This guide pertains to WB45-specific smartBASIC routines and functions. For information on functions and routines that apply to all smartBASIC modules, see the smartBASIC Core Manual.
WB45 smartBASIC Extensions
User Guide

© 2015 Laird Technologies
All Rights Reserved. No part of this document may be photocopied, reproduced, stored in a retrieval system, or transmitted, in any form or by any means whether, electronic, mechanical, or otherwise without the prior written permission of Laird Technologies.

No warranty of accuracy is given concerning the contents of the information contained in this publication. To the extent permitted by law no liability (including liability to any person by reason of negligence) will be accepted by Laird Technologies, its subsidiaries or employees for any direct or indirect loss or damage caused by omissions from or inaccuracies in this document.

Laird Technologies reserves the right to change details in this publication without notice.

Windows is a trademark and Microsoft, MS-DOS, and Windows NT are registered trademarks of Microsoft Corporation. BLUETOOTH is a trademark owned by Bluetooth SIG, Inc., U.S.A. and licensed to Laird Technologies and its subsidiaries.

Other product and company names herein may be the trademarks of their respective owners.

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

<table>
<thead>
<tr>
<th>Version</th>
<th>Revisions Date</th>
<th>Change History</th>
<th>Approved By</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1.4.21</td>
<td>10 Nov 2015</td>
<td>Initial Release</td>
<td>Steve DeRosier</td>
</tr>
</tbody>
</table>
CONTENTS

Revision History ................................................................. 3
Contents .................................................................................. 4
1. Introduction ........................................................................ 5
   What is smartBASIC? .......................................................... 5
2. Command Line Options ..................................................... 6
3. Interactive Mode Commands ............................................... 7
4. Core Language Built-in Routines ........................................ 10
   Information Routines ......................................................... 11
   Uart Interface ...................................................................... 13
   Miscellaneous Routines ..................................................... 14
5. BTC Extensions Built-in Routines ....................................... 16
   Inquiries ............................................................................. 16
   Serial Port Profile ............................................................ 23
   Stream Functions ................................................................ 32
   Pairing/Bonding Functions ............................................... 34
   Miscellaneous Functions .................................................... 46
6. BLE Extensions Built-in Routines ....................................... 52
   Bluetooth Address ............................................................ 52
   Events and Messages ........................................................ 53
   Miscellaneous Functions .................................................... 68
   Advertising Functions ....................................................... 72
   Scanning Functions .......................................................... 81
   Connection Functions ....................................................... 93
   Security Manager Functions .............................................. 103
   GATT Server Functions ..................................................... 114
   GATT Client Functions ..................................................... 149
   Attribute Encoding Functions .......................................... 189
   Attribute Decoding Functions ............................................ 199
   Pairing/Bonding Functions ................................................. 213
7. SOCKET Extensions Built-in Routines ................................. 218
   Socket Functions .............................................................. 218
8. Events and Messages ........................................................ 227
9. Miscellaneous ................................................................. 227
10. Acknowledgements .......................................................... 229
   Index ................................................................................. 230
1. INTRODUCTION

This user guide provides detailed information on WB45 smartBASIC extensions. It provides a high-level managed interface to the underlying stack in order to manage the following:

- Bluetooth Classic (BTC) Inquiries, discovery, connections
- Serial Port Profile (SPP)
- BLE advertisements and connections
- Bluetooth Low Energy (BLE) security and bonding
- GATT Table: Services, characteristics, descriptors, advert reports
- GATT server/client operation.
- Attribute encoding and decoding
- Socket IO functionality
- Events related to the above

What is smartBASIC?

smartBASIC is an event-driven programming language designed to make Bluetooth development quicker and simpler, vastly cutting down time to market. It is an implementation of a structured BASIC programming language optimized for use on embedded systems with limited memory by being highly efficient in terms of memory usage.

Being a structured programming language, smartBASIC offers typical modern constructs such as subroutines, functions, while, if, and for loops. The language also provides the standard functionality of any programming language such as arithmetic functions, binary operations, conditionals, string processing, arrays, and memory management. A smartBASIC application usually ends with WAITEVENT, a final statement which never returns. Once the run-time engine reaches the WAITEVENT statement, it waits for events to happen and, when they do, it calls the appropriate handlers (written by the user) to process them.

smartBASIC has two modes of operation: interactive mode and runtime mode. In interactive mode, commands are sent via the console and are executed immediately, analogous to the behaviour of a modem using AT commands. Interactive mode is primarily used for configuring the module and for compiling smartBASIC applications. In Run-time mode, the module runs pre-compiled smartBASIC applications from the host OS. All the Bluetooth and socket functionality can only be achieved from the runtime mode.

On the WB45, smartBASIC is used as the primary method for Bluetooth functionality. A simple smartBASIC program can be written to interface to the host OS using the socket API, and simultaneously communicate to wireless devices through Bluetooth and Bluetooth Low Energy (BLE).

To run smartBASIC, simply type:

```
#smartBASIC
```

**Note:** Please make sure that no other program is using smartBASIC before running it. If smartBASIC is already running as a daemon in the background it has to be terminated before launched again.
2. COMMAND LINE OPTIONS

smartBASIC can be run with one of the following command line options:

-a, --autorun

The autorun command line option is used to enable the autorun functionality of smartBASIC. If an `$autorun$` application exists in the filesystem, smartBASIC directly enters into runtime mode and the application is automatically launched. If the autorun command line option is not passed, then smartBASIC will enter interactive mode regardless of the existence of the autorun file.

```
#smartBASIC -a
#smartBASIC --autorun
```

-K, --eraseall

The eraseall command line option is used to erase the filesystem before entering immediate or runtime modes. The virtual filesystem will therefore be completely empty once smartBASIC fully launches.

```
#smartBASIC -K
#smartBASIC --eraseall
```

