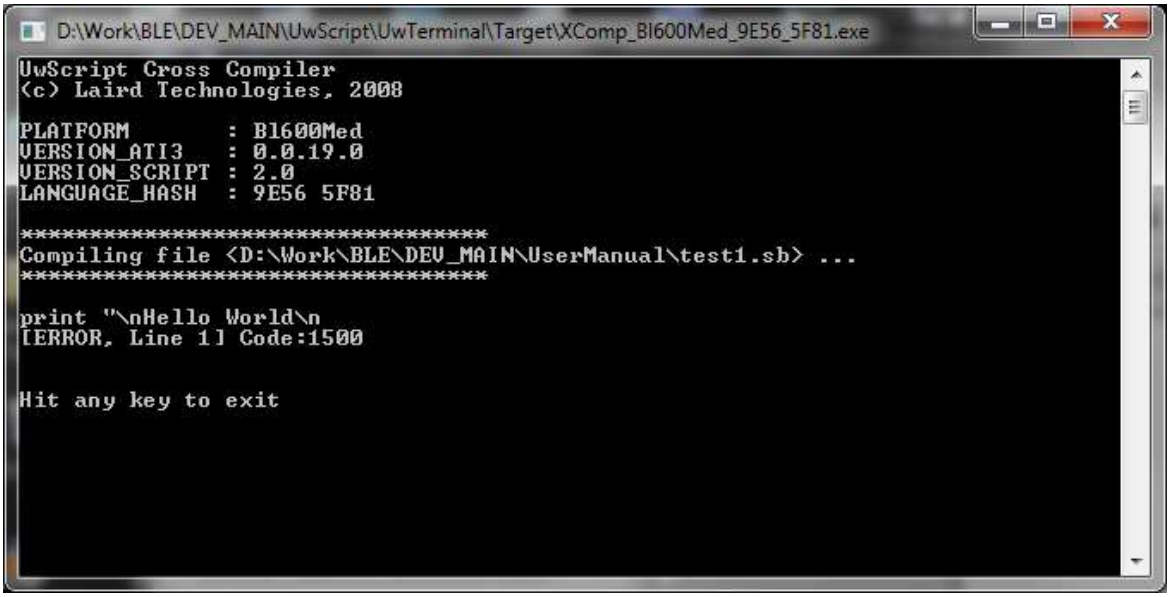


Figure 13: Compilation error window

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

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

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

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

Test1
Hello World
00
```

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

```
06     test1
00
```

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

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

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

*smart*BASIC Statement Format

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

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

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

The various elements of each line are:

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

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

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

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

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

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

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

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

```
//Example :: HelloWorld.sb

DIM i as integer           //declare our variable

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

Some notes regarding the previous program:

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

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

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

The following output displays:

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

If you now change the print statement in the application to

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

… the following output displays:

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

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

```
06     test1
06     HelloWorld
00
```

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

Autorun

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

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

In our development kits, 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 module. The DTR checkbox is always selected by default (in asserted state), which translates to a 0v at the **nAutoRUN** input of the module. This means if an autorun application exists in the module's file system, it is automatically launched on power up.

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

```
at+dir

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

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

$autorun$.HelloWorld

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

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

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

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

The following output displays:

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

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

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

Debugging Applications

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

Because of the absence of labels, *smart* BASIC provides facilities for debugging an application by inserting breakpoints into the source code prior to compilation and execution. Multiple

breakpoints can be inserted and each breakpoint can have a unique identifier associated with it. These IDs can be used to aid the developer in locating which breakpoint resulted in the break. It is up to the programmer to ensure that all IDs are unique. The compiler does not check for repeated values.

Each breakpoint statement has the following syntax:

BP nnnn

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

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

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

<linefeed>21 BREAKPOINT nnnn<carriage return>

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

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

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

 DIM i as integer

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

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

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

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

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

21    BREAKPOINT 3333
```

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

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

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

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

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

Structuring an Application

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

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

smart BASIC uses a single pass compiler which can be extremely efficient in systems with limited memory. They are called "single pass" as the source application is only passed through the parser line by line once. That means that it has no knowledge of any line which it has not yet

encountered and it forgets any previous line as soon as the first character of the next line arrives. The implication is that variables and subroutines need to be placed in position before they are first referenced by any function which dictates the structure of a typical application.

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

- **Opening Comments** – Any initial text comments to help document the application.
- **Includes** – The cross compiler which is automatically invoked by UWTerminal allows the use of #DEFINE and #INCLUDE directives to bring in additional source files and data elements.
  Variable Declarations – Declare any global variables. Local variables can be declared within subroutines and functions.
- **Subroutines and Functions** – These should be cited here, prior to any program references. If any of them refer to other subroutines or functions, these referred ones should be placed first. The golden rule is that nothing on any line of the application should be "new". Either it should be an inbuilt
  smart BASIC function or it should have been defined higher up within the application.
- **Event and error handlers** – Normally these reference subroutines, so they should be placed here.
- **Main program** – The final part of the application is the main program. In many cases this may be as simple as an invocation of one of the user functions or subroutines and then finally the WAITEVENT statement.

An example of an application *(btn.button.led.test.sb)* which monitors button presses and reflects them to leds on the BL600 development kit is as follows:

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

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

//*****************************************************************************
// Library Import
```

```
//*************************************************************************
//#include "$.lib.ble.sb"

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

dim rc                                  // declare rc as integer variable

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

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

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

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

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


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

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

```
//*****************************************************************************

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

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

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

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

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

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

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

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

Figure 14: Optional verbose explanation

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

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

AT

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

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

AT I or ATI

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

AT i num

Command

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

Where

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

Arguments

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

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

606	Flash File System: FAT Segment: Deleted Space
631	NvRecord Memory Store Segment: Total Space
632	NvRecord Memory Store Segment: Free Space
633	NvRecord Memory Store Segment: Deleted Space
1000..1999	See SYSINFO() function definition
2000..2999	See SYSINFO() function definition

Any other number currently returns the manufacturer's name.

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

00	Public IEEE format address
01	Random static address (default as shipped)
02	Random Private Resolvable (used with bonded devices) –
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

COMMAND

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

AT+DIR <"string">

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

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

Arguments:

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

string.

The match string is not case sensitive.

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

Interactive Command: YES

```
'Examples:

AT+DIR

AT+DIR "new"
```

AT+DIR is a core command.

AT+DEL

COMMAND

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

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

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

10 6 aaaa,bbbb,cccc

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

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

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

AT+DEL "filename" (+)

Returns OK

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

Arguments:

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

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

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

Interactive Command: YES

```
'Examples:

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

AT+DEL is a core command.

AT+RUN

COMMAND

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

AT+RUN "filename"

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

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

Arguments:

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

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

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

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

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

Interactive Command: YES

```
'Examples:

AT+RUN "NewApp"
```

```
or
NewApp
```

AT+RUN is a core command.

AT+DBG

COMMAND

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

AT+DBG "filename"

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

Arguments:

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

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

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

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

Interactive Command: YES

```
'Examples:

AT+DBG "NewApp"
```

AT+DBG is a core command.

AT+SET

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

AT+GET

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

AT+CFG

COMMAND

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

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

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

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

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

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

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

Arguments:

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

value Integer_constant
This is the new value for the configuration key and the syntax allows decimal, octal, hexadecimal or binary values.

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

The following Configuration Key IDs are defined.

40	Maximum size of locals simple variables
41	Maximum size of locals complex variables
42	Maximum depth of nested user defined functions and subroutines
43	The size of stack for storing user functions simple variables
44	The size of stack for storing user functions complex variables
45	The size of the message argument queue length

Interactive Command: YES

AT+CFG is a core command.

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

AT+FOW

COMMAND

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

AT+FOW "filename"

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

Arguments:

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

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

Interactive Command: YES

```
'Examples:

AT+FOW "myapp"
```

AT+FOW is a core command.

AT+FWR

COMMAND

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

AT+FWR "string"

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

Arguments:

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

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

Interactive Command: YES

```
'Examples:

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

AT+FWR is a core command.

AT+FWRH

COMMAND

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

AT+FWRH "string"

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

Arguments

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

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

Interactive Command: YES

```
'Examples:

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

'Invalid example

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

AT+FWRH is a core command.

AT+FCL

COMMAND

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

AT+FCL

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

Arguments:

None

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

Interactive Command: YES

```
'Examples:

AT+FCL
```

AT+FCL is a core command.

? (Read Variable)

COMMAND

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

? var <[index]>

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

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

\n08\tnnnnnn\r
\n00\r

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

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

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

\n01\tE023\r

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

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

? h' var returns

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

Arguments:

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

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

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

B' := Binary

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

Interactive Command: YES

```
'Examples:
? argc
08      11
00
? h'argc
08      H'0000000B
00
? B'argc
08      B'00000000000000000000001011
? argv[0]
08      "hello"
00
```

? is a core command.

= (Set Variable)

COMMAND

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

= var<[n]> value

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

\n00\r

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

\n01\tE027\r

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

\n01\tE023\r

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

\n01\tE26\r

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

Arguments:

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

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

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

Interactive Command: YES

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

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

= is a core command.

SO

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

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

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

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

SO is a core function.

RESUME

COMMAND

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

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

RESUME

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

\n01\tE029\r

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

Interactive Command: YES

```
'Examples:

RESUME
```

RESUME is a core function.

ABORT

COMMAND

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

ABORT

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

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

Interactive Command: YES

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

AT+RUN "someapp"
ABORT
```

ABORT is a core command.

AT+REN

COMMAND

Renames an existing file.

AT+REN "oldname" "newname"

Returns OK if the file is successfully renamed.

Arguments

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

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

: * ? " < > |

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

Interactive Command: YES

```
'Examples:

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

AT+REN is a core command.

AT&F

COMMAND

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

AT&F integermask

Returns OK if file successfully erased.

Arguments

Integermask Integer corresponding to a bit mask or the "*" character

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

1	Erases normal file system and system config keys (see AT+CFG for examples of config keys)
16	Erases the User config keys only
*	Erases all data segments
Else	Not applicable to current modules

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

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

Interactive Command: YES

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

AT&F is a core command.

AT Z or ATZ

Resets the CPU.

AT Z

Returns \n00\r

Arguments: None

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

Interactive Command: YES

```
'Examples:

AT Z
```

AT Z is a core command.

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, found in your module's extention manual, that are designed to expose specific functionality of the platform. For example, Bluetooth Low Energy's GATT, GAP, and security functions.

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

Syntax

smart BASIC commands are classified as one of the following:

- Functions
- Subroutines
- Statements

Functions

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

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

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

Subroutines

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

STRSHIFTLEFT (***string$***, ***num***)

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

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

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

STRSHIFTLEFT (***string$***, ***num***)

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

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

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

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

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

Statements

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

Exceptions

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

return of meaningful value, but the event needs to be trapped so that the effects of doing that operation can be lessened.

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

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

Language Definitions

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

Command

FUNCTION / SUBROUTINE / STATEMENT

Description of the command.

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

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

```
'Examples:

Examples using the command.
```

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

Variables

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

Variables are characterised by two attributes:

- Variable Scope
- Variable Class

DIM

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

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

DIM *var<,var<,…>>*

Arguments:

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

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

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

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

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

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

Interactive Command: NO

```
//Example :: DimEx1.sb (See in Firmware Zip file)

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

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

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

Expected Output:

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

Variable Scope

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

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

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

Variable Class

smart BASIC supports two generic classes of variables:

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

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

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

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

  For example, in the BL600 module, the limit is 512 bytes since it is always the largest data length for any attribute.

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

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

Expected Output:

```
LairdRocks!
```

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

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

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

Arrays

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

DIM array1 [10] AS STRING

DIM array2(10) AS STRING

```
//Example :: ArraysEx1.sb (See in Firmware Zip file)

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

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

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

Expected Output:

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

AT&W
```

General Comments on Variables

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

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

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

Declaring Variables

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

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

Constants

Numeric Constants

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

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

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

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

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

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

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

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

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

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

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

String Constants

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

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

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

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

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

Compiler Related Commands and Directives

#SET

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

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

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

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

#SET commandID, commandValue

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

Currently *smart*BASIC supports the following cmdIDs:

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

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

```
'Example
```

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

Arithmetic Expressions

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

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

Variable types should not be mixed.

```
//Example :: Arithmetic.sb (See in Firmware Zip file)

DIM sum1,bit1,bit2
bit1 = 2
bit2 = 3

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

sum1 = bit1 + bit2
volume = height * area

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

Expected Output:

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

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

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

The following is the order of precedence within operators:

- Unitary operators have the highest precedence

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

+	positive (make positive – multiplies it by +1)

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

*****	Multiply
/	Divide
%	Modulus
+	Addition
-	Subtraction
<<	Arithmetic Shift Left
>>	Arithmetic Shift Right
<	Less Than (results in a 0 or 1 value in the expression)
<=	Less Than Or Equal (results in a 0 or 1 value in the expression)
>	Greater Than (results in a 0 or 1 value in the expression)
>=	Greater Than Or Equal (results in a 0 or 1 value in the expression)
==	Equal To (results in a 0 or 1 value in the expression)
!=	Not Equal To (results in a 0 or 1 value in the expression)
&	Bitwise AND
^	Bitwise XOR (exclusive OR)
|	Bitwise OR
&&	Logical AND (results in a 0 or 1 value in the expression)
^^	Logical XOR (results in a 0 or 1 value in the expression)
||	Logical OR (results in a 0 or 1 value in the expression)

Conditionals

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

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

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

DO / UNTIL

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

DO
statement block
UNTIL *arithmetic expr*

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

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

Expected Output:

```
2345678910
```

DO / UNTIL is a core function.

DO / DOWHILE

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

DO
statement block
DOWHILE *arithmetic expr*

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

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

Interactive Command: NO

```
//Example :: DoWhile.sb (See in Firmware Zip file)
DIM a AS INTEGER  //don't really need to supply AS INTEGER
a=1
DO
  a = a+1
```

```
   PRINT a
DOWHILE a<10      //loop will end when A gets to the value 10

```

Expected Output:

```
2345678910
```

DO / DOWHILE is a core function.

FOR / NEXT

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

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

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

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

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

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

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

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

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

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

Interactive Command: NO

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

print "\n"

FOR a=2 DOWNTO 1
  PRINT "Hello"
NEXT

print "\n"

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

Expected Output:

```
HelloHello
HelloHello
HelloHello
```

FOR / NEXT is a core function.

IF THEN / ELSEIF / ELSE / ENDIF

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

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

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

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

IF *arithexpr_1* THEN
statement block A
ENDIF

IF *arithexpr_1* THEN
statement block A
ELSE

statement block B
ENDIF

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

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

All IF constructions must be terminated with an ENDIF statement.

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

IF i==3 THEN : SLEEP(200)

See the Arithmetic Expressions section for more options.

Interactive Command: NO

```
//Example :: IfThenElse.sb (See in Firmware Zip file)
DIM n
n=1
IF n>0 THEN
  PRINT "Laird Rocks\n"
ENDIF
IF n==0 THEN
  PRINT "n is 0"
ELSEIF n==1 THEN
  PRINT "n is 1"
ELSE
  PRINT "n is not 0 nor 1"
ENDIF
```

Expected Output:

```
Laird Rocks
N is 1
```

IF is a core function.

WHILE / ENDWHILE

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

WHILE *arithexpr*
statement block
ENDWHILE

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

All WHILE commands must be terminated with an ENDWHILE statement.

Interactive Command: NO

```
//Example :: While.sb (See in Firmware Zip file)
DIM n
n=0

//now print "Hello" ten times

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

Expected Output:

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

WHILE is a core function.

SELECT / CASE / CASE ELSE / ENDSELECT

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

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

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

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

SELECT *arithexpr*
unconditional statement block
CASE integerconstA
statement block A
CASE integerconstB

```
  statement block B
CASE integerconstc,integerconstd, integerconste, integerconstf, …
  statement block C
CASE ELSE
  statement block
ENDSELECT
```

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

Interactive Command: NO

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

Expected Output:

```
12
```

SELECT is a core function.

BREAK

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

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

BREAK

Interactive Command: NO

```
//Example :: Break.sb (See in Firmware Zip file)
```

```
DIM n
n=0

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

PRINT "\nFinished\n"
```

Expected Output:

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

BREAK is a core function.

CONTINUE

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

CONTINUE

Interactive Command: YES

```
//Example :: Continue.sb (See in Firmware Zip file)
DIM n
n=0

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

PRINT "\nFinished\n"
```

Expected Output:

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

CONTINUE is a core function.

Error Handling

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

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

ONERROR

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

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

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

Arguments:

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

Interactive Command: NO

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

Expected Output:

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

c now equals 4
```

ONERROR is a core function.

ONFATALERROR

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

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

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

ONFATALERROR is a core function.Event Handling

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

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

WAITEVENT

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

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

WAITEVENT

Interactive Command: NO

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

FUNCTION Func1()
  PRINT "\nEV1"
ENDFUNC 0

ONEVENT EV0 CALL Func0
ONEVENT EV1 CALL Func1

WAITEVENT               //wait  for an event  to occur

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

WAITEVENT is a core function.

ONEVENT

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

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

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

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

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

Arguments:

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

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

Some Symbolic Event Names:

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

EVTMRn	Timer n has expired (see Timer Events)
EVUARTRX	Data has arrived in UART interface

EVUARTTXEMPTY	The UART TX ring buffer is empty

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

Interactive Command: NO

Note: This example was written for the BL600 module so the signal numbers used in the GpioBindEvent() statements may be different depending on your module.