-E, --erase

The erase command line option is used to erase specific portions of the virtual file system. When smartBASIC is launched for the first time, ten flash binaries are created. These flash binaries can be deleted individually at startup using the erase command.

```
#smartBASIC -E 7
#smartBASIC --erase 7
```

-d, --daemon

The daemon command line argument allows smartBASIC to be launched in daemon mode. In this mode, smartBASIC will operate as a background process with no controlling terminal. Input/output to smartBASIC can no longer be done through stdin/stdout in this mode.

```
#smartBASIC -d
#smartBASIC --daemon
```

-c, --compile

The compile argument allows the user to compile smartBASIC applications from the command line before starting smartBASIC (as opposed to using the compile immediate command to compile applications after smartBASIC has started). If the compilation is successful, smartBASIC should be launched as usual with the compiled smartBASIC application present in the virtual filesystem.

```
#smartBASIC -c hello.world.sb
#smartBASIC --compile `$autorun$.hello.world.sb`
```

-x, --compileexit

The compileexit command line option is used to compile a smartBASIC application from the command line, but instead of the program running, it would simply exit upon the completion of the compilation process. This argument can be therefore used to compile multiple smartBASIC applications consecutively without running smartBASIC.

```
#smartBASIC -x hello.world.sb
#smartBASIC --compileexit `$autorun$.hello.world.sb`
```
-b, --btsnoop

The btsnoop command line option allows smartBASIC to save the raw HCI data to a log file in btsnoop format. This can be used for the purpose of HCI-level debugging. The log file is saved in the /tmp directory.

```
#smartBASIC -b
#smartBASIC --btsnoop
```

3. INTERACTIVE MODE COMMANDS

Below are some WB45-specific AT commands.

**AT+CFG COMMAND**

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

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 ?**

<table>
<thead>
<tr>
<th>Returns</th>
<th>If the config key is successfully updated or read, the response is \n\n\n.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| num | Integer Constant  
The ID of the required configuration key. All of the configuration keys are stored as an array of 16-bit words. |
| value | Integer_constant  
This is the new value for the configuration key and the syntax allows decimal, octal, hexadecimal, or binary values. |

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

The following Configuration Key IDs are defined.

<table>
<thead>
<tr>
<th>ID</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Maximum size of local simple variables</td>
</tr>
<tr>
<td>41</td>
<td>Maximum size of local complex variables</td>
</tr>
<tr>
<td>42</td>
<td>Maximum depth of nested user-defined functions and subroutines</td>
</tr>
<tr>
<td>43</td>
<td>The size of stack for storing user functions simple variables</td>
</tr>
<tr>
<td>44</td>
<td>The size of stack for storing user functions complex variables</td>
</tr>
<tr>
<td>45</td>
<td>The size of the message argument queue length</td>
</tr>
<tr>
<td>250</td>
<td>Deprecated, please refer to BtcSPPSetParams for alternative method.</td>
</tr>
</tbody>
</table>
ID | Definition  
---|-------------
251 | Deprecated, please refer to BtcSPPSetParams for alternative method.  
300 | Deprecated, please refer to BtcSPPSetParams for alternative method.  
301 | Deprecated, please refer to BtcSPPSetParams for alternative method.  

AT+CFG is a core command.

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

**AT+BTD**

**COMMAND**

Deletes the bonded device database from the flash.

**AT+BTD**

<table>
<thead>
<tr>
<th>Returns</th>
<th>\n00\r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

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

```
Examples:
AT+BTD*
```

**AT+BTD** is an extension command

**AT+BLX**

**COMMAND**

This command is used to stop all radio activity (adverts or connections) when in interactive mode.

**AT+BLX**

<table>
<thead>
<tr>
<th>Returns</th>
<th>\n00\r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

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

```
Examples:
AT+BLX
```
AT+BLX is an extension command.

AT&F

COMMAND

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

AT&F integermask

<table>
<thead>
<tr>
<th>Returns</th>
<th>OK if flash successfully erases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td>Integermask</td>
<td>Integer corresponding to a bit mask or the * character</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mask</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Erases normal file system and system config keys (see AT+CFG for examples of config keys)</td>
</tr>
<tr>
<td>0x40000</td>
<td>Erases the User config keys only</td>
</tr>
<tr>
<td>0x10000</td>
<td>Erase the BLE Bonding Manager</td>
</tr>
<tr>
<td>0x20000</td>
<td>Erases the Classic Bluetooth Bonding Manager</td>
</tr>
<tr>
<td>*</td>
<td>Erases all data segments</td>
</tr>
<tr>
<td>Else</td>
<td>Not applicable to current modules</td>
</tr>
</tbody>
</table>

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

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

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

AT&F is a core command.
COMPILE

COMMAND
Compile a smartBASIC application and load it into the virtual filesystem.

**COMPILE “AppName” “OutputFilename”**

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n00\r</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppName</td>
</tr>
<tr>
<td>OutputFilename</td>
</tr>
<tr>
<td>Interactive Command</td>
</tr>
</tbody>
</table>

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

```
'Examples:

compile "HelloWorld.sb" "hw1"
00
compile "../Hello.sb" "hw2"
00
compile "/tmp/hello.sb" "hw3"
00
```

QUIT

COMMAND
This command is used to quit smartBASIC.

**QUIT**

| Returns  | - |
|----------|
| Arguments | None |
| Interactive Command | Yes |

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

```
'Examples:

quit
#
```

QUIT is an extension command

4. **Core Language Built-in Routines**

Core language built-in routines are present in every implementation of smartBASIC. These routines provide the basic programming functionality. They are augmented with target-specific routines for different platforms which are described in the extension manual for each target platform.
All the core functionality is described in the document “smartBASIC Core Functionality.” Additional information is also available from our Laird Embedded Wireless Solutions Support Center at http://ews-support.lairdtech.com.

However some functions have small behavior differences; these are listed below.

**Information Routines**

**SYSINFO FUNCTION**

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

**SYSINFO(varId)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER. Value of information corresponding to integer ID requested.</th>
</tr>
</thead>
</table>
| Exceptions | ▪ Local Stack Frame Underflow  
▪ Local Stack Frame Overflow |

**Arguments:**

`byVal varId AS INTEGER`  
An integer ID which is used to determine which information is to be returned as described below.

<table>
<thead>
<tr>
<th>varId</th>
<th>Device ID. Each platform type has a unique identifier.</th>
</tr>
</thead>
</table>
| 0     | smartBASIC version number  
Example:  
X.Y.Z is returned as a 32-bit value made up as follows:  
(X<<26) + (Y<<20) + (Z) where Y is the build number and Z is the sub-build number |
| 3     | 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 |
| 1004  | Maximum STRING size |
| 1005  | Is 1 for run-time only implementation, 3 for compiler included |
| 1010  | Module Type |
| 2000  | Reset Reason  
▪ 8 : Self-Reset due to Flash Erase  
▪ 9 : ATZ  
▪ 10 : Self-Reset due to smart BASIC app invoking function RESET() |
<p>| 2001  | Cause of last reset |</p>
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Timer resolution in microseconds</td>
</tr>
<tr>
<td>2003</td>
<td>Number of timers available in a <em>smart</em> BASIC Application</td>
</tr>
<tr>
<td>2004</td>
<td>Tick timer resolution in microseconds</td>
</tr>
<tr>
<td>2005</td>
<td>LMP Version number for BT 4.0 spec</td>
</tr>
<tr>
<td>2006</td>
<td>LMP Sub Version number</td>
</tr>
<tr>
<td>2007</td>
<td>Chipset Company ID allocated by BT SIG</td>
</tr>
<tr>
<td>2008</td>
<td>Returns the current TX power setting (see also 2018)</td>
</tr>
<tr>
<td>2009</td>
<td>Number of devices in trusted device database</td>
</tr>
<tr>
<td>2010</td>
<td>Number of devices in trusted device database with IRK</td>
</tr>
<tr>
<td>2011</td>
<td>Number of devices in trusted device database with CSRK</td>
</tr>
<tr>
<td>2012</td>
<td>Max number of devices that can be stored in trusted device database</td>
</tr>
<tr>
<td>2013</td>
<td>Maximum length of a GATT Table attribute in this implementation</td>
</tr>
<tr>
<td>2014</td>
<td>Total number of transmission buffers for sending attribute NOTIFIES</td>
</tr>
<tr>
<td>2015</td>
<td>Number of transmission buffers for sending attribute NOTIFIES – free</td>
</tr>
<tr>
<td>2016</td>
<td>Radio activity of the baseband</td>
</tr>
<tr>
<td></td>
<td>0 : no activity</td>
</tr>
<tr>
<td></td>
<td>1 : advertising</td>
</tr>
<tr>
<td></td>
<td>2 : connected</td>
</tr>
<tr>
<td></td>
<td>3 : broadcasting and connected</td>
</tr>
<tr>
<td>2018</td>
<td>Returns the TX power while pairing in progress (see also 2008)</td>
</tr>
<tr>
<td>2019</td>
<td>Default ring buffer length for notify/indicates in gatt client manager (see BleGattcOpen function)</td>
</tr>
<tr>
<td>2020</td>
<td>Maximum ring buffer length for notify/indicates in gatt client manager (see BleGattcOpen function)</td>
</tr>
<tr>
<td>2040</td>
<td>Max number of devices that can be stored in trusted device database</td>
</tr>
<tr>
<td>2041</td>
<td>Number of devices in trusted device database</td>
</tr>
<tr>
<td>2042</td>
<td>Number of devices in the rolling device database</td>
</tr>
<tr>
<td>2043</td>
<td>Maximum number of devices that can be stored in the rolling device database</td>
</tr>
<tr>
<td>2100</td>
<td>Connect scan interval (ms)</td>
</tr>
<tr>
<td>2101</td>
<td>Connect scan window (ms)</td>
</tr>
<tr>
<td>2102</td>
<td>Connect slave latency (ms)</td>
</tr>
<tr>
<td>2105</td>
<td>Connect multi-link connection interval periodicity (ms)</td>
</tr>
<tr>
<td>2106</td>
<td>Minimum connection length (ms)</td>
</tr>
<tr>
<td>2107</td>
<td>Maximum connection length (ms)</td>
</tr>
<tr>
<td>2150</td>
<td>Scan interval (ms)</td>
</tr>
<tr>
<td>2151</td>
<td>Scan window (ms)</td>
</tr>
<tr>
<td>2152</td>
<td>Scan type</td>
</tr>
<tr>
<td></td>
<td>0 – Passive</td>
</tr>
<tr>
<td></td>
<td>1 – Active</td>
</tr>
<tr>
<td>2153</td>
<td>Minimum number of reports to store in cache</td>
</tr>
</tbody>
</table>

### Interactive Command

**No**

---

//Example :: SysInfo.sb
PRINT "\nSysInfo 601 = ";SYSINFO(601)  // Flash File System: Total Space (Data Segment)

Expected Output:

SysInfo 601 = 49152

SYSINFO is a core language function.

SYSINFO$(varId)

FUNCTION

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

SYSINFO$(varId)

<table>
<thead>
<tr>
<th>Returns</th>
<th>STRING. Value of information corresponding to integer ID requested.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions</td>
<td>• Local Stack Frame Underflow</td>
</tr>
<tr>
<td></td>
<td>• Local Stack Frame Overflow</td>
</tr>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>varId</td>
<td>byVal varId AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>An integer ID which is used to determine which information is to be returned as described below.</td>
</tr>
<tr>
<td></td>
<td>4 The Bluetooth address of the module.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Interactive</td>
<td>No</td>
</tr>
<tr>
<td>Command</td>
<td></td>
</tr>
</tbody>
</table>

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

Expected Output:

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

SYSINFO$ is a core language function.