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

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

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

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

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

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

onevent evgpiochan0 call Btn0rel     //detects when button 0 is released and calls
the function
onevent evgpiochan1 call Btn0press   //detects when button 0 is pressed and calls the
function
onevent evgpiochan2 call Btn1rel     //detects when button 1 is released and calls
the function
onevent evgpiochan3 call Btn1press   //detects when button 1 is pressed and calls the
function

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

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

Expected Output:

```
Waiting for an event...

Button 0 has been pressed
Button 0 released. Resume waiting for an event

Button 1 has been pressed
Button 1 released. No more waiting for events

Got here because evgpiochan3 happened
```

ONEVENT is a core function.

Miscellaneous Commands

PRINT

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

PRINT *exprlist*

Arguments:

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

Formatting with PRINT – Expression Lists

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

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

Numeric Expression Lists

Numeric variables are formatted in the following form:

<type.base> arithexpr <separator>

Where,

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

  , Insert a tab before the next variable.

; Print the next variable without a space.

String Expression Lists

String variables are formatted in the following form:

<type . minchar> strexpr< separator>

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

  , Insert a tab before the next variable.

  ; Print the next variable without a space.

Interactive Command: YES

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

Expected Output:

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

PRINT is a core function.

SPRINT

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

It is very useful for creating strings with formatted data.

SPRINT #stringvar, *exprlist*

Arguments:

Stringvar – A pre-declared string variable.

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

Formatting with SPRINT – Expression Lists

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

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

Numeric Expression Lists

Numeric variables are formatted in the following form:

<type.base> arithexpr <separator>

Where,

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

  , Insert a tab before the next variable.

  ; Print the next variable without a space.

String Expression Lists

String variables are formatted in the following form:

<type . minchar> strexpr< separator>

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

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

Interactive Command: YES

```
//Example :: SPrint.sb (See in Firmware Zip file)
DIM a,s$ : a=100
//Note: SPRINT replaces the content of s$ with exprlist each time it is used
SPRINT #s$,a                //s$ now contains 100
PRINT "\n";s$;"\n"
```

```
SPRINT #s$,INTEGER.H'a      //s$ now contains 64
PRINT s$;"\n"
SPRINT #s$,INTEGER.O'a      //s$ now contains 144
PRINT s$;"\n"
SPRINT #s$,INTEGER.B'a      //s$ now contains 1100100
PRINT s$;"\n"
```

Expected Output:

```
100
00000064
00000000144
00000000000000000000000001100100
```

SPRINT is a core function.

STOP

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

STOP

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

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

Interactive Command: NO

```
//Example :: Stop.sb (See in Firmware Zip file)
DIM a, s$
a=100

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

Expected Output:

```
100

01     0702
resume
00000064
00000000144
00000000000000000000000001100100
```

STOP is a core function.

BP

COMMAND

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

BP nnnn

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

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

Arguments

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

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

Interactive Command: NO

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

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

```
hello
21     BREAKPOINT    1234
resume
worldLairdRocks
21     BREAKPOINT    5678
so
the
21     BREAKPOINT    5678
so
world
21     BREAKPOINT    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 extention manual for the target platform.

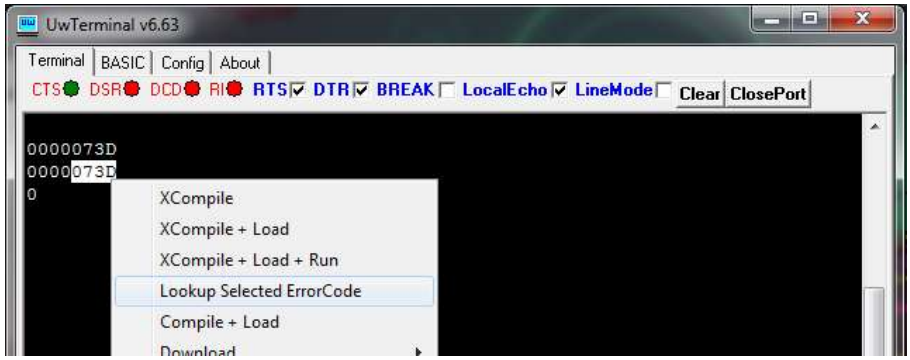
Result Codes

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

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

```
//Example :: ResultCodes.sb (See in Firmware Zip file)
DIM cB,nItems,rc,s$

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

PRINT "\n";             //New line

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

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

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



Expected Output:

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

Information Routines

GETLASTERROR

FUNCTION

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

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

GETLASTERROR ()

Returns INTEGER Last error that was generated.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments None

Interactive Command: NO

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

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

```
error  = 0x00000000
```

GETLASTERROR is a core function.

RESETLASTERROR

SUBROUTINE

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

RESETLASTERROR ()

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments None

Interactive Command: NO

```
//Example :: ResetLastError.sb (See in Firmware Zip file)
DIM err : err = GETLASTERROR()
```

```
RESETLASTERROR()
PRINT "\nerror  = 0x" ; INTEGER.H'err
```

Expected Result:

```
error  = 0x00000000
```

RESETLASTERROR is a core function.

SYSINFO

FUNCTION

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

SYSINFO(varId)

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

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

varId *byVal* varId *AS INTEGER*

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

0	ID of device, for the BL600 module the value will be 0x42460600. Each platform type has a unique identifier
3	Version number of Module Firmware. For example W.X.Y.Z will be returned as a 32 bit value made up as follows: (W<<26) + (X<<20) + (Y<<6) + (Z) where Y is the Build number and Z is the 'Sub-Build' number
33	BASIC core version number
601	Flash File System: Data Segment: Total Space
602	Flash File System: Data Segment: Free Space
603	Flash File System: Data Segment: Deleted Space
611	Flash File System: FAT Segment: Total Space
612	Flash File System: FAT Segment: Free Space
613	Flash File System: FAT Segment: Deleted Space
631	NvRecord Memory Store Segment: Total Space
632	NvRecord Memory Store Segment: Free Space
633	NvRecord Memory Store Segment: Deleted Space
1000	BASIC compiler HASH value as a 32 bit decimal value
1001	How RAND() generates values: 0 for PRNG and 1 for hardware assist
1002	Minimum baudrate
1003	Maximum baudrate
1004	Maximum STRING size

1005	Will be 1 for run-time only implementation, 3 for compiler included
2000	Reset Reason 8 : Self-Reset due to Flash Erase 9 : ATZ 10 : Self-Reset due to *smart* BASIC app invoking function RESET()
2002	Timer resolution in microseconds
2003	Number of timers available in a *smart* BASIC Application
2004	Tick timer resolution in microseconds

Interactive Command: No

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

Expected Output (For BL600):

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

SYSINFO is a core language function.

SYSINFO$

FUNCTION

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

SYSINFO$(varId)

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

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

varId *byVal* varId *AS INTEGER*

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

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

bytes are the address.

Interactive Command: No

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

Expected Output:

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

SYSINFO$ is a core language function.

Event & Messaging Routines

SENDMSGAPP

FUNCTION

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

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

ONEVENT EVMSGAPP CALL HandlerMsgApp

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

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

SENDMSGAPP(msgId, msgCtx)

Returns INTEGER 0000 if successfully sent.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

msgId ***byVal* msgId *AS INTEGER***

Will be presented to the EVMSGAPP handler in the msgId field

msgCtx ***byVal* msgCtx *AS INTEGER***

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

Interactive Command: NO

```
//Example :: SendMsgApp.sb (See in Firmware Zip file)
DIM rc

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

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

Expected Output:

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

SENDMSGAPP is a core function.

Arithmetic Routines

ABS

FUNCTION

Returns the absolute value of its INTEGER argument.

ABS (var)

Returns INTEGER Absolute value of ***var***.

Exceptions

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

Arguments:

var ***byVal* var *AS INTEGER***

The variable whose absolute value is required.

Interactive Command: No

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

Expected Output:

```
-2      2
4       4
```

ABS is a core language function.

MAX

FUNCTION

Returns the maximum of two integer values.

MAX (var1, var2)

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

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

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

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

Interactive Command: No

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

Expected Output:

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

MAX is a core language function.

MIN

FUNCTION

Returns the minimum of two integer values.

MIN (var1, var2)

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

Exceptions
- Local Stack Frame Underflow

- Local Stack Frame Overflow

Arguments:

var1	***byVal* var1 *AS INTEGER*** The first of two variables to be compared.
var2	***byVal* var2 *AS INTEGER*** The second of two variables to be compared.

Interactive Command: No

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

Expected Output:

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

MIN is a core language function.

String Routines

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

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

LEFT$

Retrieves the leftmost n characters of a string.

LEFT$(string,length)

Function

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

Exceptions

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

Arguments:

string	***byRef string AS STRING*** The target string which cannot be a const string.
length	***byVal length AS INTEGER*** The number of leftmost characters that are returned.

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

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

Interactive Command: NO

```
//Example :: LEFT$.sb (See in Firmware Zip file)
DIM newstring$
DIM s$

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

Expected Output:

```
Arse
```

LEFT$ is a core language function.

MID$

FUNCTION

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

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

MID$(string, pos, length)

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

Exceptions

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

Arguments:

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

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

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

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

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

Interactive Command: NO

```
//Example :: MID.sb (See in Firmware Zip file)
DIM s$ : s$="Arsenic"
DIM new$ : new$ = MID$(s$,2,4)
PRINT new$; "\n"
```

Expected Output:

```
Arse
abcdef
cdefg
hij
```

MID$ is a core language function.

RIGHT$

FUNCTION

Retrieves the rightmost n characters from a string.

RIGHT$(string, len)

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

Exceptions

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

Arguments:

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

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

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

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

Interactive Command: NO

```
//Example :: RIGHT$.sb (See in Firmware Zip file)
```

```
DIM s$ : s$="Parse"
DIM new$ : new$ = RIGHT$(s$,4)
PRINT new$; "\n"
```

Expected Output:

```
arse
```

RIGHT$ is a core function.

STRLEN

FUNCTION

STRLEN returns the number of characters within a string.

STRLEN (string)

Returns INTEGER The number of characters within the string.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

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

Interactive Command: NO

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

Expected Output:

```
HelloWorld is 10 bytes long
```

STRLEN is a core function.

STRPOS

FUNCTION

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

STRPOS (string1, string2, startpos)

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

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

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

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

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

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

Note: STRPOS does a case sensitive search.

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

Interactive Command: NO

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

Expected Output:

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

STRPOS is a core function.

STRSETCHR

FUNCTION

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

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

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

STRSETCHR (string, nChr, nIndex)

Returns INTEGER Represents command execution status.

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

Exceptions

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

Arguments:

string ***byRef string AS STRING***
The target string.

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

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

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

Interactive Command: NO

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

Expected Output:

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

STRSETCHR is a core function.

STRGETCHR

FUNCTION

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

STRGETCHR (string, nIndex)

Returns INTEGER The ASCII value of the character at position ***nIndex*** within ***string***, where ***nIndex*** is zero based. If ***nIndex*** is greater than the number of characters in the

string or <=0 then an error value of -1 is returned.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

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

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

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

Interactive Command: NO

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

Expected Output:

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

STRGETCHR is a core function.

STRSETBLOCK

FUNCTION

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

STRSETBLOCK (string, nChr, nIndex, nBlocklen)

Function

Returns INTEGER Represents command execution status.

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

-4 If ***nChr*** is greater than 255 or less than 0
-5 if the nBlockLen value is negative

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string	***byRef string AS STRING*** The target string to be modified
nChr	***byVal nChr AS INTEGER*** The character that will overwrite the existing characters. ***nChr*** must be within the range 0 – 255
nindex	***byVal nIndex AS INTEGER*** The starting point for the filling block, referenced to a zero index.
nBlocklen	***byVal nBlocklen AS INTEGER*** The number of characters to be overwritten

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

Interactive Command: NO

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

Expected Output:

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

STRSETBLOCK is a core function.

STRFILL

FUNCTION

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

STRFILL (string, nChr, nCount)

Returns INTEGER Represents command execution status.

0	If successful
-1	If ***nChr*** is greater than 255 or less than 0

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

STRING

string contains the modified string

Exceptions

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

Arguments:

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

nChr ***byVal nChr AS INTEGER***
ASCII value of the character to be inserted. The value of nChr should be between 0 and 255 inclusive.

nCount ***byVal nCount AS INTEGER***
The number of occurrences of ***nChr*** to be added.

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

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

Interactive Command: NO

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

Expected Output:

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

STRFILL is a core function.

STRSHIFTLEFT

SUBROUTINE

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

STRSHIFTLEFT (string, numChars)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

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

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

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

Interactive Command: NO

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

Expected Output:

```
123456789
56789
```

STRSHIFTLEFT is a core function.

STRCMP

FUNCTION

Compares two string variables.

STRCMP(string1, string2)

Returns INTEGER A value indicating the comparison result:
0 – if ***string1*** exactly matches ***string2*** (the comparison is case sensitive)
1 – if the ASCII value of ***string1*** is greater than ***string2***
-1 - if the ASCII value of ***string1*** is less than ***string2***

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

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

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

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

Interactive Command: NO

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

Expected Output:

```
-1
1
0
```

STRCMP is a core function.

STRHEXIZE$

FUNCTION

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

STRHEXIZE$ (string)

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

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

Arguments:

String ***byRef string AS STRING***

The string to be converted into hex characters.

Interactive Command: NO

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

Associated Commands: STRHEX2BIN

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

Expected Output:

```
Laird
5
4C61697264
10
```

STRHEXIZE$ is a core function.

STRDEHEXIZE$

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

STRDEHEXIZE$ (string)

Function

Returns STRING A dehexed version of **string**

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string ***byRef string AS STRING***

The string to be converted in-situ.

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

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

Interactive Command: NO

```
//Example :: StrDehexize$.sb (See in Firmware Zip file)

DIM s$ : s$="40414243"

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

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

Expected Output:

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

STRDEHEXIZE$ is a core function.

STRHEX2BIN

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

STRHEX2BIN (string,offset)

Function

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

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

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

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

Interactive Command: NO

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

Associated Commands: STRHEXIZE

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

Expected Output:

```
3
97
```

STRHEX2BIN is a core function.

STRESCAPE$

FUNCTION

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

STRESCAPE$ (string)

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

The following input characters are escaped as follows:

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

Exceptions

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

Arguments:

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

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

Interactive Command: NO

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

Associated Commands: STRDEESCAPE

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

Expected Output:

```
11
13
```

STRESCAPE$ is a core function.

STRDEESCAPE

SUBROUTINE

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

STRDEESCAPE (string)

Returns None

string now contains de-escaped characters converted as follows:

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

Exceptions

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

Arguments:

string ***byRef string AS STRING***

The string to be converted in-situ.

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

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

Interactive Command: NO

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

Expected Output:

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

STRDEESCAPE is a core function.

STRVALDEC

FUNCTION

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

STRVALDEC (string)

Function

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

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string ***byRef string AS STRING***
The target string

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

Any leading whitespace within the string is ignored.

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

Interactive Command: NO

```
//Example :: StrValDec.sb (See in Firmware Zip file)
DIM s$
s$="  1234"
PRINT "\n";StrValDec(s$)
s$=" -1234"
PRINT "\n";StrValDec(s$)
s$=" +1234"
PRINT "\n";StrValDec(s$)
s$="  2345hello"
PRINT "\n";StrValDec(s$)
s$="  hello"
PRINT "\n";StrValDec(s$)
```

Expected Output:

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

STRVALDEC is a core function.

STRSPLITLEFT$

FUNCTION

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

STRSPLITLEFT$ (string, length)

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

Exceptions

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

Arguments:

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

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

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

Interactive Command: NO

```
//Example :: StrSplitLeft$.sb (See in Firmware Zip file)
DIM origStr$
origStr$ = "12345678"
PRINT StrSplitLeft$ (origStr$, 3);"\n"
PRINT origStr$
```

Expected Output:

```
123
45678
```

STRSPLITLEFT$ is a core function.

STRSUM

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

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

STRSUM (string, nIndex, nBytes, initVal)

Function

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

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string ***byRef string AS STRING***

String that contains the unsigned bytes which need to be arithmetically added

nIndex ***byVal nIndex AS INTEGER***

Index of first byte into the string

nBytes ***ByVal nBytes AS INTEGER***

Number of bytes to process

initVal ***ByVal initVal AS INTEGER***

Initial value of the sum

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

Interactive Command: NO

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

Expected Output:

```
307
317
160
```

STRSUM is a core function.

STRXOR

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

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

STRXOR (string, nIndex, nBytes, initVal)

Function

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

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string	***byRef string AS STRING*** String that contains the unsigned bytes which need to be XOR'd
nIndex	***byVal nIndex AS INTEGER*** Index of first byte into the string
nBytes	***ByVal nBytes AS INTEGER*** Number of bytes to process
initVal	***ByVal initVal AS INTEGER*** Initial value of the XOR

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

Interactive Command: NO

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

Expected Output:

```
52
62
988
```

STRXOR is a core function.

EXTRACTSTRTOKEN

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

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

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

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

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

EXTRACTSTRTOKEN (sentence$,token$)

Function

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

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

sentence$	***byRef sentence$ AS STRING*** String that contains the sentence containing the tokens to be extracted
token$	***byRef token$ AS STRING*** The leftmost token from the sentence and will have been removed from the sentence.