Uart Interface

UartOpen

FUNCTION

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

See core manual for further details.
Note: Currently, UartOpen only opens the stdin/stdout file descriptors when called. All the parameters passed to the function are placeholders only. The actual parameters can be configured through the stty command-line tool outside the scope of smartBASIC.

**UARTOPEN (baudrate,txbuflen,rxbuflen,stOptions)**

```vbnet
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:

<table>
<thead>
<tr>
<th></th>
<th>DTE/DCE role request:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T – DTE</td>
</tr>
<tr>
<td></td>
<td>C – DCE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Parity:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N – None</td>
</tr>
<tr>
<td></td>
<td>O – Odd</td>
</tr>
<tr>
<td></td>
<td>E – Even</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Databits: 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Stopbits: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Flow Control:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>N – None</td>
</tr>
<tr>
<td></td>
<td>H – CTS/RTS hardware</td>
</tr>
<tr>
<td></td>
<td>X – Xon/Xof (Not Available)</td>
</tr>
</tbody>
</table>
```

**Miscellaneous Routines**

This section describes all miscellaneous functions and subroutines.

**ERASEFILESYSTEM**

**FUNCTION**

This function is used to erase the flash file system which contains the application that invoked this function. After erasing the file system, smartBASIC resets and reboots into command mode. This facility allows the current $autorun$ application to be replaced with a new one.

**ERASEFILESYSTEM (nArg)**

**Returns**

INTEGER Indicates success of command:

<table>
<thead>
<tr>
<th></th>
<th>Successful erasure. smartBASIC reboots.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<>0  Failure.

**Exceptions**

- Local Stack Frame Underflow
- Local Stack Frame Overflow

**Arguments:**

**nArg**

byVal nArg AS INTEGER
This is for future use and MUST always be set to 1. Any other value will
result in a failure.

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

Expected Output:

```
Failed to erase file system because incorrect parameter
00
```

ERASEFILESYSTEM is an extension function.
5. **BTC EXTENSIONS BUILT-IN Routines**

**Inquiries**

This section describes routines related to inquiries.

**Events and Messages**

*EVINQRESP*

This event is thrown when there is an BTC inquiry report waiting to be read. The message, which is passed to a handler which should be registered in the *smartBASIC* application, contains *respType*, the type of inquiry response received. It is one of the following values:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Standard</td>
</tr>
<tr>
<td>1</td>
<td>With RSSI</td>
</tr>
<tr>
<td>2</td>
<td>Extended (contains EIR data)</td>
</tr>
</tbody>
</table>

```basic
dim rc
dim adr$

adr$=""

//==============================================================================
// This handler is called when there is an inquiry report waiting to be read
// Algorithm will prevent display of data from the same peer consecutively
//==============================================================================
function HandlerInqResp(respType) as integer
    dim ad$, dta$, ndx, rsi, tag
    rc = BtcInquiryGetReport(ad$, dta$, ndx, rsi)
    //if Bluetooth address is different from the previous one
    if strcmp(ad$, ad$)! = 0 then
        print "Bluetooth: "; StrHexize$(ad$)
        if respType > 0 then
            print " "; rsi
        if respType == 2 then
            print "EIR: "; StrHexize$(dta$)
            dim tg$
            while BtcGetEIRbyIndex(ndx, dta$, tag, ad$)==0
                //write tag value as hex to string tg$
                sprint #tg$, integer.h'tag
            //hexize eir tag data if not a shortened or complete local name
            if tag < 0x08 || tag > 0x09 then
                ad$ = StrHexize$(ad$)
            else
                StrDeescape(ad$)
            endif
        endif
    //print the last 2 hex digits of the tag, and the data
    if strlen(ad$)! = 0 then
        print "Tag 0x" + RIGHT$(tg$, 2) +": "; ad$
    endif
```
ndx=ndx+1
endwhile
print "\n"
endif
endif
endfunc 1

function HandlerBtcInqTimOut() as integer
    print "\nScanning stopped via timeout"
endfunc 0

OnEvent EVINQRESP    call HandlerInqResp
OnEvent EVBTC_INQUIRY_TIMEOUT    call HandlerBtcInqTimOut

rc = BtcInquiryConfig(1,2)    //extended inquiry mode
rc = BtcInquiryStart(10)
WaitEvent

Expected Output:

<table>
<thead>
<tr>
<th>Bluetooth: 0C8FD515094 -57</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIR: 0D094C4F4E444C3139545B58525931020A0A</td>
</tr>
<tr>
<td>- Tag 0x09: LONDL19TXRY1</td>
</tr>
<tr>
<td>- Tag 0x0A: 0A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bluetooth: 94350AA99A3C -45</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIR: 140946176696420446176697327732050686F6E65170305110A11C111211511611F112D112F11001</td>
</tr>
<tr>
<td>- Tag 0x09: David Davis's Phone</td>
</tr>
<tr>
<td>- Tag 0x03: 05110A110C11121115116111F112D112F110012321</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bluetooth: B00594F52133 -63</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIR: 0D094C4F4E444C3139545B58525931020A00</td>
</tr>
<tr>
<td>- Tag 0x09: LONDL19TXRY1</td>
</tr>
<tr>
<td>- Tag 0x0A: 00</td>
</tr>
</tbody>
</table>

**EVBTC_INQUIRY_TIMEOUT**

This event is thrown when an inquiry times out. When an inquiry times out this doesn’t necessarily mean that there are no more responses waiting, so you can obtain the remaining responses after a timeout by calling `BtcInquiryGetReport()`.

See example for `EvInqResp`.

**BtcInquiryConfig**

**FUNCTION**

This function sets the parameters for all subsequent BTC inquiries which are started using the function `BtcInquiryConfig()`.

**Note:** Limited inquiry is not currently supported and will be implemented in future releases of the firmware.
BTCINQUIRYCONFIG (nConfigID,nValue)

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

Arguments:

<table>
<thead>
<tr>
<th>nConfigID</th>
<th>byVal nConfigID</th>
<th>AS INTEGER.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This identifies the value to update as follows:</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Inquiry Type (0 for General Inquiry, 1 for Limited Inquiry)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Inquiry Mode (0 for Standard, 1 for with RSSI, 2 for Extended)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Max number of inquiry responses to receive (Range is from 0-255)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Inquiry Tx Power (Range is from -70 to 20 dBm)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nValue</th>
<th>byVal nValue</th>
<th>AS INTEGER.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The new value to set for the parameter identified by configID.</td>
<td></td>
</tr>
</tbody>
</table>

See example for EvInqResp

BTCINQUIRYCONFIG is an extension function.

BtcInquiryStart

FUNCTION

Start inquiries with the parameters set using the function BtcInquiryConfig().

BTCINQUIRYSTART (nTimeout)

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

Arguments:

<table>
<thead>
<tr>
<th>nTimeout</th>
<th>byVal nTimeout</th>
<th>AS INTEGER.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is how long in seconds the inquiry lasts. If the timer times out then the event EVBTC_INQUIRY_TIMEOUT is thrown to the smartBASIC application.</td>
<td></td>
</tr>
</tbody>
</table>

See example for EvInqResp

BTCINQUIRYSTART is an extension function.

BtcInquiryCancel

FUNCTION

Cancel an ongoing inquiry.

BTCINQUIRYCANCEL()

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

Arguments: None

Interactive Command: No

```
dim rc
rc=BtcInquiryStart(10)
if rc == 0 then
    print "\nInquiry Started"
```
else
    print "\nError: ";rc
endif

TimerStart(0,2000,0)

Function TimerExpr()
    rc=BtcInquiryCancel()
    if rc == 0 then
        print "\nInquiry Cancelled"
    else
        print "\nError: ";rc
    endif
EndFunc 0

OnEvent EvTmr0 call TimerExpr
waitevent

Expected Output:

Inquiry Started
Inquiry Cancelled

BTCINQUIRYCANCEL is an extension function.

**BTCInquiryGetReport**

**FUNCTION**

When an inquiry is in progress (after having called BTCInquiryStart() for report), the information is cached in a queue buffer and a EVINQRESP event is thrown to the *smartBASIC* application.

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

**BTCINQUIRYGETREPORT (addr$, inqData$, nDiscarded, nRssi)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| addr$       | byREF periphAddr$ AS STRING
The address of the advertiser is returned in this string. It is a 6-byte string.|
| inqData$    | byREF advData$ AS STRING
The data payload is returned in this string.|
| nDiscarded  | byREF nDiscarded AS INTEGER
On return, this parameter is updated with the number of adverts that were discarded because there was no space in the internal queue.|
| nRssi       | byREF nRssi AS INTEGER
On return, this parameter is updated with the RSSI as reported by the stack for that advert.
**Note:** This is not a value that is sent by the peripheral but rather a value that is calculated by the receiver in this module.|
| Interactive Command | No |
See example for EvInqResp. BTCINQUIRYGETREPORT is an extension function.

**BtcGetEIRbyIndex**

**FUNCTION**

This function is used to extract the nth EIR element from the STRING data$. If the last EIR element is malformed, it is treated as non-existent.

**BTCGETEIRBYINDEX**(nIndex, data$, EIRtag, EIRval$)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nIndex</td>
<td>byVAL nIndex AS INTEGER. Extract the nth element from the advert report in data$. It is 0 based. Specifying a -ve or a value more than the number of EIR elements will result in an error</td>
</tr>
<tr>
<td>data$</td>
<td>byREF data$ AS STRING On exit this will contain the report containing concatenated EIR elements</td>
</tr>
<tr>
<td>EIRtag</td>
<td>byREF EIRtag AS INTEGER On exit this will contain the tag value</td>
</tr>
<tr>
<td>EIRval$</td>
<td>byREF EIRval$ AS STRING On exit this contains the data from the nth EIR element if it exists.</td>
</tr>
</tbody>
</table>

**Note:** Only the data portion of the EIR element is returned. The Tag is separately provided in the EIRtag argument and the length of the data is strlen(EIRval$).

**Interactive Command**

No

```plaintext
dim rc

dim adr$

adr$=""
```

```plaintext
// This handler is called when there is an inquiry report waiting to be read
// Algorithm will prevent display of data from the same peer consecutively
//===========================================
function HandlerInqResp(respType) as integer
    dim ad$, dta$, ndx, rsi, tag
    rc = BtcInquiryGetReport(ad$, dta$, ndx, rsi)

    //if Bluetooth address is different from the previous one
    if strcmp(adr$, ad$) != 0 then
        print "Bluetooth Address: "; StrHexize$(ad$)

    if respType > 0 then
        print " "; rsi

    if respType == 2 then
        print " EIR: "; StrHexize$(dta$)
        dim tg$
        while BtcGetEIRbyIndex(ndx, dta$, tag, ad$) == 0
            //write tag value as hex to string tg$
            sprint #tg$, integer.h"tag
```

Embedded Wireless Solutions Support Center: 20 Americas: +1-800-492-2320
http://ews-support.lairdtech.com Europe: +44-1628-858-940
www.lairdtech.com/wireless Hong Kong: +852 2923 0610
//hexize eir tag data if not a shortened or complete local name
if tag < 0x08 || tag > 0x09 then
    ad$ = StrHexize$(ad$)
else
    StrDeescape(ad$)
endif

//print the last 2 hex digits of the tag, and the data
if strlen(ad$)! = 0 then
    print "\n - Tag 0x" + RIGHT$(tg$,2) +": "; ad$
endif

ndx=ndx+1
endwhile
print "\n"
endif
endif
dendfunc

function HandlerBtcInqTimOut() as integer
print "\nScanning stopped via timeout"
endfunc