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

Interactive Command: NO

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

Expected Output:

```
Sentence is :My name is BL600, from     Laird
Token (len 2) = :My
Token (len 4) = :name
Token (len 2) = :is
Token (len 5) = :BL600
Token (len 1) = :,
Token (len 4) = :from
Token (len 5) = :Laird
Token (len 0) = :
```

ExtractStrToken is a core function.

EXTRACTINTTOKEN

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

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

result in "**hex**" then "," and then "**value**"

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

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

EXTRACTINTTOKEN (sentence$,intValue)

Function

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

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

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

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

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

Interactive Command: NO

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

Expected Output:

```
Sentence is :0x100 is a hex,value
intValue (bytes 5) = :256
Token (len 2) = :is
Token (len 1) = :a
Token (len 3) = :hex
Token (len 1) = :,
Token (len 5) = :value
Token (len 0) = :
```

ExtractIntToken is a core function.

Table Routines

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

Tables are one dimensional string variables, which are configured by using the TABLEINIT command.

Tables should not be confused with Arrays. Tables provide the ability to perform pattern matching in a highly optimised manner. As a general rule, use tables where you want to perform efficient pattern matching and arrays where you want to automate setup strings or send data using looping variables.

TABLEINIT

FUNCTION

TABLEINIT initialises a string variable so that it can be used for storage of multiple TLV tokens, allowing a lookup table to be created.

TLV = Tag, Length, Value

TABLEINIT (string)

Returns INTEGER Indicates success of command:

0 Successful initialisation
<>0 Failure

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string ***byRef string AS STRING***

String variable to be used for the Table. Since it is byRef the compiler will not allow a constant string to be passed as an argument. On entry the string can be non-empty, on exit the string will be empty.

Interactive Command: NO

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

Associated Commands: **TABLEADD, TABLELOOKUP**

```
//Example :: TableInit.sb (See in Firmware Zip file)
DIM t$ :t$="Hello"
PRINT "\n";"[";t$;"]"
PRINT "\n";TableInit(t$)
PRINT "\n";"[";t$;"]"     //String now blank after being initialised as a table
```

Expected Output:

```
[Hello]
0
[]
```

TABLEINIT is a core function.

TABLEADD

FUNCTION

TABLEADD adds the token specified to the lookup table in the string variable and associates the index specified with it. There is no validation to check if nIndex has been duplicated as it is entirely valid that more than one token generate the same ID value

TABLEADD (string, strtok, nID)

Returns INTEGER Indicates success of command:

- 0 Signifies that the token was successfully added
- 1 Indicates an error if ***nID*** > 255 or < 0
- 2 Indicates no memory is available to store token
- 3 Indicates that the token is too large
- 4 Indicates the token is empty

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string ***byRef string AS STRING***
A string variable that has been initialised as a table using TABLEINIT.

strtok ***byVal strtok AS STRING***
The string token to be added to the table.

nID ***byVal nID AS INTEGER***
The identifier number that is associated with the token and should

be in the range 0 to 255.

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

Interactive Command: NO

Associated Commands: **TABLEINIT, TABLELOOKUP**

```
//Example :: TableAdd.sb (See in Firmware Zip file)
DIM t$ : PRINT TableInit(t$);"\n"
PRINT TableAdd(t$,"Hello",1);"\n"
PRINT TableAdd(t$,"everyone",2);"\n"
PRINT TableAdd(t$,"to",300);"\n"
PRINT TableAdd(t$,"",3);"\n"
PRINT t$
//Tokens are stored in TLV format: \Tag\LengthValue
```

Expected Output:

```
0
0
0
1
4
\01\05Hello\02\08everyone
```

TABLEADD is a core function.

TABLELOOKUP

FUNCTION

TABLELOOKUP searches for the specified token within an existing lookup table which was created using TABLEINIT and multiple TABLEADDs and returns the ID value associated with it.

It is especially useful for creating a parser, for example, to create an AT style protocol over a uart interface.

TABLELOOKUP (string, strtok)

Returns INTEGER Indicates success of command:

- >=0 signifies that the token was successfully found and the value is the ID
- -1 if the token is not found within the table
- -2 if the specified table is invalid
- -3 if the token is empty or > 255 characters

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string	***byRef string AS STRING*** The lookup table that is being searched
strtok	***byRef strtok AS STRING*** The token whose position is being found

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

Interactive Command: NO

Associated Commands: **TABLEINIT, TABLEADD**

```
//Example :: TableLookup.sb (See in Firmware Zip file)
DIM t$
PRINT TableInit(t$);"\n\n"

PRINT TableAdd(t$,"Hello",1);"\n"
PRINT TableAdd(t$,"world",2);"\n"
PRINT TableAdd(t$,"to",3);"\n"
PRINT TableAdd(t$,"you",4);"\n\n"

PRINT TableLookup(t$,"to");"\n"
PRINT TableLookup(t$,"Hello");"\n"
PRINT TableLookup(t$,"you");"\n"
```

Expected Output:

```
0

0
0
0
0

3
1
4
```

TABLELOOKUP is a core function.

Miscellaneous Routines

This section describes all miscellaneous functions and subroutines

RESET

SUBROUTINE

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

RESET (nType)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nType	**byVal *nType* AS INTEGER.** This is for future use. Set to 0.

Interactive Command: NO

```
//Example :: RESET.sb (See in Firmware Zip file)
RESET(0)    //force a reset of the module
```

Expected Output:

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

RESET is a core subroutine.

Random Number Generation Routines

Random numbers are either generated using pseudo random number generator algorithms or using thermal noise or equivalent in hardware. The routines listed in this section provide the developer with the capability of generating random numbers.

The Interactive Mode command "AT I 1001" or at runtime SYSINFO(1001) will return 1 if the system generates random numbers using hardware noise or 0 if a pseudo random number generator.

RAND

FUNCTION

The RAND function returns a random 32 bit integer. Use the command 'AT I 1001' or from within an application the function SYSINFO(1001), to determine whether the random number is pseudo random or generated in hardware via a thermal noise generator. If 1001 returns 0 then it is pseudo random and 1 if generated using hardware.

RAND ()

Returns INTEGER A 32 bit integer.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments: None

Depending on the platform, the RAND function can be seeded using the RANDSEED function to seed the pseudo random number generator. If used, RANDSEED must be called before using RAND. If the platform has a hardware Random Number Generator, then RANDSEED has no effect.

Interactive Command: NO

Associated Commands: RANDSEED

```
//Example :: RAND.sb (See in Firmware Zip file)
PRINT "\nRandom number is ";RAND()
```

Expected Output:

```
Random number is -2088208507
```

RAND is a core language function.

RANDEX

FUNCTION

The RANDEX function returns a random 32 bit **positive** integer in the range 0 to X where X is the input argument. Use the command 'AT I 1001' or from within an application the function SYSINFO(1001) to determine whether the random number is pseudo random or generated in hardware via a thermal noise generator. If 1001 returns 0 then it is pseudo random and 1 if generated using hardware.

RANDEX (maxval)

Returns INTEGER A 32 bit integer.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

maxval ***byVal maxval AS INTEGER***

The return value will not exceed the absolute value of this variable

Depending on the platform, the RANDEX function can be seeded using the RANDSEED function to seed the pseudo random number generator. If used, RANDSEED must be called before using RANDEX. If the platform has a hardware Random Number Generator, then RANDSEED has no effect.

Interactive Command: NO

Associated Commands: RANDSEED

```
//Example :: RANDEX.sb (See in Firmware Zip file)
DIM x : x=500
PRINT "\nRandom number between 0 and ";x;" is ";RANDEX(x)
```

Expected Output:

```
Random number between 0 and 500 is 193
```

RAND is a core language function.

RANDSEED

SUBROUTINE

On platforms without a hardware random number generator, the RANDSEED function sets the starting point for generating a series of pseudo random integers. To reinitialize the generator, use 1 as the seed argument. Any other value for seed sets the generator to a random starting point. RAND retrieves the pseudo random numbers that are generated.

It has no effect on platforms with a hardware random number generator.

RANDSEED (seed)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

Seed ***byVal seed AS INTEGER***

The starting seed value for the random number generator function RAND.

Interactive Command: No

Associated Commands: RAND

```
RandSeed(1234)
```

Note: This subroutine has no effect on modules that have a hardware random number generator..

RANDSEED is a core language subroutine.

Timer Routines

In keeping with the event driven paradigm of *smart* BASIC, the timer subsystem enables *smart* BASIC applications to be written which allow future events to be generated based on timeouts. To make use of this feature up to N timers, where N is platform dependent, are made available and that many event handlers can be written and then enabled using the ONEVENT statement so that those handlers are automatically invoked. The ONEVENT statement is described in detail elsewhere in this manual.

Briefly the usage is, select a timer, register a handler for it, and start it with a timeout value and a flag to specify whether it is recurring or single shot. Then when the timeout occurs AND when the application is processing a WAITEVENT statement, the handler will be automatically called.

It is important to understand the significance of the WAITEVENT statement. In a nutshell, a timer handler callback will NOT happen if the runtime engine does not encounter a WAITEVENT statement. Events are synchronous not asynchronous like say interrupts.

All this is illustrated in the sample code fragment below where timer 0 is started so that it will recur automatically every 500 milliseconds and timer 1 is a single shot 1000ms later.

Note, as explained in the WAITEVENT section of this manual, if a handler function returns a non-zero value then the WAITEVENT statement is reprocessed, otherwise the *smart* BASIC runtime engine will proceed to process the next statement **after** the WAITEVENT statement – not after the handlers ENDFUNC or EXITFUNC statement. This means that if the WAITEVENT is the very last statement in an application and a timer handler returns a 0 value, then the application will exit the module from Run Mode into Interactive Mode which will be disastrous for unattended operation.

Timer Events

EVTMRn Where n=0 to N, where N is platform dependent, it is generated when timer n expires. The number of timers (that is, N+1) is returned by the command AT I 2003 or at runtime by SYSINFO(2003)

```
//Example :: EVTMRn.sb (See in Firmware Zip file)
FUNCTION HandlerTimer0()
  PRINT "\nTimer 0 has expired"
ENDFUNC 1                           //remain blocked in WAITEVENT

FUNCTION HandlerTimer1()
  PRINT "\nTimer 1 has expired"
ENDFUNC 0                           //exit from WAITEVENT

ONEVENT EVTMR0 CALL HandlerTimer0
ONEVENT EVTMR1 CALL HandlerTimer1

TimerStart(0,500,1)                 //start a 500 millisecond recurring timer
PRINT "\nWaiting for Timer 0"
TimerStart(1,1000,0)                //start a 1000 millisecond timer
PRINT "\nWaiting for Timer 1"

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

Expected Output:

```
Waiting for Timer 0
Waiting for Timer 1
Timer 0 has expired
Timer 0 has expired
Timer 1 has expired
Got 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 1,000,000,000 (11.6 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 > 1,000,000,000 then a runtime error will be thrown with code INVALID_INTERVAL. An error will be thrown for lesser values dependent on the platform and the hardware constraints. For example, the BL600 module has a maximum time of 8192000 (2 hrs 16 min).

If the ***recurring*** argument is set to non-zero, then the minimum value of the interval is 10ms

recurring ***byVal recurring AS INTEGER***
Set to 0 for a once-only timer, or non-0 for a recurring timer.

When the timer expires, it will set the corresponding EVTMRn event. That is, timer number 0 sets EVTMR0, timer number 3 sets EVTMR3. The ONEVENT statement should be used to register handlers that will capture and process these events.

If the timer is already running, calling TIMERSTART will reset it to count down from the new value, which may be greater or smaller than the remaining time.

If either ***number*** or ***interval*** is invalid an Error is thrown.

Interactive Command: No

Related Commands: ONEVENT, TIMERCANCEL

```
//Example :: EVTMRn.sb (See in Firmware Zip file)
SUB HandlerOnErr()
  PRINT "Timer Error: ";GetLastError()
ENDSUB

FUNCTION HandlerTimer1()
  PRINT "\nTimer 1 has expired"
ENDFUNC 1                              //remain blocked in WAITEVENT

FUNCTION HandlerTimer2()
  PRINT "\nTimer 2 has expired"
ENDFUNC 0                              //exit from WAITEVENT

ONERROR NEXT HandlerOnErr

ONEVENT EVTMR1 CALL HandlerTimer1
ONEVENT EVTMR2 CALL HandlerTimer2

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

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

TimerStart(2,1000,0)                   //start a 1000 millisecond timer
PRINT "\nWaiting for Timer 2"

WAITEVENT
PRINT "\nGot here because TIMER 2 expired and Handler returned 0"
```

Expected Output:

```
Timer Error: 1770
Started Timer 0 with invalid inerval
Waiting for Timer 1
Waiting for Timer 2
Timer 1 has expired
Timer 1 has expired
Timer 2 has expired
Got here because TIMER 2 expired and Handler returned 0
```

TIMERSTART is a core subroutine.

TimerRunning

FUNCTION

This function determines if a timer identified by an index number is still running. The command AT I 2003 will return the valid range of Timer index numbers. It returns 0 to signify that the timer is not running and a non-zero value to signify it is still running and the value is the number of milliseconds left for it to expire.

TIMERRUNNING (number)

Function

Returns: 0 if the timer has expired, otherwise the time in milliseconds left to expire.

Arguments:

number ***byVal number AS INTEGER***
The number of the timer. 0 to N where N can be determined by submitting the command AT I 2003 or at runtime returned via SYSINFO(2003).

If the value is not valid, then a runtime error will be thrown with code INVALID_TIMER.

Interactive Command: No

Related Commands: ONEVENT, TIMERCANCEL

```
//Example :: TimerRunning.sb (See in Firmware Zip file)
SUB HandlerOnErr()
  PRINT "Timer Error ";GetLastError()
ENDSUB

FUNCTION HandlerTimer0()
  PRINT "\nTimer 0 has expired"
  PRINT "\nTimer 1 has ";TimerRunning(1);" milliseconds to go"
ENDFUNC 1                              //remain blocked in WAITEVENT

FUNCTION HandlerTimer1()
  PRINT "\nTimer 1 has expired"
ENDFUNC 0                              //exit from WAITEVENT

ONERROR NEXT HandlerOnErr

ONEVENT EVTMR0 CALL HandlerTimer0
ONEVENT EVTMR1 CALL HandlerTimer1

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

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

WAITEVENT
```

Expected Output:

```
Waiting for Timer 0
Waiting for Timer 1
Timer 0 has expired
Timer 1 has 1500 milliseconds to go
Timer 0 has expired
Timer 1 has 1000 milliseconds to go
Timer 0 has expired
Timer 1 has 500 milliseconds to go
Timer 0 has expired
Timer 1 has 0 milliseconds to go
Timer 1 has expired
```

TIMERRUNNING is a core function

TimerCancel

SUBROUTINE

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

TIMERCANCEL (number)

Arguments:

number ***byVal number AS INTEGER***
The number of the timer. 0 to N where N can be determined by submitting the command AT I 2003 or at runtime returned via SYSINFO(2003).

If the value is not valid, then a runtime error will be thrown with code INVALID_TIMER.

Interactive Command: NO

Related Commands: ONEVENT, TIMERCANCEL,TIMERRUNNING

```
//Example :: TimerCancel.sb (See in Firmware Zip file)
DIM i,x
i=0 : x=1   //'x' is HandlerTimer0's return value
          //Will switch to 0 when timer0 has expired so that the application can
stop
FUNCTION HandlerTimer0()
    PRINT "\nTimer 0 has expired, starting again"
    IF i==4 THEN
        PRINT "\nCancelling Timer 0"
        TimerCancel(0)
        PRINT "\nTimer 0 ran ";i+1;" times"
        x=0
    ENDIF
    i=i+1
ENDFUNC x
ONEVENT EVTMR0 CALL HandlerTimer0
TimerStart(0,800,1)
PRINT "\nWaiting for Timer 0. Should run 5 times"
WAITEVENT
```

Expected Output:

```
Waiting for Timer 0. Should run 5 times
Timer 0 has expired, starting again
Timer 0 has expired, starting again
Timer 0 has expired, starting again
Timer 0 has expired, starting again
Timer 0 has expired, starting again
Cancelling Timer 0
Timer 0 ran 5 times
```

TIMERCANCEL is a core subroutine.

GetTickCount

FUNCTION

There is a 31 bit free running counter that increments every 1 millisecond. The resolution of this counter in microseconds can be determined by submitting the command AT I 2004 or at runtime SYSINFO(2004) . This function returns that free running counter. It wraps to 0 when the counter reaches 0x7FFFFFFF.

GETTICKCOUNT ()

Returns: INTEGER A value in the range 0 to 0x7FFFFFFF (2,147,483,647) in units of milliseconds.

Arguments: **None**

Interactive Command: No

Related Commands: GETTICKSINCE

```
//Example :: GetTickCount.sb (See in Firmware Zip file)
FUNCTION HandlerTimer0()
  PRINT "\n\nTimer 0 has expired"
ENDFUNC 0

PRINT "\nThe value on the counter is ";GetTickCount()

ONEVENT EVTMR0 CALL HandlerTimer0

TimerStart(0,1000,0)
PRINT "\nWaiting for Timer 0"

WAITEVENT
PRINT "\nThe value on the counter is now ";GetTickCount();
```

Expected Output:

```
The value on the counter is 159297
Waiting for Timer 0

Timer 0 has expired
The value on the counter is now 160299
```

GETTICKCOUNT is a core subroutine.

GetTickSince

FUNCTION

This function returns the time elapsed since the 'startTick' variable was updated with the return value of GETTICKCOUNT(). It signifies the time in milliseconds. If 'startTick' is less than 0 which is a value that GETTICKCOUNT() will never return, then a 0 will be returned.

GETTICKSINCE (startTick)

Returns: INTEGER A value in the range 0 to 0x7FFFFFFF (2,147,483,647) in units of milliseconds.

startTickr ***byVal startTick AS INTEGER***
This is a variable that was updated using the return value from GETTICKCOUNT() and it is used to calculate the time elapsed since that update.

Interactive Command: No

Related Commands: GETTICKCOUNT

```
//Example :: GetTickSince.sb (See in Firmware Zip file)
DIM startTick, elapseMs, x
x=1
startTick = GetTickCount()

DO
    PRINT x;" x 2 = "
    x=x*2
    PRINT x;"\n"
UNTIL x==32768

elapseMs = GetTickSince(startTick)
PRINT "\n\nThe Do Until loop took ";elapseMS; " msec to process"
```

Expected Output:

```
1 x 2 = 2
2 x 2 = 4
4 x 2 = 8
8 x 2 = 16
16 x 2 = 32
32 x 2 = 64
64 x 2 = 128
128 x 2 = 256
256 x 2 = 512
512 x 2 = 1024
1024 x 2 = 2048
2048 x 2 = 4096
4096 x 2 = 8192
8192 x 2 = 16384
16384 x 2 = 32768

The Do Until loop took 21 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 inherited in any interpreted language. The basic entity that is managed is the INTEGER variable in smartBASIC. Hence be aware that for a buffer size of N, 4 times N is the memory that will be taken from the internal heap.

These buffers are referenced using handles provided at creation time.

CircBufCreate

FUNCTION

This function is used to create a circular buffer with a maximum capacity set by the caller. Most often it will be used as a first-in, first-out queue.

CIRCBUFCREATE (nItems, circHandle)

Returns: INTEGER

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

Arguments:

nItems **byVal *nItems* AS INTEGER**
This specifies the maximum number of INTEGER values that can be stored in the buffer. If there isn't enough free memory in the heap, then this function will fail and return an appropriate result code.

circHandle **byRef circHandle AS INTEGER**
If the circular buffer is successfully created, then this variable will return a handle that should be used to interact with it.

Interactive Command: NO

```
//Example :: CircBufCreate.sb (See in Firmware Zip file)
DIM circHandle, circHandle2, rc

rc = CircBufCreate(16,circHandle)
PRINT "\n";rc
IF rc!=0 THEN
  PRINT "\nThe circular buffer ";circHandle; "was not created"
ENDIF

rc = CircBufCreate(32000,circHandle2)
PRINT "\n\n";rc
IF rc!=0 THEN
  PRINT "\n---> The circular buffer 'circHandle2' was not created"
ENDIF
```

Expected Output:

```
0

20736
---> The circular buffer 'circHandle2' was not created
```

CIRCBUFCREATE is an extension function.

CircBufDestroy

SUBROUTINE

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

CIRCBUFDESTROY (circHandle)

Arguments:

circHandle **byRef circHandle AS INTEGER**
A handle referencing the circular buffer that needs to be deleted. On exit an invalid handle value will be returned

Interactive Command: NO

```
//Example :: CircBufDestroy.sb (See in Firmware Zip file)
DIM circHandle, circHandle2, rc

rc = CircBufCreate(16,circHandle)
PRINT "\n";rc
IF rc!=0 THEN
    PRINT "\nThe circular buffer ";circHandle; " was not created"
ENDIF

CircBufDestroy(circHandle)
PRINT "\nThe handle value is now ";circHandle; " so it has been destroyed"
```

Expected Output:

```
0
The handle value is now -1 so it has been destroyed
```

CIRCBUFDESTROY is an extension function.

CircBufWrite

FUNCTION

This function is used to write an integer at the head end of the circular buffer and if there is no space available to write, then it will return with a failure resultcode and NOT write the value.

CIRCBUFWRITE (circHandle, nData)

Returns: INTEGER

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

Arguments:

circHandle **byRef circHandle AS INTEGER**
This identifies the circular buffer to write into.

nData **byVal *nData* AS INTEGER**
This is the integer value to write into the circular buffer

Interactive Command: NO

```
// Example :: CircBufWrite.sb (See in Firmware Zip file)
DIM rc
DIM circHandle
DIM i

rc = CircBufCreate(16,circHandle)
IF rc != 0 then
    PRINT "\nThe circular buffer was not created\n"
ELSE
    PRINT "\nThe circular buffer was created successfully\n"
ENDIF

//write 3 values into the circular buffer
FOR i = 1 TO 3
    rc = CircBufWrite(circHandle,i)
    IF rc != 0 then
        PRINT "\nFailed to write into the circular buffer\n"
    ELSE
        PRINT i;" was successfuly written to the circular buffer\r"
    ENDIF
NEXT
```

Expected output:

```
The circular buffer was created successfully
1 was successfuly written to the circular buffer
2 was successfuly written to the circular buffer
3 was successfuly written to the circular buffer
```

CIRCBUFWRITE is an extension function.

CircBufOverWrite

FUNCTION

This function is used to write an integer at the head end of the circular buffer and if there is no space available to write, then it will return with a failure resultcode but still write into the circular buffer by first discarding the oldest item.

CIRCBUFOVERWRITE (circHandle, nData)

Returns: INTEGER

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

Note if the buffer was full and the oldest value was overwritten then a non-zero value of 0x5103 will still be returned.

Arguments:

circHandle **byRef circHandle AS INTEGER**
This identifies the circular buffer to write into.

nData **byVal *nData* AS INTEGER**
This is the integer value to write into the circular buffer. It is always written into the buffer. Oldest is discarded to make space for this.

Interactive Command: NO

```
// Example :: CircBufOverwrite.sb (See in Firmware Zip file)
DIM rc,circHandle,i

rc = CircBufCreate(4,circHandle)
IF rc != 0 THEN
    PRINT "\nThe circular buffer was not created\n"
ELSE
    PRINT "\nThe circular buffer was created successfully\n"
ENDIF

FOR i = 1 TO 5
    rc = CircBufOverwrite(circHandle,i)
    IF rc == 0x5103 THEN
        PRINT "\nOldest value was discarded to write ";i
    ELSEIF rc !=0 THEN
        PRINT "\nFailed to write into the circular buffer"
    ELSE
        PRINT "\n";i
    ENDIF
NEXT
```

Expected Output:

```
The circular buffer was created successfully
1
2
3
4
Oldest value was discarded to write 5
```

CIRCBUFOVERWRITE is an extension function.

CircBufRead

FUNCTION

This function is used to read an integer from the tail end of the circular buffer. A nonzero resultcode will be returned if the buffer is empty or if the handle is invalid.

CIRCBUFREAD(circHandle, nData)

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation. If 0x5102 is returned it implies the buffer was empty so nothing was read.

Arguments:

circHandle **byRef circHandle AS INTEGER**
This identifies the circular buffer to read from.

nData **byRef *nData* AS INTEGER**
This is the integer value to read from the circular buffer

Interactive Command: NO

```
// Example :: CircBufRead.sb (See in Firmware Zip file)
DIM rc,circHandle,i,nData

rc = CircBufCreate(4,circHandle)
IF rc != 0 THEN
    PRINT "\nThe circular buffer was not created"
ELSE
    PRINT "\nThe circular buffer was created successfully\n"
    PRINT "Writing..."
ENDIF

FOR i = 1 TO 5
    rc = CircBufOverwrite(circHandle,i)
    IF rc == 0x5103 THEN
        PRINT "\nOldest value was discarded to write ";i;"\n"
    ELSEIF rc !=0 THEN
        PRINT "\nFailed TO write inTO the circular buffer"
    ELSE
        PRINT "\n";i
    ENDIF
NEXT

//read 4 values from the circular buffer
PRINT "\nReading...\n"
FOR i = 1 to 4
  rc = CircBufRead(circHandle,nData)
  IF rc == 0x5102 THEN
    PRINT "The buffer was empty"
  ELSEIF rc != 0 THEN
    PRINT "Failed to read from the circular buffer"
  ELSE
    PRINT nData;"\n"
  ENDIF
NEXT
```

Expected Output:

```
The circular buffer was created successfully
Writing...
1
2
3
4
Oldest value was discarded to write 5

Reading...
2
3
4
5
```

CIRCBUFREAD is an extension function.

CircBufItems

FUNCTION

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

CIRCBUFITEMS(circHandle, nItems)

Returns: INTEGER, a result code. The typical value is 0x0000, indicating a successful operation. If 0x5102 is returned it implies the buffer was empty so nothing was read.

Arguments:

circHandle **byRef circHandle AS INTEGER**
This identifies the circular buffer which needs to be queried.

nItems **byRef *nItems* AS INTEGER**
This returns the total items waiting to be read in the circular buffer.

Interactive Command: NO

```
// Example :: CircBufItems.sb (See in Firmware Zip file)
DIM rc,circHandle,i,nItems
rc = CircBufCreate(4,circHandle)
IF rc != 0 THEN
    PRINT "\nThe circular buffer was not created\n"
ELSE
    PRINT "\nThe circular buffer was created successfully\n"
ENDIF

FOR i = 1 TO 5
    rc = CircBufOverwrite(circHandle,i)
    IF rc == 0x5103 THEN
        PRINT "\nOldest value was discarded to write ";i
    ELSEIF rc !=0 THEN
        PRINT "\nFailed TO write inTO the circular buffer"
    ENDIF

    rc = CircBufItems(circHandle,nItems)
    IF rc == 0 THEN
        PRINT "\n";nItems;" items in the circular buffer"
    ENDIF
```

```
NEXT
```

Expected Output:

```
The circular buffer was created successfully

1 items in the circular buffer
2 items in the circular buffer
3 items in the circular buffer
4 items in the circular buffer
Oldest value was discarded to write 5
4 items in the circular buffer
```

CIRCBUFITEMS is an extension function.

Serial Communications Routines

In keeping with the event driven architecture of *smart* BASIC, the serial communications subsystem enables *smart* BASIC applications to be written which allow communication events to trigger the processing of user *smart* BASIC code.

Note that if a handler function returns a non-zero value then the WAITEVENT statement is reprocessed, otherwise the *smart* BASIC runtime engine will proceed to process the next statement **after** the WAITEVENT statement – not after the handlers ENDFUNC or EXITFUNC statement. Please refer to the detailed description of the WAITEVENT statement for further information.

UART (Universal Asynchronous Receive Transmit)

This section describes all the events and routines used to interact with the UART peripheral available on the platform. Depending on the platform, at a minimum, the UART will consist of a transmit, a receive, a CTS (Clear To Send) and RTS (Ready to Send) line. The CTS and RTS lines are used for hardware handshaking to ensure that buffers do not overrun.

If there is a need for the following low bandwidth status and control lines found on many peripherals, then the user is able to create those using the GPIO lines of the module and interface with those control/status lines using *smart* BASIC code.

Output	DTR	Data Terminal Ready
Input	DSR	Data Set Ready
Output/Input	DCD	Data Carrier Detect
Output/Input	RI	Ring Indicate

The lines DCD and RI are marked as Output or Input because it is possible, unlike a device like a PC where they are always inputs and modems where they are always outputs, to configure the pins to be either so that the device can adopt a DTE (Data Terminal Equipment) or DCE (Data Communications Equipment) role. *Please note that both DCD and RI have to be BOTH outputs or BOTH inputs, one cannot be an output and the other an input.*

UART Events

In addition to the routines for manipulating the UART interface, when data arrives via the receive line it is stored locally in an underlying ring buffer and then an event is generated.

Similarly when the transmit buffer is emptied, events are thrown from the underlying drivers so that user *smart* BASIC code in handlers can perform user defined actions.

The following is a detailed list of all events generated by the UART subsystem which can be handled by user code.

EVUARTRX This event is generated when one or more new characters have arrived and have been stored in the local ring buffer.

EVUARTTXEMPTY This event is generated when the last character is transferred from the local transmit ring buffer to the hardware shift register.

```
// Example :: EVUARTRX.sb (See in Firmware Zip file)
DIM rc
FUNCTION HndlrUartRx()
  PRINT "\nData has arrived\r"
ENDFUNC 1       //remain blocked in WAITEVENT

FUNCTION Btn0Pressed()
ENDFUNC 0

rc = GPIOBindEvent(0,16,1)
PRINT "\nPress Button 0 to exit this application \n"

ONEVENT EVUARTRX    CALL HndlrUartRx
ONEVENT EVGPIOCHAN0 CALL Btn0Pressed

WAITEVENT       //wait for rx, tx and modem status events
PRINT "Exiting..."
```

Expected Output:

```
Press Button 0 to exit this application
e

Data has arrived
Data has arrived
Data has arrived
Exiting...
```

Note: If you type unknown commands, an E007 error displays in UwTerminal.

```
// Example :: EVUARTTXEMPTY.sb (See in Firmware Zip file)
FUNCTION HndlrUartTxEty()
  PRINT "\nTx buffer is empty"
ENDFUNC 0

ONEVENT EVUARTTXEMPTY CALL HndlrUartTxEty
```

```
PRINT "\nSend this via uart"

WAITEVENT
```

Expected Output:

```
Send this via uart
Tx buffer is empty
```

UartOpen

Note: Until further notice, the parity parameter shall not be changed when using this function.

Function

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

If the uart is already open then this function will fail.

If this function is used to alter the communications parameters, like say the baudrate and the application exits to interactive mode, then those settings will be inherited by the interactive mode parser. Hence this is the only way to alter the communications parameters for Interactive mode.

While the uart is open, if a BREAK is sent to the module, then it will force the module into deep sleep mode as long as BREAK is asserted. As soon as BREAK is deasserted, the module will wake up through a reset as if it had been power cycled.

UARTOPEN (baudrate,txbuflen,rxbuflen,stOptions)

Returns: INTEGER Indicates success of command:

0	Opened successfully
0x5208	Invalid baudrate
0x5209	Invalid parity
0x520A	Invalid databits
0x520B	Invalid stopbits
0x520C	Cannot be DTE (because DCD and RI cannot be inputs)
0x520D	Cannot be DCE (because DCD and RI cannot be outputs)
0x520E	Invalid flow control request
0x520F	Invalid DTE/DCE role request
0x5210	Invalid length of stOptions parameter (must be 5 chrs)
0x5211	Invalid tx buffer length
0x5212	Invalid rx buffer length

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

baudrate ***byVal baudrate AS INTEGER***
The baudrate for the uart. Note that, the higher the baudrate, the more power will be drawn from the supply pins.
AT I 1002 or SYSINFO(1002) returns the minimum valid baudrate
AT I 1003 or SYSINFO(1003) returns the maximum valid baudrate

txbuflen ***byVal txbuflen AS INTEGER***
Set the transmit ring buffer size to this value. If set to 0 then a default value will be used by the underlying driver

rxbuflen ***byVal rxbuflen AS INTEGER***
Set the receive ring buffer size to this value. If set to 0 then a default value will be used by the underlying driver

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

Character Offset :
0: DTE/DCE role request - 'T' for DTE and 'C' for DCE
1: Parity – 'N' for none, 'O' for odd and 'E' for even
2: Databits – '5','6','7','8',9'
3: Stopbits – '1','2'
4: Flow Control – 'N' for none, 'H' for CTS/RTS hardware, 'X' for xon/xof

---

Note: There will be further restrictions on the options based on the hardware as for example a PC implementation cannot be configured as a DCE role. Likewise many microcontroller uart peripherals are not capable of 5 bits per character – but a PC is.

Note: In DTE equipment DCD and RI are inputs, while in DCE they are outputs.
Interactive Command: No

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

```
// Example :: UartOpen.sb (See in Firmware Zip file)
DIM rc

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

FUNCTION Btn0Pressed()
    UartClose()
ENDFUNC 0

rc = GPIOBindEvent(0,16,1)                  //For button0

ONEVENT EVUARTRX CALL HndlrUartRx
ONEVENT EVGPIOCHAN0 CALL Btn0Pressed
```

```
UartClose()                                     //Since Uart port is already open we must
                                                //close it before opening it again with
                                                //different settings.