OnEvent EVINQRESP	call HandlerInqResp
OnEvent EVBTC_INQUIRY_TIMEOUT	call HandlerBtcInqTimOut

rc = BtcInquiryConfig(1,2) //extended inquiry mode
rc = BtcInquiryStart(10)
WaitEvent

Expected Output:

Bluetooth: 0C8BF515094 -57
  EIR: 0D094C4F4E444C31395458525931020A0A
  - Tag 0x09: LONDL19TXRY1
  - Tag 0x0A: 0A

Bluetooth: 94350A99A3C -45
  EIR: 140946176696420446176697327732050686F6E65170305110A110C112115116111F112D112F110012321101050107
  - Tag 0x09: David Davis's Phone
  - Tag 0x03: 05110A110C112115116111F112D112F1100123211

Bluetooth: B00594F52133 -63
  EIR: 0D094C4F4E444C43564B51525931020A00
  - Tag 0x09: LONDLCVKQRY1
  - Tag 0x0A: 00

BTCGETEIRBYINDEX is an extension function.

BtcGetEIRbyTag

FUNCTION
This function is used to extract the first instance of an EIR element from the STRING data$ identified by the tag EIRtag. Any malformed EIR elements are ignored.

**BTCGETEIRBYTAG (data$, EIRtag, EIRval$)**

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

**Arguments:**

- **data$** byREF data$ AS STRING
  - On exit this will contain the report containing concatenated EIR elements
- **EIRtag** byREF EIRtag AS INTEGER
  - The tag to look for. Only the first instance can be extracted. If multiple instances are suspected, then use BtcGetEIRbyIndex()
- **EIRval$** byREF EIRval$ AS STRING
  - On exit this contains the data from the nth EIR element if it exists.

**Note**: Only the data portion of the EIR element is returned. The tag is separately provided in the EIRtag argument and the length of the data is strlen(EIRval$).

**Interactive Command**: No

```plaintext
dim rc
dim adr$

adr$=""

//==============================================================================
// This handler is called when there is an inquiry report waiting to be read
// Algorithm will prevent display of data from the same peer consecutively
//==============================================================================
function HandlerInqRpt(cType) as integer
  dim ad$,dta$,ndx,rsi,tag
  rc = BtcInquiryGetReport(ad$,dta$,ndx,rsi)
  while rc==0
    if strcmp(adr$,ad$) != 0 then
      adr$=ad$
      print "$INQ:";strhexize$(ad$);" ";rsi
      if cType == 2 then
        // If its extended print the raw EIR data, then the complete local name
        print "$ EIR RAW:";strhexize$(dta$)
        print "$ EIR:"
        tag = 0x09 //complete local name
        rc=BtcGetEIRbyTag(dta$,tag,ad$)
      endif
      print "Complete Local Name: ";ad$
      print "Hex: ";strhexize$(ad$)
    endif
  endwhile
endfunc
```
WB45 smartBASIC Extensions
User Guide

function HandlerBtcInqTimOut() as integer
    print "\nScanning stopped via timeout"
endfunc 0

OnEvent EVINQRESP call HandlerInqRpt
OnEvent EVBTC_INQUIRY_TIMEOUT call HandlerBtcInqTimOut

rc = BtcInquiryConfig(1,2) //Mode with Extended
rc = BtcInquiryStart(5)
WaitEvent

Expected Output:

```
INQ:0016A4FEF009 -74
  EIR RAW:0A084C6169726420464546050301110012
  EIR:Complete Local Name: Hex:
INQ:0016A4093D92 -74
  EIR RAW:1409736D6172745A2D303031364134303933443932
  EIR:Complete Local Name: smartZ-0016A4093D92Hex:
    736D6172745A2D303031364134303933443932
INQ:0016A4093A89 -61
  EIR RAW:1409736D6172745A2D30303136413430393343839
  EIR:Complete Local Name: smartZ-0016A4093A89Hex:
    736D6172745A2D30303136413430393343839
INQ:C4D98776AE3E-65
  EIR RAW:0E094C4F4E444C48515356575A31020A04
  EIR:Complete Local Name: LONDLHQSVVWZ1Hex
```

BTCGETEIRBYTAG is an extension function.

Serial Port Profile

The SPP is for serial data transmission with a remote device in both directions. It behaves like a wireless replacement for a serial cable.

Events and Messages

**EVSPPCONN**

This event is thrown when a new SPP connection has been established or an error has occurred. The message is passed to a handler, which should be registered in the smartBASIC application, and contains `nHandle` (the handle of the connection) and `result` (a result code). `nHandle` is only valid on a successful result code (0).

Possible errors are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPP_CONNECTION_TIMEOUT</td>
<td>0x01</td>
</tr>
<tr>
<td>SPP_CONNECTION_REFUSED</td>
<td>0x02</td>
</tr>
<tr>
<td>SPP_UNKNOWN_ERROR</td>
<td>0x03</td>
</tr>
<tr>
<td>SDP_TIMEOUT</td>
<td>0x10</td>
</tr>
<tr>
<td>SDP_CONNECTION_ERROR</td>
<td>0x11</td>
</tr>
<tr>
<td>SDP_ERROR_RESPONSE</td>
<td>0x12</td>
</tr>
</tbody>
</table>
SDP_RFCOMM_NOT_FOUND          0xFF

See example given for BtcSppWrite.

**EVBTC_SPP_CONN_TIMEOUT**

This event is thrown when a connection attempt to an SPP device times out.

**EVBTC_SPP_DATA_RECEIVED**

This event is thrown when data is received via the Serial Port Profile. Usage is as shown in the example given for BtcSPPRead.

**EVSPPTXEMPTY**

This event is generated when the last byte in the SPP Tx buffer is transmitted. See example for BtcSppWrite().