//--- Open comport so that DCD and RI are inputs
rc = UartOpen(9600,0,0,"CN81H")                 //Open as DCE, no parity, 8 databits,
                                                //1 stopbits, cts/rts flow control

IF rc!= 0 THEN
    PRINT "\nFailed to open UART interface with error code ";INTEGER.H' rc
ELSE
    PRINT "\nUART open success"
ENDIF

PRINT "\nPress button0 to exit this application\n"

WAITEVENT                                       //wait for rx, events
PRINT "\nExiting..."
```

Expected Output:

```
UART open successful
Press button0 to exit this application
laird

Data has arrived
Data has arrived
Data has arrived
Data has arrived
Data has arrived
Data has arrived

Exiting...
```

UARTOPEN is a core function.

UartClose

FUNCTION

This subroutine is used to close a uart port which had been opened with UARTOPEN.

If after the uart is closed, a print statement is encountered, the uart will be automatically re-opened at the default rate (see hardware specific user manual for actual default value) so that the data generated by the PRINT statement is sent.

This routine will throw an exception if the uart is already closed, so if you are not sure then it is best to call it if UARTINFO(1) returns a non-zero value.

When this subroutine is invoked, the receive and transmit buffers are both flushed. If there is any data in either of these buffers when the UART is closed, it will be lost. This is because the execution of UARTCLOSE takes a very short amount of time, while the transfer of data from the buffers will take much longer.

In addition please note that when a *smart* BASIC application completes execution with the UART closed, it will automatically be reopened in order to allow continued communication with the module in Interactive Mode using the default communications settings.

UARTCLOSE()

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments: **None**

Interactive Command: No

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

```
//Example :: UartClose.sb (See in Firmware Zip file)
UartClose()

IF UartInfo(0)==0 THEN
    PRINT "\nThe Uart port was closed"
ELSE
    PRINT "\nThe Uart port was not closed"
ENDIF

IF UartInfo(0)!=0 THEN
    PRINT "\nand now it is open"
ENDIF
```

Expected Output:

```
The Uart port was closed
and now it is open
```

UARTCLOSE is a core subroutine.

UartCloseEx

FUNCTION

This function is used to close a uart port which had been opened with UARTOPEN depending on the flag mask in the input parameter.

Please see UartClose() for more details

UARTCLOSEEX(nFlags)

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation. If 0x5231 is returned it implies one of the buffers was not empty so not closed.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nFlags ***byVal nFlags AS INTEGER***
If Bit 0 is set, then only close if both rx and tx buffers are empty. Setting this bit to 0 has the same effect as UartClose() routine.
Bits 1 to 31 are for future use and must be set to 0.

Interactive Command: No

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

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

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

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

Expected Output:

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

UARTCLOSEEX is a core function.

UartInfo

FUNCTION

This function is used to query information about the default uart, such as buffer lengths, whether the port is already open or how many bytes are waiting in the receive buffer to be read.

UARTINFO (infoId)

Function

Returns: INTEGER The value associated with the type of uart information requested

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

infoId ***byVal infoId AS INTEGER***
This specifies the type of uart information requested as follows if the uart is open:-
0 := 1 (the port is open), 0 (the port is closed)
And the following specify the type of uart information when the port is open:-
1 := Receive ring buffer capacity
2 := Transmit ring buffer capacity
3 := Number of bytes waiting to be read from receive ring buffer
4 := Free space available in transmit ring buffer
5 := Number of bytes still waiting to be sent in transmit buffer
6 := Total number of bytes waiting in rx and tx buffer

If the uart is closed, then regardless of the value of ***infoId***, a 0 will be returned.
Note: UARTINFO(0) will always return the open/close state of the uart.
Interactive Command: No

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

```
//Example :: UartInfo.sb (See in Firmware Zip file)
DIM rc,start

UartClose()

IF UartInfo(0)==0 THEN
    PRINT "\nThe Uart port was closed\n"
ELSE
    PRINT "\nThe Uart port was not closed\n"
ENDIF

PRINT "\nReceive ring buffer capacity:          ";UartInfo(1)
PRINT "\nTransmit ring buffer capacity:     ";UartInfo(2)
PRINT "\nNo. bytes waiting in transmit buffer:  ";UartInfo(5)

start = GetTickCount()
DO
UNTIL UartInfo(5)==0
PRINT "\n\nTook ";GetTickSince(start);" milliseconds for transmit buffer to be
emptied"
```

Expected Output:

```
The Uart port was closed

Receive ring buffer capacity:              256
Transmit ring buffer capacity:             256
No. bytes waiting in transmit buffer:      134

Took 142 milliseconds for transmit buffer to be emptied
```

UARTINFO is a core subroutine.

UartWrite

FUNCTION

This function is used to transmit a string of characters.

UARTWRITE (strMsg)

Returns: INTEGER 0 to N : Actual number of bytes successfully written to the local transmit ring buffer

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPEN (or auto-opened with PRINT statement)

Arguments:

strMsg ***byRef strMsg AS STRING***
The array of bytes to be sent. STRLEN(strMsg) bytes are written to the local transmit ring buffer. If STRLEN(strMsg) and the return value are not the same,this implies the transmit buffer did not have enough space to accommodate the data. If the return value does not match the length of the original string, then use STRSHIFTLEFT function to drop the data from the string, so that subsequent calls to this function only retries with data which was not placed in the output ring buffer.

Interactive Command: No

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

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

```
//Example :: UartWrite.sb (See in Firmware Zip file)
DIM rc,str$,i,done,d
```

```
 //str$ contains a lot of space so that we can satisfy the condition in the IF
statement
 str$="
Hello World"

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

 rc=UartWrite(str$)

 //Shift 'str$' if there isn't enough space in the buffer until 'str$' can be written
 WHILE done == 0
     IF rc < StrLen(str$) THEN
         PRINT rc;" bytes written"
         PRINT "\nStill have ";StrLen(str$)-rc;" bytes to write\n"
         PRINT "\nShifting 'str$' by ";rc
         StrShiftLeft(str$,rc)
         done = 0
     ELSE
         PRINT "\nString 'str$' written successfully"
         done=1
     ENDIF
 ENDWHILE

 ONEVENT EVUARTTXEMPTY CALL HndlrUartTxEty

 WAITEVENT
```

Expected Output:

```
256 bytes written
Still have 18 bytes to write

Shifting 'str$' by 256
String 'str$' written successfully
Tx buffer is now empty
```

UARTWRITE is a core subroutine.

UartRead

FUNCTION

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

UARTREAD(strMsg)

Returns: INTEGER 0 to N : The total length of the string variable – not just what got appended. This means the caller does not need to call strlen() function to determine how many bytes in the string that need to be processed.

Exceptions

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

Arguments:

strMsg ***byRef strMsg AS STRING***
The content of the receive buffer will get **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

```
//Example :: UartRead.sb (See in Firmware Zip file)
DIM rc,strLength,str$
str$="Your name is "

FUNCTION HndlrUartRx()
    TimerStart(0,100,0)     //Allow enough time for data to reach rx buffer
ENDFUNC 1

FUNCTION HndlrTmr0()
    strLength=UartRead(str$)
    PRINT "\n";str$
ENDFUNC 0

ONEVENT EVTMR0      CALL HndlrTmr0
ONEVENT EVUARTRX    CALL HndlrUartRx

PRINT "\nWhat is your name?\n"

WAITEVENT
```

Expected Output:

```
What is your name?
David

Your name is David
```

UARTREAD is a core subroutine.

UartReadN

FUNCTION

This function is used to read the content of the receive buffer and **append** it to the string variable supplied but it ensures that the string is not longer than nMaxLen.

UARTREADN(strMsg, nMaxLen)

Returns: INTEGER 0 to N : The total length of the string variable – not just what got appended. This means the caller does not need to call strlen() function to determine how many bytes in the string that need to be processed.

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

Arguments:

strMsg ***byRef strMsg AS STRING***
The content of the receive buffer will get **appended** to this string.

nMaxLen ***byval nMaxLen AS INTEGER***
The output string strMsg will never be longer than this value. If a value less than 1 is specified, it will be clipped to 1 and if > that 0xFFFF it will be clipped to 0xFFFF.

Interactive Command: No

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

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

```
//Example
DIM rc,strLength,str$
str$="Your name is "

FUNCTION HndlrUartRx()
    TimerStart(0,100,0)     //Allow enough time for data to reach rx buffer
ENDFUNC 1

FUNCTION HndlrTmr0()
    strLength=UartReadn(str$,11)
    PRINT "\n";str$
ENDFUNC 0

ONEVENT EVTMR0      CALL HndlrTmr0
ONEVENT EVUARTRX    CALL HndlrUartRx

PRINT "\nWhat is your name?\n"

WAITEVENT
```

Expected Output:

```
What is your name?
David

Your name i
```

UARTREADN is a core subroutine.

UartReadMatch

FUNCTION

This function is used to read the content of the underlying receive ring buffer and **append** it to the string variable supplied, up to and including the first instance of the specified matching character OR the end of the ring buffer.

This function is very useful when interfacing with a peer which sends messages terminated by a constant character such as a carriage return (0x0D). In that case, in the handler, if the return value is greater than 0, it implies a terminated message arrived and so can be processed further.

UARTREADMATCH(strMsg , chr)

Returns: INTEGER Indicates the presence of the match character in **strMsg** as follows:

0 : data **may** have been appended to the string, but no matching character.
1 to N : The total length of the string variable up to and including the match **chr**.

Note: When 0 is returned you can use STRLEN(strMsg) to determine the length of data stored in the string. On some platforms with low amount of RAM resources, the underlying code may decide to leave the data in the receive buffer rather than transfer it to the string.

Exceptions

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

Arguments:

strMsg ***byRef strMsg AS STRING***
The content of the receive buffer will get **appended** to this string up to and including the match character.

chr ***byVal chr AS INTEGER***
The character to match in the receive buffer, for example the carriage return character 0x0D

Interactive Command: No

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

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

```
//Example :: UartReadMatch.sb (See in Firmware Zip file)
DIM rc,str$,ret,char,str2$
ret=1                                 //Function return value
char=13                               //ASCII decimal value for 'carriage return'

str$="Your name is "
str2$="\n\nMatch character ' ' not found \nExiting.."


FUNCTION HndlrUartRx()
    TimerStart(0,10,0)                //Allow time for data to reach rx buffer
ENDFUNC 1

FUNCTION HndlrTmr0()
    rc = UartReadMatch(str$,char)
    PRINT "\n";str$
    IF rc==0 THEN
        rc=StrSetChr(str2$,char,19) //Insert 'char', the match character
        PRINT str2$
        str2$="\n\nMatch character  not found \nExiting.."  //reset str2$
        ret=0
    ELSE
        PRINT "\n\n\nNow type something without the letter 'a'\n"
        str$="You sent "                                    //reset str$
        char=97                     //ASCII decimal value for 'a'
        ret=1
    ENDIF
ENDFUNC ret

ONEVENT EVTMR0      CALL HndlrTmr0
ONEVENT EVUARTRX    CALL HndlrUartRx

PRINT "\nWhat is your name?\n"

WAITEVENT
```

Expected Output:

```
What is your name?

Your name is David

Now type something without the letter 'a'

You sent hello

Match character 'a' not found
Exiting..
```

UARTREADMATCH is a core subroutine.

UartFlush

SUBROUTINE

This subroutine is used to flush either or both receive and transmit ring buffers.

This is useful when, for example, you have a character terminated messaging system and the peer sends a very long message and the input buffer fills up. In that case, there is no more space for an incoming termination character and the RTS handshaking line would have been asserted so the message system will stall. A flush of the receive buffer is the best approach to recover from that situation.

Note: Execution of UARTFLUSH is much quicker than the time taken to transmit data to/from the buffers

UARTFLUSH(bitMask)

Exceptions

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

Arguments:

bitMask ***byVal bitMask AS INTEGER***

This bit mask is used to choose which ring buffer to flush.

Bit	Description
0	Set to flush the rx buffer
1	Set to flush the tx buffer
	Set both bits to flush both buffers.

Interactive Command: No

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

```
 //Example :: UartFlushRx.sb (See in Firmware Zip file)
 FUNCTION HndlrUartRx()
     TimerStart(0,2,0)                                  //Allow time for data to reach rx
buffer
 ENDFUNC 1

 FUNCTION HndlrTmr0()
     PRINT UartInfo(3);" bytes in the rx buffer,\n"
     UartFlush(01)                                      //clear rx buffer
     PRINT UartInfo(3);" bytes in the rx buffer after flushing"
 ENDFUNC 0

 ONEVENT EVUARTRX    CALL HndlrUartRx
 ONEVENT EVTMR0      CALL HndlrTmr0

 PRINT "\nSend me some text\n"

 WAITEVENT
```

Expected Output:

```
Send me some data
Laird
6 bytes in the rx buffer,
0 bytes in the rx buffer after flushing
```

```
//Example :: UartFlushTx.sb (See in Firmware Zip file)
DIM s$ : s$ = "Hello World"
DIM rc : rc = UartWrite(s$)

UartFlush(10)       //Will flush before all chars have been transmitted
PRINT UartInfo(5); " bytes in the tx buffer after flushing"
```

Expected Output:

```
H0 bytes in the tx buffer after flushing
```

UARTFLUSH is a core subroutine.

UartGetCTS

FUNCTION

This function is used to read the current state of the CTS modem status input line.

If the device does not expose a CTS input line, then this function will return a value that signifies an asserted line.

UARTGETCTS()

Returns: INTEGER Indicates the status of the CTS line:

0 : CTS line is NOT asserted
1 : CTS line is asserted

Exceptions

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

Arguments: **None**

Interactive Command: No

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

```
//Example :: UartGetCTS.sb (See in Firmware Zip file)

IF UartGetCTS()==0 THEN
```

```
    PRINT "\nCTS line is not asserted"
ELSEIF UartGetCTS()==1 THEN
    PRINT "\nCTS line is asserted"
ENDIF
```

Expected Output:

```
CTS line is not asserted
```

UARTGETCTS is a core subroutine.

UartSetRTS

SUBROUTINE

This function is used to set the state of the RTS modem control line. When the UART port is closed, the RTS line can be configured as an input or an output and can be available for use as a general purpose input/output line.

When the uart port is opened, the RTS output is automatically defaulted to the asserted state. If flow control was enabled when the port was opened then the RTS output cannot be manipulated as it is owned by the underlying driver.

UARTSETRTS(newState)

Exceptions

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

Arguments:

newState ***byVal newState AS INTEGER***
0 to deassert and non-zero to assert

Interactive Command: No

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

Note: This subroutine is not implemented in some modules. Refer to module specific user manual if this is available

UARTSETRTS is a core subroutine.

UartBREAK

SUBROUTINE

This subroutine is used to assert/deassert a BREAK on the transmit output line. A BREAK is a condition where the line is in non idle state (that is 0v) for more than 10 to 13 bit times, depending on whether parity has been enabled and the number of stopbits.

On certain platforms the hardware may not allow this functionality, contact Laird to determine if your device has the capability. On platforms that do not have this capability, this routine has no effect.

Not all modules have the capability to send a BREAK signal so please refer to the specific user manual if the functionality is present even if this function is present..

UARTBREAK(state)

Exceptions

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

Arguments:

newState ***byVal newState AS INTEGER***
0 to deassert and non-zero to assert

Interactive Command: No

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

UARTBREAK is a core subroutine.

I2C - Also known as Two Wire Interface (TWI)

Note: The routines in this section will only work if I2C is supported on the platform

This section describes all the events and routines used to interact with the I2C peripheral if it is 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.

Some modules can only be configured as an I2C master with the additional constraint that it be the only master on the bus and only 7 bit slave addressing is supported. Please refer to the specific user manual for clarification.

The two signal lines are called SCL and SDA. The former is the clock line which is always sourced by the master and the latter is a bi-directional data line which can be driven by any device on the bus.

It is essential to remember that pull up resistors on both SCL and SDA lines are not provided in the module and MUST be provided external to the module.

A very good introduction to I2C can be found at http://www.i2c-bus.org/i2c-primer/ and the reader is encouraged to refer to it before using the api described in this section.

I2C Events

The API provided in the module is synchronous and so there is no requirement for events.

I2cOpen

FUNCTION

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

See the module reference manual for details of which pins expose the SCL and SDA functions.

I2COPEN (nClockHz, nCfgFlags, nHande)

Returns: INTEGER Indicates success of command:

0	Opened successfully
0x5200	Driver not found
0x5207	Driver already open
0x5225	Invalid Clock Frequency Requested
0x521D	Driver resource unavailable
0x5226	No free PPI channel
0x5202	Invalid Signal Pins
0x5219	I2C not allowed on pins specified

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nClockHz ***byVal nClockHz AS INTEGER***
This is the clock frequency to use, See module specific documentation for valid frequencies.

nCfgFlags ***byVal nCfgFlags AS INTEGER***
This is a bit mask used to configure the I2C interface. All unused bits are allocated as for future use and MUST be set to 0. Used bits are as follows:-

Bit	Description
0	If set, then a 500 microsecond low pulse will NOT be sent on open. This low pulse is used to create a start and stop condition on the bus so that any signal transitions on these lines prior to this open which may have confused a slave can initialise that slave to a known state. The STOP condition should be detected by the slave.
1-31	Unused and MUST be set to 0

nHandle ***byRef nHandle AS INTEGER***
The handle for this interface will be returned in this variable if it was successfully opened. This handle is subsequently used to read/write and close the interface.

Related Commands: I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cOpen.sb (See in Firmware Zip file)
DIM handle
DIM rc : rc=I2cOpen(100000,0,handle)

IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.h' rc
ELSE
    PRINT "\nI2C open success \nHandle is ";handle
ENDIF
```

Expected Output:

```
I2C open success
Handle is 0
```

I2COPEN is a core function.

I2cClose

SUBROUTINE

This subroutine is used to close a I2C port which had been opened with I2COPEN.

This routine is safe to call if it is already closed.

I2CCLOSE(handle)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

handle ***byVal handle AS INTEGER***
This is the handle value that was returned when I2COPEN was called which identifies the I2C interface to close.

Interactive Command: No

Related Commands: I2COPEN, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cClose.sb (See in Firmware Zip file)
DIM handle
DIM rc : rc=I2cOpen(100000,0,handle)

IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.h' rc
```

```
ELSE
    PRINT "\nI2C open success \nHandle is ";handle
ENDIF

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

I2CCLOSE is a core subroutine.

I2cWriteREG8

SUBROUTINE

This function is used to write an 8 bit value to a register inside a slave which is identified by an 8 bit register address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CWRITEREG8(nSlaveAddr, nRegAddr, nRegValue)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr ***byVal nSlaveAddr AS INTEGER***
This is the address of the slave in range 0 to 127.

nRegAddr ***byVal nRegAddr AS INTEGER***
This is the 8 bit register address in the addressed slave in range 0 to 255.

nRegValue ***byVal nRegValue AS INTEGER***
This is the 8 bit value to written to the register in the addressed slave. Please note only the lowest 8 bits of this variable are written.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cWriteReg8.sb (See in Firmware Zip file)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc, handle, nSlaveAddr, nRegAddr, nRegVal


//--- Open I2C Peripheral
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.H' rc
ELSE
    PRINT "\nI2C open success"
ENDIF


//--- Write 'nRegVal' to register 'nRegAddr'
```

```
nSlaveAddr=0x6f :  nRegAddr = 23 : nRegVal = 0x63
rc = I2cWriteReg8(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Write to slave/register "; INTEGER.H'rc
ELSE
    PRINT "\n";nRegVal; " written successfully to register ";nRegAddr
ENDIF

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

Expected Output:

```
I2C open success
99 written successfully to register 23
```

I2CWRITEREG8 is a core function.

I2cReadREG8

SUBROUTINE

This function is used to read an 8 bit value from a register inside a slave which is identified by an 8 bit register address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CREADREG8(nSlaveAddr, nRegAddr, nRegValue)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr ***byVal nSlaveAddr AS INTEGER***
This is the address of the slave in range 0 to 127.

nRegAddr ***byVal nRegAddr AS INTEGER***
This is the 8 bit register address in the addressed slave in range 0 to 255.

nRegValue ***byRef nRegValue AS INTEGER***
The 8 bit value from the register in the addressed slave will be returned in this variable.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cReadReg8.sb (See in Firmware Zip file)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc, handle, nSlaveAddr, nRegAddr, nRegVal
```

```
//--- Open I2C Peripheral
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.H' rc
ELSE
    PRINT "\nI2C open success"
ENDIF


//---Read value from address 0x34
nSlaveAddr=0x6f :  nRegAddr = 23
rc = I2cReadReg8(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Read from slave/register "; INTEGER.H'rc
ELSE
    PRINT "\nValue read from register is ";nRegVal
ENDIF

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

Expected Output:

```
I2C open success
Value read from register is 99
```

I2CREADREG8 is a core function.

I2cWriteREG16

SUBROUTINE

This function is used to write a 16 bit value to 2 registers inside a slave and the first register is identified by an 8 bit register address supplied.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CWRITEREG16(nSlaveAddr, nRegAddr, nRegValue)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr ***byVal nSlaveAddr AS INTEGER***
This is the address of the slave in range 0 to 127.

nRegAddr ***byVal nRegAddr AS INTEGER***
This is the 8 bit start register address in the addressed slave in range 0 to 255.

nRegValue ***byVal nRegValue AS INTEGER***
This is the 16 bit value to be written to the register in the addressed slave. Please note only the lowest 16 bits of this variable are written.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cWriteReg16.sb (See in Firmware Zip file)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc, handle, nSlaveAddr, nRegAddr, nRegVal

//--- Open I2C Peripheral
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.H' rc
ELSE
    PRINT "\nI2C open success"
ENDIF

//--- Write 'nRegVal' to register 'nRegAddr'
nSlaveAddr=0x6f :  nRegAddr = 0x34 : nRegVal = 0x4210
rc = I2cWriteReg16(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Write to slave/register "; INTEGER.H'rc
ELSE
    PRINT "\n";nRegVal; " written successfully to register ";nRegAddr
ENDIF

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

Expected Output:

```
I2C open success
16912 written successfully to register 52
```

I2CWRITEREG16 is a core function.

I2cReadREG16

SUBROUTINE

This function is used to read a 16 bit value from two registers inside a slave which is identified by an 8 bit register address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CREADREG16(nSlaveAddr, nRegAddr, nRegValue)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr ***byVal nSlaveAddr AS INTEGER***
This is the address of the slave in range 0 to 127.

nRegAddr ***byVal nRegAddr AS INTEGER***
This is the 8 bit start register address in the addressed slave in range 0 to 255.

nRegValue ***byRef nRegValue AS INTEGER***
The 16 bit value from two registers in the addressed slave will be returned in this variable.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cReadReg16.sb (See in Firmware Zip file)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc, handle, nSlaveAddr, nRegAddr, nRegVal

//--- Open I2C Peripheral
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.H' rc
ELSE
    PRINT "\nI2C open success"
ENDIF

//---Read value from address 0x34
nSlaveAddr=0x6f :  nRegAddr = 0x34
rc = I2cReadReg16(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Read from slave/register "; INTEGER.H'rc
ELSE
    PRINT "\nValue read from register is ";nRegVal
ENDIF

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

Expected Output:

```
I2C open success
Value read from register is 16912
```

I2CREADREG16 is a core function.

I2cWriteREG32

SUBROUTINE

This function is used to write a 32 bit value to 4 registers inside a slave and the first register is identified by an 8 bit register address supplied.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CWRITEREG32(nSlaveAddr, nRegAddr, nRegValue)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr ***byVal nSlaveAddr AS INTEGER***
This is the address of the slave in range 0 to 127.

nRegAddr ***byVal nRegAddr AS INTEGER***
This is the 8 bit start register address in the addressed slave in range 0 to 255.

nRegValue ***byVal nRegValue AS INTEGER***
This is the 32 bit value to be written to the register in the addressed slave.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cWriteReg32.sb (See in Firmware Zip file)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal
DIM rc : rc=I2cOpen(100000,0,handle)

IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code ";INTEGER.h' rc
ELSE
    PRINT "\nI2C open success"
ENDIF

nSlaveAddr = 0x6f :  nRegAddr = 0x56 : nRegVal = 0x4210FEDC
rc = I2cWriteReg32(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Write to slave/register "; INTEGER.H'rc
ELSE
    PRINT "\n";nRegVal; " written successfully to register ";nRegAddr
ENDIF

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

Expected Output:

```
I2C open success
1108410076 written successfully to register 86
```

I2CWRITEREG32 is a core function.

I2cReadREG32

FUNCTION

This function is used to read a 32 bit value from four registers inside a slave which is identified by a starting 8 bit register address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CREADREG32(nSlaveAddr, nRegAddr, nRegValue)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr ***byVal nSlaveAddr AS INTEGER***
This is the address of the slave in range 0 to 127.

nRegAddr ***byVal nRegAddr AS INTEGER***
This is the 8 bit start register address in the addressed slave in range 0 to 255.

nRegValue ***byRef nRegValue AS INTEGER***
The 32 bit value from four registers in the addressed slave will be returned in this variable.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cReadREG32.sb (See in Firmware Zip file)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal
DIM rc : rc=I2cOpen(100000,0,handle)

IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code ";INTEGER.h' rc
ELSE
    PRINT "\nI2C open success"
ENDIF


//---Read value from address 0x56
nSlaveAddr = 0x6f :  nRegAddr = 0x56
rc = I2cReadReg32(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to read from slave/register"
ELSE
    PRINT "\nValue read from register is "; nRegVal
ENDIF

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

Expected Output:

```
I2C open success
Value read from register is 1108410076
```

I2CREADREG16 is a core function.

I2cWriteRead

SUBROUTINE

This function is used to write from 0 to 255 bytes and then immediately after that read 0 to 255 bytes in a single transaction from the addressed slave. It is a 'free-form' function that allows communication with a slave which has a 10 bit address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CWRITEREAD(nSlaveAddr, stWrite$, stRead$, nReadLen)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr ***byVal nSlaveAddr AS INTEGER***
This is the address of the slave in range 0 to 127.

stWrite$*** ***byRef stWrite$ AS STRING
This string contains the data that must be written first. If the length of this string is 0 then the write phase is bypassed.

stRead$*** ***byRef stRead$ AS STRING
This string will be written to with data read from the slave if and only if nReadLen is not 0.

nReadLen ***byRef nReadLen AS INTEGER***
On entry this variable contains the number of bytes to be read from the slave and on exit will contain the actual number that were actually read. If the entry value is 0, then the read phase will be skipped.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cWriteRead.sb (See in Firmware Zip file)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc
DIM handle
DIM nSlaveAddr
DIM stWrite$, stRead$, nReadLen
```

```
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
  PRINT "\nFailed to open I2C interface with error code ";integer.h' rc
ELSE
  PRINT "\nI2C open success"
ENDIF

//Write 2 bytes and read 0
nSlaveAddr=0x6f :  stWrite$ = "\34\35" : stRead$="" : nReadLen = 0
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
IF rc!= 0 THEN
  PRINT "\nFailed to WriteRead "; integer.h'rc
ELSE
  PRINT "\nWrite = ";StrHexize$(stWrite$);" Read = ";StrHexize$(stRead$)
ENDIF

//Write 3 bytes and read 4
nSlaveAddr=0x6f :  stWrite$ = "\34\35\43" : stRead$="" : nReadLen = 4
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
IF rc!= 0 THEN
  PRINT "\nFailed to WriteRead "; integer.h'rc
ELSE
  PRINT "\nWrite = ";StrHexize$(stWrite$);" Read = ";StrHexize$(stRead$)
ENDIF

//Write 0 bytes and read 8
nSlaveAddr=0x6f :  stWrite$ = "" : stRead$="" : nReadLen = 8
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
IF rc!= 0 THEN
  PRINT "\nFailed to WriteRead "; integer.h'rc
ELSE
  PRINT "\nWrite = ";StrHexize$(stWrite$);" Read = ";StrHexize$(stRead$)
ENDIF

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

Expected Output:

```
I2C open success
Write = 3435 Read =
Write = 343543 Read = 1042D509
Write =  Read = 2B322380ED236921
```

I2CWRITEREAD is a core function.

SPI Interface

Note: The routines in this section will only work if SPI is supported on the hardware you are developing for.

This section describes all the events and routines used to interact with the SPI peripheral if it is available on the platform.

The three signal lines are called SCK, MOSI and MISO, where the first two are outputs and the last is an input.

A very good introduction to SPI can be found at http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus and the reader is encouraged to refer to it before using the api described in this section.

It is possible to configure the interface to operate in any one of the 4 modes defined for the SPI bus which relate to the phase and polarity of the SCK clock line in relation to the data lines MISO and MOSI. In addition, the clock frequency can be configured from 125,000 to 8000000 and it can be configured so that it shifts data in/out most significant bit first or last.

Note: A dedicated SPI Chip Select (CS) line is not provided and it is up to the developer to dedicate any spare gpio line for that function if more than one SPI slave is connected to the bus. The SPI interface in this module assumes that prior to calling SPIREADWRITE, SPIREAD or SPIWRITE functions the slave device has been selected via the appropriate gpio line.

SPI Events

The API provided in the module is synchronous and so there is no requirement for events.

SpiOpen

FUNCTION

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

SPIOPEN (nMode, nClockHz, nCfgFlags, nHande)

Returns: INTEGER Indicates success of command:

0	Opened successfully
0x5200	Driver not found
0x5207	Driver already open
0x5225	Invalid Clock Frequency Requested
0x521D	Driver resource unavailable
0x522B	Invalid mode

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nMode ***byVal nMode AS INTEGER***

This is the mode, as in phase and polarity of the clock line, that the interface shall operate at. Valid values are 0 to 3 inclusive:

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

nClockHz ***byVal nClockHz AS INTEGER***
This is the clock frequency to use, and can be one of 125000, 250000, 500000, 1000000, 2000000, 4000000 or 8000000.

nCfgFlags ***byVal nCfgFlags AS INTEGER***
This is a bit mask used to configure the SPI interface. All unused bits are allocated as *for future use* and MUST be set to 0. Used bits are as follows:-

Bit	Description
0	If set then the least significant bit is clocked in/out first.
1-31	Unused and MUST be set to 0

nHandle ***byRef nHandle AS INTEGER***
The handle for this interface will be returned in this variable if it was successfully opened. This handle is subsequently used to read/write and close the interface.

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

SPIOPEN is a core function.

On the following page is an example which demonstrates usage of all the SPI related functions available in smartBASIC.

SPI Example

```
//Example :: SpiExample.sb (See in Firmware Zip file)

//The SPI slave used here is the Microchip 25A512
//See http://ww1.microchip.com/downloads/en/DeviceDoc/22237C.pdf

DIM rc
DIM h   //handle
DIM rl  //readlen
DIM rd$,wr$,p$
DIM wren


//-------------------------------------------------------------
//Get eeprom Status Register
//-------------------------------------------------------------
FUNCTION EepromStatus()
  GpioWrite(13,0)
  wr$="\05\00" : rd$="" : rc=SpiReadWrite(wr$,rd$)
  GpioWrite(13,1)
ENDFUNC StrGetChr(rd$,1)

//-------------------------------------------------------------
//Wait for WR bit in status flag to reset
//-------------------------------------------------------------
SUB WaitWrite()
  DO
    GpioWrite(13,0)
    wr$="\05\00" : rd$="" : rc=SpiReadWrite(wr$,rd$)
```

```
    GpioWrite(13,1)
  UNTIL ((StrGetChr(rd$,1)&1)==0)
ENDSUB

//-------------------------------------------------------------
//Enable writes in eeprom
//-------------------------------------------------------------
SUB EnableWrite()
  GpioWrite(13,0)
  wr$="\06" : rd$="" : rc=SpiWrite(wr$)
  GpioWrite(13,1)
ENDSUB

//-------------------------------------------------------------
// Configure the Chip Select line using SIO13 as an output
//-------------------------------------------------------------
rc= GpioSetFunc(13,2,1)
// ensure CS is not enabled
GpioWrite(13,1)

//-------------------------------------------------------------
//open the SPI
//-------------------------------------------------------------
rc=SpiOpen(0,125000,0,h)


//.............................................................
//Write DEADBEEFBAADC0DE 8 bytes to memory at location 0x0180
//.............................................................
EnableWrite()
wr$="\02\01\80\DE\AD\BE\EF\BA\AD\C0\DE"
PRINT "\nWriting to location 0x180  ";StrHexize$(wr$)
GpioWrite(13,0)
rc=SpiWrite(wr$)
GpioWrite(13,1)
WaitWrite()

//.............................................................
//Read from written location
//.............................................................
wr$="\03\01\80\00\00\00\00\00\00\00"
rd$=""
GpioWrite(13,0)
rc=SpiReadWrite(wr$,rd$)
GpioWrite(13,1)
PRINT "\nData at location 0x0180 is ";StrHexize$(rd$)

//.............................................................
//Prepare for reads from location 0x180 and then read 4 and then 8 bytes
//.............................................................
wr$="\03\01\80"
GpioWrite(13,0)
rc=SpiWrite(wr$)
rd$=""
rc=SpiRead(rd$,4)
PRINT "\nData at location 0x0180 is ";StrHexize$(rd$)
rd$=""
rc=SpiRead(rd$,8)
GpioWrite(13,1)
PRINT "\nData at location 0x0184 is ";StrHexize$(rd$)
```

```
//-------------------------------------------------------------
//close the SPI
//-------------------------------------------------------------
SpiClose(h)
```

Expected Output:

```
Writing to location 0x180  020180DEADBEEFBAADC0DE
Data at location 0x0180 is 000000DEADBEEFBAADC0DE
Data at location 0x0180 is DEADBEEF
Data at location 0x0184 is BAADC0DEFFFFFFFF
```

SpiClose

SUBROUTINE

This subroutine is used to close a SPI port which had been opened with SPIOPEN.

This routine is safe to call if it is already closed.

SPICLOSE(handle)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

handle ***byVal handle AS INTEGER***
This is the handle value that was returned when SPIOPEN was called which identifies the SPI interface to close.

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
//Example :: See SpiExample.sb
```

SPICLOSE is a core subroutine.

SpiReadWrite

FUNCTION

This function is used to write data to a SPI slave and at the same time read the same number of bytes back. Every 8 clock pulses result in one byte being written and one being read.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one SPI interface is made available.

SPIREADWRITE(stWrite$, stRead$)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

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

Arguments:

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

stRead$*** ***byRef stRead$ AS STRING
While the data in stWrite$ is being written, the slave sends data back and that data is stored in this variable. Note that on exit this variable will contain the same number of bytes as stWrite$.

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
//Example :: See SpiExample.sb
```

SPIWRITEREAD is a core function.

SpiWrite

FUNCTION

This function is used to write data to a SPI slave and any incoming data will be ignored.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one SPI interface is made available.

SPIWRITE(stWrite$)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

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

Arguments:

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

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
//Example :: See SpiExample.sb
```

SPIWRITE is a core function.

SpiRead

FUNCTION

This function is used to read data from a SPI slave.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one SPI interface is made available.

SPIREAD(stRead$, nReadLen)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

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

Arguments:

stRead$*** ***byRef stRead$ AS STRING
This string will contain the data that is read from the slave.

nReadLen ***byVal nReadLen AS INTEGER***
This specifies the number of bytes to be read from the slave.

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
//Example :: See SpiExample.sb
```

SPIREAD is a core function.