**EVSPPDISON**

This event is thrown when an SPP disconnection occurs. The message contains **nHandle**, the handle of the connection.

```vbs
dim rc, hPort, n$, a$

function HandlerSppConn(hConn, result) as integer
    dim s$, len
    print "\n --- Connect : ",hConn
    print "\nResult: ",integer.h' result

    s$ = "Hello"
    rc=BtcSppWrite(hConn, s$, len)
    if rc==0 then
        print "\nWrote ";len;" bytes"
    else
        print "\nError: "; integer.h'rc
    endif
    rc=BtcSppDisconnect(hConn)
endfunc

function HandlerSppDiscon(portHndl) as integer
    print "\n --- Disconnect : ", portHndl
endfunc

onevent EvSppConn   call HandlerSppConn
onevent EvSppDiscon call HandlerSppDiscon

rc=BtcSetConnectable(1)
rc=BtcSetDiscoverable(1,60)
rc=BtcSppOpen(hPort)

if rc == 0 then
    print "\nSPP service open. Handle: ";hPort
else
    print "\nError: ";rc
endif

rc=BtcGetFriendlyName(n$)
a$ = SysInfo$(4)
print "\n";n$;" : ";StrHexize$(a$)
```
print "\nModule is Discoverable. Make an SPP connection to the module.\n"
waitevent

Expected Output:

SPP service open. Handle: 56833
LAIRD WB : 000016A4093A5F
Module is Discoverable. Make an SPP connection to the module.
--- Connect : 40449
Result: 00000000
Wrote 5 bytes
--- Disconnect : 40449

BtcSPPSetParams

FUNCTION

This function is used to set the parameters of newly opened SPP connections. Must be called with no active open connections. Adjusting these values from the default will effect the maximum number of SPP connections achievable.

BTCSPPPSETPARAMS (nFrameSize, nReceiveCreds)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, indicating the success of command:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Opened successfully</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nFrameSize</strong> byRef nFrameSize AS INTEGER</td>
</tr>
<tr>
<td>The maximum frame size supported on new SPP connections. Default 192 Bytes, Range is from 23-1011 bytes.</td>
</tr>
<tr>
<td><strong>nReceiveCreds</strong> byRef nReceiveCreds AS INTEGER</td>
</tr>
<tr>
<td>Number of receive packets to queue. Default 3, Range is from 1-10 packets.</td>
</tr>
</tbody>
</table>

dim rc
rc=BtcSppSetParams(256,6)
if rc == 0 then
  print "\nSPP Parameters updated."
else
  print "\nError: ";rc
endif

Expected Output:

SPP Parameters updated.

BTCSPPPSETPARAMS is an extension function.

BtcSPPOpen

FUNCTION

This function is used to open the serial port service and listen for SPP connections.
### BTCSPPOPEN (nHandle)

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

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
</table>
| **nHandle** | byVal nHandle AS INTEGER  
On return this will contain the handle for the SPP service. |

```vbnet
dim rc, hSpp  
rc=BtcSppOpen(hSpp)  
if rc == 0 then  
    print "\nSPP service open. Handle: ";hSpp  
else  
    print "\nError: ";rc  
endif  
rc=BtcSppClose(hSpp)
```

**Expected Output:**

SPP service open. Handle: 56833

BTCSPPOPEN is a extension function.

### BtcSPPCLOSE (nHandle)

**FUNCTION**  
Close the Serial Port being expedited by SPP Service.

**BTCSPPPCLOSE (nHandle)**

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
</table>
| INTEGER, indicating the success of command:  
0       Opened successfully |

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
</table>
| **nHandle** | byVal nHandle AS INTEGER  
The handle of the SPP connection to close |

```vbnet
dim rc, hSpp  
rc=BtcSppOpen(hSpp)  
rc=BtcSppClose(hSpp)  
if rc == 0 then  
    print "\nSPP port closed ";hSpp  
else  
    print "\nError: ";rc  
endif
```

**Expected Output:**

SPP port closed 56323
BTCSPPCLOSE is an extension function.

**BtcSPPWrite**

**FUNCTION**

This function is used to transmit a string of characters via the Serial Port service.

**BTCSPPWRITE (nHandle, data$, nLen)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td>nHandle</td>
<td><em>byVal</em> nHandle AS INTEGER This contains the handle for the applicable SPP connection (the WB45 can be in a connection with multiple devices).</td>
</tr>
<tr>
<td>data$</td>
<td><em>byRef</em> data$ AS STRING This contains the data to send over SPP.</td>
</tr>
<tr>
<td>nLen</td>
<td><em>byRef</em> nLen AS INTEGER On return this will contain the number of bytes written.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
<tr>
<td>Related Commands</td>
<td>BTCSSPOPEN, BTCSSPCLOSE, BTCSSPCONNECT, BTCSPPDISCONNECT, BTCSPREAD</td>
</tr>
</tbody>
</table>

**Note:** data$ cannot be a string constant (for example, “the cat”) but must be a string variable. If you must use a const string, first save it to a temp string variable and then pass it to the function.

```basic
dim rc, hPort, n$, m$

function HandlerSppCon(hConn, result) as integer
    dim s$, len
    print "\n --- Connect : ", hConn
    print "\nResult: ", integer.h' result
    s$ = "Hello"
    rc=BtcSppWrite(hConn, s$, len)
    if rc==0 then
        print "\nWrote "; len; " bytes"
    else
        print "\nError: "; integer.h'rc
    endif
endfunc 1

function HandlerSppTxEmpty(hSppConn)
endfunc 0

onevent EvSppConn     call HandlerSppCon
onevent EvSppTxEmpty   call HandlerSppTxEmpty

rc=BtcSppOpen(hPort)
rc=BtcDiscoveryConfig(0,0) //general discoverability
rc=BtcSetDiscoverable(1,60) //discoverable for 1 minute
rc=BtcSetConnectable(1) //connectable
```
WB45 smartBASIC Extensions
User Guide

rc=BtcSetPairable(0) //not pairable
rc=BtcGetFriendlyName(n$)
m$ = SysInfo$(4)
print "\n";n$;": ";StrHexize$(m$);"\n"
waitevent
print "\nExiting.."