Cryptographic Functions

This section describes cryptographic functions that can be used to encrypt and decrypt data, over and above and in addition to any crypting applied at the transport layer.

In cryptography there are many algorithms which could be symmetric or assymetric. Each function described in this section will detail the type and modes catered for.

AesSetKeyIV

FUNCTION

This function is used to initialise a context for AES encryption and decription using the mode, key and initialisation vector supplied. The modes that are catered for is EBC and CBC with a block size of 128 bits.

AESSETKEYIV (mode, blockSize,key$, initVector$)

Returns: INTEGER

Will be 0x0000 if the context was created successfully. Otherwise an appropriate resultcode will be returned which will convery the reason it failed.

Arguments:

mode **BYVAL mode AS INTEGER**
This shall be as follows:-
0x100 for EBC mode
0x101 for EBC mode but data is XORed with same initVector$ everytime
0x200 for CBC mode

blockSize **BYVAL blockSize AS INTEGER**
Must always be set to16, which is the size in bytes.

***key$*** **BYREF *key$* AS STRING**
This string specifies the key to use for encryption and decryption and MUST be exactly 16 bytes long

***initVector$*** **BYREF *initVector$* AS STRING**
If mode is 0x101 or 0x200, then this string MUST be supplied and it shall be 16 bytes long. It is left to the caller to ensure a sensible value is supplied. For example, providing a string where all bytes is 0 is going to be of no value.

Interactive Command: NO

```
//Example :: AesSetKeyIv.sb (See in Firmware Zip file)
DIM key$, initVector$
DIM rc
//Create context for EBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$=""   //EBC does not require initialisation vector
rc=AesSetKeyIv(0x100,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nEBC context created successfully"
ELSE
  PRINT "\nFailed to create EBC context"
ENDIF
//Create context for EBC mode with XOR, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$="\FF\01\FF\03\FF\05\FF\07\FF\09\FF\0B\FF\0D\FF\0F"
rc=AesSetKeyIv(0x101,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nEBC-XOR context created successfully"
ELSE
  PRINT "\nFailed to create EBC-XOR context"
ENDIF
//Create context for CBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$="\FF\01\FF\03\FF\05\FF\07\FF\09\FF\0B\FF\0D\FF\0F"
rc=AesSetKeyIv(0x200,16,key$,initVector$)
IF rc==0 THEN
```

```
  PRINT "\nCBC context created successfully"
ELSE
  PRINT "\nFailed to create CBC context"
ENDIF
```

Expected Output:

```
EBC context created successfully
EBC-XOR context created successfully
CBC context created successfully
```

AESSETKEYIV is a core language function.

AesEncrypt

FUNCTION

This function is used to encrypt a string up to 16 bytes long using the context that was precreated using the most recent call of the function AesSetKeyIv.

For all modes, AesSetKeyIV is called only once which means in CBC mode the cyclic data is kept in the context object that was created by AesSetKeyIV.

For example, on the BL600, which has AES 128 **encryption** hardware assist, the function has been timed to take roughly 125 microseconds, otherwise it can take about 500 microseconds on a 16Mhz ARM Cortex M0 processor.

AESENCRYPT (inData$,outData$)

Returns: INTEGER

Will be 0x0000 if the data was encrypted successfully. Otherwise an appropriate resultcode will be returned which will convey the reason it failed. ALWAYS check this.

Arguments:

***inData$*** **BYREF *inData$* AS STRING**
This string is up to 16 bytes long and should contain the data to encrypt

***outData$*** **BYREF *outData$* AS STRING**
On exit, if the function was successful, then this string will contain the encrypted cypher data. If unsuccessful, then string will be 0 bytes long.

Interactive Command: NO

```
//Example :: AesEncrypt.sb (See in Firmware Zip file)
DIM key$, initVector$
DIM inData$, outData$
DIM rc
//Create context for EBC mode, 128 bit
```

```
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$=""   //EBC does not require initialisation vector
rc=AesSetKeyIv(0x100,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nEBC context created successfully"
ELSE
  PRINT "\nFailed to create EBC context"
ENDIF
inData$="30313233343536373839 3A3B3C3D3E3F"
inData$=StrDehexize$(inData$)
rc=AesEncrypt(inData$,outData$)
IF rc==0 THEN
  PRINT "\nEncrypt OK"
ELSE
  PRINT "\nFailed to encrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(inData$)
PRINT "\noutData = "; strhexize$(outData$)
```

Expected Output:

```
EBC context created successfully
Encrypt OK
inData  = 30313233343536373839A3B3C3D3E3F
outData = 03F2C3BDCA826BF082D7CFB035CDB8C1
```

AESENCRYPT is a core language function.

AesDecrypt

FUNCTION

This function is used to decrypt a string of exactly 16 bytes using the context that was precreated using the most recent call of the function AesSetKeyIv.

For all modes, AesSetKeyIV is called only once which means in CBC mode the cyclic data is kept in the context object that was created by AesSetKeyIV.

In terms of speed of exscution, for example on the BL600, which does **not** have AES 128 decryption hardware assist, the function has been timed to take roughly 570 microseconds.

AESDECRYPT (inData$,outData$)

Returns: INTEGER

Will be 0x0000 if the data was decrypted successfully. Otherwise an appropriate resultcode will be returned which will convey the reason it failed. ALWAYS check this.

Arguments:

***inData$*** **BYREF *inData$* AS STRING**
This string MUST be eactly 16 bytes long and should contain the data to decrypt

***outData$*** **BYREF *outData$* AS STRING**
On exit, if the function was successful, then this string will contain the decrypted plaintext data. If unsuccessful, then string will be 0 bytes long.

Interactive Command: NO

```
//Example :: AesDecrypt.sb (See in Firmware Zip file)
DIM key$, initVector$
DIM inData$, outData$, c$[3]
DIM rc
//Create context for CBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$="\FF\01\FF\03\FF\05\FF\07\FF\09\FF\0B\FF\0D\FF\0F"
rc=AesSetKeyIv(0x200,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nCBC context created successfully"
ELSE
  PRINT "\nFailed to create EBC context"
ENDIF
//encrypt some data
inData$="303132333435363738393A3B3C3D3E3F"
inData$=StrDehexize$(inData$)
rc=AesEncrypt(inData$,c$[0])
IF rc==0 THEN
  PRINT "\nEncrypt OK"
ELSE
  PRINT "\nFailed to encrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(inData$)
PRINT "\noutData = "; strhexize$(c$[0])
//encrypt same data again
rc=AesEncrypt(inData$,c$[1])
IF rc==0 THEN
  PRINT "\nEncrypt OK"
ELSE
  PRINT "\nFailed to encrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(inData$)
PRINT "\noutData = "; strhexize$(c$[1])
//ecrypt same data again
rc=AesEncrypt(inData$,c$[2])
IF rc==0 THEN
  PRINT "\nEncrypt OK"
ELSE
  PRINT "\nFailed to encrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(inData$)
PRINT "\noutData = "; strhexize$(c$[2])
//Rereate context for CBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$="\FF\01\FF\03\FF\05\FF\07\FF\09\FF\0B\FF\0D\FF\0F"
rc=AesSetKeyIv(0x200,16,key$,initVector$)
IF rc==0 THEN
```

```
  PRINT "\nCBC context created successfully"
ELSE
  PRINT "\nFailed to create EBC context"
ENDIF
//now decrypt the data
rc=AesDecrypt(c$[0],outData$)
IF rc==0 THEN
  PRINT "\n**Decrypt OK**"
ELSE
  PRINT "\nFailed to decrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(c$[0])
PRINT "\noutData = "; strhexize$(outData$)
//now decrypt the data
rc=AesDecrypt(c$[1],outData$)
IF rc==0 THEN
  PRINT "\n**Decrypt OK**"
ELSE
  PRINT "\nFailed to decrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(c$[1])
PRINT "\noutData = "; strhexize$(outData$)
//now decrypt the data
rc=AesDecrypt(c$[2],outData$)
IF rc==0 THEN
  PRINT "\n**Decrypt OK**"
ELSE
  PRINT "\nFailed to decrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(c$[2])
PRINT "\noutData = "; strhexize$(outData$)
```

Expected Output:

```
CBC context created successfully
**Decrypt OK**
inData  = 55EAFC8281CC28054C4AA268763AFA3B
outData = 303132333435363738393A3B3C3D3E3F
**Decrypt OK**
inData  = 2A8640BD480E092B432139CF28BA2C80
outData = 303132333435363738393A3B3C3D3E3F
**Decrypt OK**
inData  = A418B500A3E0AC30F18DE6AE2E923314
outData = 303132333435363738393A3B3C3D3E3F
```

AESDECRYPT is a core language function.

File I/O Functions

A portion of module's flash memory is dedicated to a file system which is used to store smartBASIC applications and user data files.

Due to the internal requirement, set by the smartBASIC runtime engine (because applications are interpreted in-situ), compiled application files have to be stored entirely in one continguous memory block. This means the file system is currently restricted so that it is NOT possible for an application to open a file and then write to it. To store application data so that they are non-volatile, use the functions described in the section "Non-Volatile Memory Management Routines"

This means any and all user data files need to be preloaded using the commands:-

AT+FOW
AT+FWR or AT+FWRH
AT+FCL

which are described in the section "Interactive Mode Commands".

The utility UwTerminal helps with downloading such files, but not strictly required.

This section describes all the functions that are available to an application to interact with data files in read mode.

With the use of READ, FTELL and FSEEK downloading configuration files (like say digital certificates) can be a very useful and convenient way of making an app behave in custom manner from data derived from these data files as demonstrated by the example application listed in the description of FOPEN.

FOPEN

FUNCTION

This function is used to open a file in mode specified by the 'mode$' string parameter. When the file is opened the file pointer is set to 0 which effectively means that a read operation will happen from the beginning of the file and then after the read the file pointer will be adjusted to offset equal to the size of the read.

Function FSEEK is provided to move that file pointer to an offset relative to the beginning, or current position or from the end of the file and function FTELL is provided to obtain the current position as an offset from the beginning of the file.

FOPEN (filename$, mode$)

Returns: INTEGER

A non-zero integer representing an opaque handle to the file that was opened. If the file failed to open, like for example because the mode specified writing to the file which is not allowed on certain platforms, then the returned value will be 0.

Arguments:

***filename$*** **BYREF filename$ AS STRING**
This string specifies the name of the file to open.

***mode$*** **BYVAL mode$ AS STRING**
Must always be set to "r"
This string specifies the mode the file should be opened, and for this module, as only reading is allowed must always be specified as "r".

Interactive Command: NO

```
//Example :: FileIo.sb (See in Firmware Zip file)
//
// First download a file into the module by submitting the following
// commands manually (wait for a 00 response after each command) :-
//
//     at+fow "myfile.dat"
//     at+fwr "Hello"
//     at+fwr " World. "
//     at+fwr " This is something"
//     at+fwr " in a file which we can read"
//     at+fcl
//
// You can check you have the file in the file system by submitting
// the command AT+DIR and you should see myfile.dat listed
//
DIM handle,fname$,flen,frlen,data$,fpos,rc

fname$="myfile.dat" : handle = fopen(fname$,"r")
IF handle != 0 THEN
  //determine the size of the file
  flen = filelen(handle)
  print "\nThe file is ";flen;" bytes long"
  //get the current position in the file (should be 0)
  rc = ftell(handle,fpos)
  print "\nCurrent position is ";fpos
  //read the first 11 bytes from the file
  frlen = fread(handle,data$,11)
  print "\nData from file is : ";data$
  //get the current position in the file (should be 11)
  rc = ftell(handle,fpos)
  print "\nCurrent position is ";fpos
  //reposition the file pointer to 6 so that we can read 5 bytes again
  rc = fseek(handle,6,0)
  //get the current position in the file
  rc = ftell(handle,fpos)
  //read 5 bytes
  frlen = fread(handle,data$,5)
  print "\nData from file is : ";data$
  //reposition to the start of 'is'
  rc = fseek(handle,19,0)
  //read until a 'w' is encountered : w = ascii 0x77
```

```
  frlen = freaduntil(handle,data$,0x77,32)
  print "\nData from file is : ";data$
  //finally close the file, which on exit will set the handle to 0
  fclose(handle)
ELSE
  print "\nFailed to open file ";fname$
ENDIF
```

```
The file is 59 bytes long
Current position is 0
Data from file is : Hello World
Current position is 11
Data from file is : World
Data from file is : is something in a file w
```

FOPEN is a core language function.

FCLOSE

FUNCTION

This function is used to close a file previously opened with FOPEN. It takes a handle parameter as a reference and will on exit set that handle to 0 which signifies an invalid file handle.

FCLOSE (fileHandle)

Returns: N/A as it is a subroutine

Arguments:

fileHandle **BYREF fileHandle AS INTEGER**
The handle of the file to be closed. On exit it will be set to 0

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FCLOSE is a core language function.

FREAD

FUNCTION

This function is used to read X bytes of data from a file previously opened with FOPEN and will return the actual number of bytes read.

FREAD (fileHandle, data$, maxReadLen)

Returns: INTEGER

The actual number of bytes read from the file. Will be 0 if read from end of file is attempted.

Arguments:

fileHandle **BYVAL fileHandle AS INTEGER**
The handle of the file to be read from

***data$*** **BYREF data$ AS STRING**
The data read from file is returned in this string

maxReadLen **BYVAL maxReadLen AS INTEGER**
The max number of bytes to read from the file

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FREAD is a core language function.

FREADUNTIL

FUNCTION

This function is used to read X bytes or until (and including) a match byte is encountered, whichever comes earlier, from a file previously opened with FOPEN and will return the actual number of bytes read (includes the match byte if encountered).

FREADUNTIL (fileHandle, data$, matchByte, maxReadLen)

Returns: INTEGER

The actual number of bytes read from the file. Will be 0 if read from end of file is attempted.

Arguments:

fileHandle **BYVAL fileHandle AS INTEGER**
The handle of the file to be read from

***data$*** **BYREF data$ AS STRING**
The data read from file is returned in this string

matchByte **BYVAL matchByte AS INTEGER**
Read until this matching byte is encountered or the max number of bytes are read. Whichever condition is asserted first.

maxReadLen **BYVAL maxReadLen AS INTEGER**
The max number of bytes to read from the file

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FREADUNTIL is a core language function.

FILELEN

FUNCTION

This function is used determine the total size of the file in bytes.

FILELEN (fileHandle)

Returns: INTEGER

The total number of bytes read from the file specified by the handle. Will be 0 if an invalid handle is supplied.

Arguments:

fileHandle **BYVAL fileHandle AS INTEGER**
The handle of a file for which the total size is to be returned.

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FILELEN is a core language function.

FTELL

FUNCTION

This function is used determine the current file position in the open file specified by the handle. It will be a value from 0 to N where N is the size of the file.

FTELL (fileHandle, curPosition)

Returns: INTEGER

The total number of bytes read from the file specified by the handle. Will be 0 if an invalid handle is supplied.

Arguments:

fileHandle **BYVAL fileHandle AS INTEGER**
The handle of a file for which the total size is to be returned.

curPosition **BYREF curPosition AS INTEGER**
This will be updated with the current file position for the file specified by the fileHandle.

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FTELL is a core language function.

FSEEK

FUNCTION

This function is used to move the file pointer of the open file specified by the handle supplied. The offset is relative to the beginning of the file or the current position or the end of the file which is specified by the 'whence' parameter.

FSEEK (fileHandle, offset, whence)

Returns: INTEGER

Will be 0 if successful

Arguments:

fileHandle **BYVAL fileHandle AS INTEGER**
The handle of a file for which the file pointer is to be moved

offset **BYVAL offset AS INTEGER**
This is the offset relative to the position defined by the 'whence' parameter.

whence **BYVAL whence AS INTEGER**
This parameter specifies from which position the offset is to be calculated. It shall be 1 to specify from the current position, 2 from the end of the while and then for all other values from the beginning of the file.
When the start position is 'end of file' then a positive 'offset' value is used to calculate backwards from the end of file. Hence supplying a negative value has no meaning.

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FSEEK is a core language function.

Non-Volatile Memory Management Routines

These commands provide access to the non-volatile memory of the module and provide the ability to use non-volatile storage for individual records.

NvRecordGet

FUNCTION

NVRECORDGET reads the value of a user record as a string from non-volatile memory.

NVRECORDGET (*recnum, strvar$*)

Returns: INTEGER, the number of bytes that were read into ***strvar$***. A negative value is returned if an error was encountered:

Error	Description
-1	***Recnum*** is not in valid range or is unrecognised.
-2	Failed to determine the size of the record.
-3	The raw record is less than 2 bytes long (possible flash corruption).
-4	Insufficient RAM.
-5	Failed to read the data record.

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

recnum ***byVal recnum AS INTEGER***
The record number to be read, in the range 1 to n, where n is the maximum number of records allowed by the specific module.

strvar$*** ***byRef strvar$ AS STRING
The string variable that will contain the data read from the record.

Interactive Command: NO

```
//Example :: NvRecordGet.sb (See in Firmware Zip file)
DIM r$
PRINT NvRecordGet(100,r$);" bytes read"
PRINT "\n";r$
```

Expected Output (When no data present in record):

```
0 bytes read
```

NVRECORDGET is a module function.

NvRecordGetEx

FUNCTION

NVRECORDGETX reads the value of a user record as a string from non-volatile memory and if it does not exist or an error occurred, then the specified default string is returned.

NVRECORDGETEX (*recnum, strvar$, strdef*)

Returns: INTEGER, the number of bytes that are read into ***strvar$***.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Out of Memory

Arguments:

recnum ***byVal recnum AS INTEGER***
The record number that is to be read, in the range *1* to *n*, where n is the maximum number of records allowed by the specific module.

strvar$*** ***byRef strvar$ AS STRING
The string variable that will contain the data read from the record.

strdef$*** ***byVal strdef$ AS STRING
The string variable that will supply the default data if the record does not exist.

Interactive Command: NO

```
//Example :: NvRecordGetEx.sb (See in Firmware Zip file)
DIM r$
PRINT NvRecordGetEx(100,r$,"default");" bytes read"
PRINT "\n";r$
```

Expected Output:

```
7 bytes read
default
```

NVRECORDGETEX is a module function.

NvRecordSet

FUNCTION

NVRECORDSET writes a value to a user record in non-volatile memory. For each record saved, an extra 28 bytes is used as an overhead, so it is recommended to minimise the writing of small records.

NVRECORDSET (*recnum, strvar$*)

Returns: INTEGER Returns the number of bytes written.

If an invalid record number is specified then -1 is returned. There are a limited number of user records which can be written to, depending on the specific module.

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

recnum ***byVal recnum AS INTEGER***
The record number that is to be read, in the range *1* to *n*, where n depends on the specific module.

strvar$*** ***byRef strvar$ AS STRING
The string variable that will contain the data to be written to the record.

WARNING: You should minimise the number of writes. Each time a record is changed, empty flash is used up. The flash filing system does not overwrite previously used locations. Eventually there will be no more free memory and an automatic defragmentation will occur. This operation takes much longer than normal as a lot of data may need to be re-written to a new flash segment. This sector erase operation could affect the operation of the radio and result in a connection loss.

Interactive Command: NO

```
//Example :: NvRecordSet.sb (See in Firmware Zip file)
DIM w$, r$, rc : w$ = "HelloWorld"
PRINT NvRecordSet(500,w$);" bytes written\n"
PRINT NvRecordGetEx(500,r$,"default");" bytes read\n"
PRINT "\n";r$
```

Expected Output:

```
10 bytes written
10 bytes read

HelloWorld
```

NVRECORDSET is a module function.

NvCfgKeyGet

FUNCTION

NVCFGKEYGET reads the value of a built-in configuration key. See AT+CFG for a list of configuration keys.

NVCFGKEYGET (keyId, *value*)

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

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

keyId ***byVal keyId AS INTEGER***
The configuration key that is to be read, in the range 1 to n, where n depends on the specific module and the full list is described for the AT+CFG command.

value ***byRef value AS INTEGER***
The integer variable that will be updated with the value of the configuration key if it exists.

Interactive Command: see AT+CFG

```
//Example :: NvCfgKeyGet.sb (See in Firmware Zip file)
DIM v : v = 0                    //initial the value just in case the key does not
exist
PRINT NvCfgKeyGet(100,v)
PRINT "\n";v
```

Expected Output:

```
0
33031
```

NVCFGKEYGET is a module function.

NvCfgKeySet

FUNCTION

NVCFGKEYSET writes a value to a pre-existing configuration key. See AT+CFG for a complete list of configuration keys. If a key does not exist, calling this function will not create a new one. The set of configuration keys are created at firmware build time. If you wish to create a database of non-volatile configuration keys for your own application use the NvRecordSet/Get() commands.

NVCFGKEYSET (keyId, *value*)

Returns: INTEGER

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

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

keyId ***byVal keyId AS INTEGER***
The configuration key that is to be read, in the range 1 to n, where n depends on the specific module and the full list is described for the AT+CFG command.

value ***byVal value AS INTEGER***
If the configuration key 'keyId' exists then it is updated with the new value.

WARNING: You should minimise the number of writes, as each time a record is changed, empty flash is used up. The flash filing system does not overwrite previously used locations. At some point there will be no more free memory and an automatic defragmentation will occur. This operation takes much longer than normal as a lot of data may need to be re-written to a new flash segment. This sector erase operation could affect the operation of the radio and result in a connection loss.

Interactive Command: NO

```
//Example :: NvCfgKeyGet.sb (See in Firmware Zip file)
DIM rc, r, w : w=0x8107
PRINT "\n";NvCfgKeySet(100,w)
PRINT "\n";NvCfgKeyGet(100,r)
PRINT "\nValue for 100 is ";r
```

Expected Output:

```
0
0
Value for 100 is 33031
```

NVCFGKEYSET is a module function.

Input/Output Interface Routines

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

GpioSetFunc

FUNCTION

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

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

GPIOSETFUNC (*nSigNum, nFunction, nSubFunc*)

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

See module specific user manual for details.

GpioConfigPwm

FUNCTION

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

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

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

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

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

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

On exit the function will return with the actual frequency in the nMinFreqHz parameter.

GPIOCONFIGPWM (*nMinFreqHz, nMaxPeriodus*)

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

See module specific user manual for details.

GpioRead

FUNCTION

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

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

GPIOREAD (*nSigNum)*

Returns: INTEGER, the value from the signal. If the signal number is invalid, then it will return value 0. For digital pins, the value will be 0 or 1. For ADC pins it will be a value in the range 0 to M where M is the max value based on the bit resolution of the analogue to digital converter.

See module specific user manual for details.

GpioWrite

SUBROUTINE

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

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

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

GPIOWRITE (*nSigNum, nNewValue)*

See module specific user manual for details.

GPIO Events

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

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

GpioBindEvent/GpioAssignEvent

FUNCTION

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

GPIOBINDEVENT (*nEventNum, nSigNum, nPolarity)*
GPIOASSIGNEVENT (*nEventNum, nSigNum, nPolarity)*

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

See module specific user manual for details.

Generally BindEvent will consume more power than the AssignEvent function and the choice as to which is used is based on the specific use case with regards to how much power can be used.

GpioUnbindEvent/ GpioAssignEvent

FUNCTION

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

GPIOUNBINDEVENT (*nEventNum*)
GPIOUNASSIGNEVENT (*nEventNum*)

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

See module specific user manual for details.

User Routines

As well as providing a comprehensive range of built-in functions and subroutines, *smart* BASIC provides the ability for users to write their own, which are referred to as 'user' routines as opposed to 'built-in' routines.

These are often used to perform frequently repeated tasks in an application and to write event and message handler functions. An application with user routines is highly modular, allowing reusable functionality.

SUB

A subroutine is a block of statements which constitute a user routine which does not return a value but takes arguments.

SUB routinename (arglist)
EXITSUB
ENDSUB

A SUB routine MUST be defined before the first instance of it being called. It is good practice to define SUB routines and functions at the beginning of an application, immediately after global variable declarations.

A typical example of a subroutine block would be

```
SUB somename(arg1 AS INTEGER arg2 AS STRING)
  DIM S AS INTEGER
  S = arg1
  IF arg1 == 0 THEN
    EXITSUB
  ENDIF
ENDSUB
```

Defining the routine name

The function name can be any valid name that is not already in use as a routine or global variable.

Defining the arglist

The arguments of the subroutine may be any valid variable types, i.e. INTEGER or STRING.

Each argument can be individually specified to be passed either as byVal or byRef. By default, simple variables (INTEGER) are passed by value (byVal) and complex variables (STRING) are passed by reference (byRef).

However, this default behaviour can be varied by using the #SET directive during compilation of an application.

```
#SET 1,0   'Default Simple arguments are BYVAL
#SET 1,1   'Default Simple arguments are BYREF
#SET 2,0   'Default Complex arguments are BYVAL
#SET 2,1   'Default Complex arguments are BYREF
```

When a value is passed by value to a routine, any modifications to that variable will not reflect back to the calling routine. However, if a variable is passed by reference then any changes in the variable will be reflected back to the caller on exit.

The SUB statement marks the beginning of a block of statements which will consist of the body of a user routine. The end of the routine is marked by the ENDSUB statement.

ENDSUB

This statement ends a block of statements belonging to a subroutine. It MUST be included as the last statement of a SUB routine, as it instructs the compiler that there is no more code for the SUB routine. Note that any variables declared within the subroutine lose their scope once ENDSUB is processed.

EXITSUB

This statement provides an early **run-time** exit from the subroutine.

FUNCTION

A statement beginning with this token marks the beginning of a block of statements which will consist of the body of a user routine. The end of the routine is marked by the ENDFUNC statement.

A function is a block of statements which constitute a user routine that returns a value. A function takes arguments, and can return a value of type simple or complex.

FUNCTION routinename (arglist) AS vartype
EXITFUNC arithemetic_expression_or_string_expression
ENDFUNC arithemetic_expression_or_string_expression

A function MUST be defined before the first instance of its being called. It is good practice to define subroutines and functions at the beginning of an application, immediately after variable declarations. A typical example of a function block would be:

```
FUNCTION somename(arg1 AS INTEGER arg2 AS STRING) AS INTEGER
  DIM S AS INTEGER
  S = arg1
  IF arg1 == 0 THEN
```

```
    EXITFUNC arg1*2
  ENDIF
ENDFUNC arg1 * 4
```

Defining the routine name

The function name can be any valid name that is not already in use. The return variable is always passed as byVal and shall be of type **varType**.

Return values are defined within zero or more optional EXITFUNC statements and ENDFUNC is used to mark the end of the block of statements belonging to the function.

Defining the return value

The variable type **AS varType** for the function may be explicitly stated as one of INTEGER or STRING prior to the routine name. If it is omitted, then the type is derived in the same manner as in the DIM statement for declaring variables. Hence, if function name ends with the $ character then the type will be a STRING. Otherwise, it is an INTEGER.

Since functions return a value, when used, they must appear on the right hand side of an expression statement or within a *[ ]* index for a variable. This is because the value has to be 'used up' so that the underlying expression evaluation stack does not have 'orphaned' values left on it.

Defining the arglist

The arguments of the function may be any valid variable type, i.e. INTEGER or STRING.

Each argument can be individually specified to be passed either as byVal or byRef. By default, simple variables (INTEGER) are passed byVal and complex variables (STRING) are passed byRef. However, this default behaviour can be varied by using the #SET directive.

```
# SET 1,0   Default Simple arguments are BYVAL
# SET 1,1   Default Simple arguments are BYREF
# SET 2,0   Default Complex arguments are BYVAL
# SET 2,1   Default Complex arguments are BYREF
```

Interactive Command: NO

ENDFUNC

This statement marks the end of a function declaration. Every function must include an ENDFUNC statement, as it instructs the compiler that here is no more code for the routine.

ENDFUNC arithemetic_expression_or_string_expression

This statement marks the end of a block of statements belonging to a function. It also marks the end of scope on any variables declared within that block.

ENDFUNC must be used to provide a return value, through the use of a simple or complex expression.

```
FUNCTION doThis$( byRef s$ as string) AS STRING
  S$=S$+" World"
ENDFUNC S$ + "world"

FUNCTION doThis( byRef v as integer) AS INTEGER
  v=v+100
ENDFUNC v * 3
```

EXITFUNC

This statement provides a run-time exit point for a function before reaching the ENDFUNC statement.

EXITFUNC arithemetic_expression or string expression

EXITFUNC can be used to provide a return value, through the use of a simple or complex expression. It is usually invoked in a conditional statement to facilitate an early exit from the function.

```
FUNCTION doThis$( byRef s$ as string) AS STRING
  S$=S$+" World"
  IF a==0 THEN
    EXITFUNC S$ + "earth"
  ENDIF
ENDFUNC S$ + "world"
```

6. Other Extension Built-in Routines

This chapter describes non BLE-related extension routines that are not part of the core *smart* BASIC language.

System Configuration Routines

SystemStateSet

FUNCTION

This function is used to alter the power state of the module as per the input parameter.

SYSTEMSTATESET (nNewState)

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

Arguments:

nNewState **byVal *nNewState* AS INTEGER**
New state of the module as follows:
0 System OFF (Deep Sleep Mode)

Note: You may also enter this state when UART is open and a BREAK condition is asserted. Deasserting BREAK makes the module resume through reset i.e. power cycle.

Interactive Command: NO

```
//Example :: SystemStateSet.sb (See in Firmware Zip file)

//Put the module into deep sleep
PRINT "\n"; SystemStateSet(0)
```

SYSTEMSTATESET is an extension function.

Miscellaneous Routines

ReadPwrSupplyMv

FUNCTION

This function is used to read the power supply voltage and the value will be returned in millivolts.

READPWRSUPPLYMV ()

Returns: INTEGER, the power supply voltage in millivolts.

Arguments: None

Interactive Command: NO

```
//Example :: ReadPwrSupplyMv.sb (See in Firmware Zip file)

//read and print the supply voltage
PRINT "\nSupply voltage is "; ReadPwrSupplyMv();"mV"
```

Expected Output:

```
Supply voltage is 3343mV
```

READPWRSUPPLYMV is an extension function.

SetPwrSupplyThreshMv

FUNCTION

This function sets a supply voltage threshold. If the supply voltage drops below this then the BLE_EVMSG event is thrown into the run time engine with a MSG ID of BLE_EVBLEMSGID_POWER_FAILURE_WARNING (19) and the context data will be the current voltage in millivolts.

Events & Messages

MsgId	Description
19	The supply voltage has dropped below the value specified as the argument to this function in the most recent call. The context data is the current reading of the supply voltage in millivolts

SETPWRSUPPLYTHRESHMV(nThresh)

Returns: INTEGER, 0 if the threshold is successfully set, 0x6605 if the value cannot be implemented.

Arguments: None

nThreshMv **byVal *nThresMv* AS INTEGER**
The BLE_EVMSG event is thrown to the engine if the supply voltage drops below this value. Valid values are 2100, 2300, 2500 and 2700.

Interactive Command: NO

```
//Example :: SetPwrSupplyThreshMv.sb (See in Firmware Zip file)

DIM rc
DIM mv

//============================================================================
// Handler for generic BLE messages
//============================================================================
FUNCTION HandlerBleMsg(BYVAL nMsgId, BYVAL nCtx) AS INTEGER
    SELECT nMsgId
        CASE 19
            PRINT "\n --- Power Fail Warning ",nCtx
            //mv=ReadPwrSupplyMv()
            PRINT "\n --- Supply voltage is "; ReadPwrSupplyMv();"mV"
        CASE ELSE
            //ignore this message
     ENDSELECT
ENDFUNC 1


//============================================================================
// Handler to service button 0 pressed
//============================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
    //just exit and stop waiting for events
ENDFUNC 0


ONEVENT  EVBLEMSG   CALL HandlerBleMsg
ONEVENT  EVGPIOCHAN1 CALL HndlrBtn0Pr

rc=GpioBindEvent(1,16,1)     //Channel 1, bind to low transition on GPIO pin 16
PRINT "\nSupply voltage is "; ReadPwrSupplyMv();"mV\n"
mv=2700
rc=SetPwrSupplyThreshMv(mv)

PRINT "\nWaiting for power supply to fall below ";mv;"mV"

//wait for events and messages
WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Supply voltage is 3343mV

Waiting for power supply to fall below 2700mV
Exiting...
```

SETPWRSUPPLYTHRESHMV is an extension function.

7. Events & Messages

smart BASIC is designed to be event driven, which makes it suitable for embedded platforms where it is normal to wait for something to happen and then respond.

To ensure that access to variables and resources ends up in race conditions, the event handling is done synchronously, meaning the *smart* BASIC runtime engine has to process a WAITEVENT statement for any events or messages to be processed. This guarantees that *smart* BASIC will never need the complexity of locking variables and objects.

There are many subsystems which generate events and messages as follows:-

- Timer events, which generate timer expiry events and are described here.
- Messages thrown from within the user's BASIC application as described here.
- Events related to the UART interface as described here.

8. Module Configuration

There are many features of the module that cannot be modified programmatically which relate to interactive mode operation or alter the behaviour of the smartBASIC runtime engine. These configuration objects are stored in non-volatile flash and are retained until the flash file system is erased via AT&F* or AT&F 1.

To write to these objects, which are identified by a positive integer number, the module must be in interactive mode and the command AT+CFG must be used which is described in detail here.

To read current values of these objects use the command AT+CFG, described here.

Predefined configuration objects are as listed under details of the AT+CFG command.

9. ACKNOWLEDGEMENTS

The following are required acknowledgements to address our use of open source code in smartBASIC to implement AES encryption.

Laird's implementation includes the following files: **aes.c** and **aes.h**.

LICENSE TERMS

The redistribution and use of this software (with or without changes) is allowed without the payment of fees or royalties providing the following:

- Source code distributions include the above copyright notice, this list of conditions and the following disclaimer;
- Binary distributions include the above copyright notice, this list of conditions and the following disclaimer in their documentation;
- The name of the copyright holder is not used to endorse products built using this software without specific written permission.

DISCLAIMER

This software is provided 'as is' with no explicit or implied warranties in respect of its properties, including, but not limited to, correctness and/or fitness for purpose.

Issue 09/09/2006

This is an AES implementation that uses only 8-bit byte operations on the cipher state (there are options to use 32-bit types if available).

The combination of mix columns and byte substitution used here is based on that developed by Karl Malbrain. His contribution is acknowledged.

INDEX