Expected Output:

--- Connect : 40449
Result: 00000000
Wrote 5 bytes

BTCSPWRITE is an extension function.

BtcSPPRead

FUNCTION
Read data from the oldest SPP data event. Since the event EVBTC_SPP_DATA_RECEIVED is envoked everytime data is received via the SPP service, and data can be received from multiple SPP connections, this function should be called in the EVBTC_SPP_DATA_RECEIVED handler to process all waiting data.

BTCSPREAD (nHandle, data$, nLen)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nHandle</td>
<td>byRef nHandle AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>On return, this will contain the handle of the SPP connection from which the data came.</td>
</tr>
<tr>
<td>data$</td>
<td>byRef data$ AS STRING</td>
</tr>
<tr>
<td></td>
<td>On return, this will contain the data received from the connection identified by the handle above.</td>
</tr>
<tr>
<td>nLen</td>
<td>byRef nLen AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>On return this will contain the number of bytes read.</td>
</tr>
</tbody>
</table>

Interactive Command No

Related Commands BTCSPPOPEN, BTCSPPCLOSE, BTCSPPCONNECT, BTCSPPDOCONNECTION, BTCSPWRITE

Note: data$ cannot be a string constant (for example, “the cat”) but must be a string variable.

dim rc
dim hSpp
dim n$, a$

function HandlerSppConn(portHandle, result)
    print "\n --- Connect : ",portHandle
    print "\nResult: ";integer.h' result
endfunc

'//called when data is received via spp
function HandlerSppData()

dim hPort

dim data$

dim readLen

'//read and print data while there is data available to read
while BtcSppRead(hPort, data$, readLen) == 0
    if readLen>0 then
        print"\nPort Handle: ";hPort; "\nData: ";data$;"\nLength: ";readLen
    endif
endwhile
endfunc 1

rc=BtcSppOpen(hSpp)

if rc == 0 then
    print "\nSPP service open. Handle: ";hSpp
else
    print "\nError: ";rc
endif

OnEvent EVSPPCONN call HandlerSppConn
OnEvent EVBTC_SPP_DATA_RECEIVED call HandlerSppData

rc=BtcSetConnectable(1)
rc=BtcSetDiscoverable(1,60)
rc=BtcSppOpen(hSpp)

rc=BtcGetFriendlyName(n$)
a$ = SysInfo$(4)
print "\n";n$;" : ";StrHexize$(a$)
print "\nModule is Discoverable. Make an SPP connection\n"

WaitEvent

Expected Output:

SPP service open. Handle: 56833
LAIRD WB : 000016A4093A5F
Module is Discoverable. Make an SPP connection

--- Connect : 40449
Result: 00000000
Port Handle: 40449
Data: hello
Length: 6

Btcsppread is an extension function.

BtcSPPConnect
FUNCTION
Connect to an SPP device defined by btaddr$.

Btcsppconnect (btaddr$)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
</table>

Embedded Wireless Solutions Support Center: America: +1-800-492-2320
http://ews-support.lairdtech.com Europe: +33-1-10-420-700
www.lairdtech.com/wireless Hong Kong: +852 2923 0610
Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>btaddr$</td>
<td>Bluetooth address of the device for connection</td>
</tr>
</tbody>
</table>

Interactive Command: No

Related Commands: BTCSPPOPEN, BTCSPPCLOSE, BTCSPPDISCONNECT, BTCSPPREAD, BTCSPPWRITE

```vbnet
dim rc, i

'//BT address of device to connect to. You will have to change this
dim BTA$
BTA$ = "00\16\A4\09\3A\5F"

'//array with handles for spp connections
dim hSpp

rc=BtcSetConnectable(1)
rc=BtcSetDiscoverable(1,60)

'//make spp connection
rc=BTCSPPConnect(BTA$)

function HandlerSppConn(portHndl, result) as integer
    hSpp = portHndl
    print "\n--- Connect : ",hSpp, StrHexize$(BTA$)
    print "\nResult: ",integer.h' result
endfunc

onevent EvSppConn call HandlerSppConn
waitevent

print "\nExiting..."
```

Expected Output:

```
Connecting to device 0016A4093A5F
--- Connect : 40449 0016A4093A5F
Result: 00000000
Exiting...
```

BTCSPPCONNECT is an extension function.

**BtSPPDisconnect**

**FUNCTION**

Disconnect from an SPP device

**BTCSPPDISCONNECT(nHandle)**

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

<table>
<thead>
<tr>
<th>nHandle</th>
<th>BYREF nHandle AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The handle of the connection to be dropped</td>
</tr>
</tbody>
</table>

Interactive Command

No

Related Commands

BTCSPPOPEN, BTCSPPCLOSE, BTCSPPREAD, BTCSPPWRITE, BTCSPPCONNECT

dim rc, hConn, n$, hPort, a$

function HandlerSppConn(portHndl, result) as integer

dim s$, len
hConn = portHndl
print "\n --- Connect :","", hConn
print "\nResult: ";integer.h' result

rc=BtcSppDisconnect(hConn)
if rc==0 then
    print "\n\nDisconnecting..."
else
    print "\nError:", integer.h'rc
endif
endfunc

// Called on an SPP disconnection
// Function HandlerSppDiscon(hConn) as integer
// rc=BtcSppClose(hPort)
endfunc

onevent EvSppConn    call HandlerSppConn
onevent EvSppDiscon  call HandlerSppDiscon

rc=BtcGetFriendlyName(n$)
a$ = SysInfo$(4)
print "\n\n\nModule is Discoverable. Make an SPP connection\n"

rc=BtcSetConnectable(1)
rc=BtcSetDiscoverable(1,60)
rc=BtcSppOpen(hPort)

waitevent

Expected Output:

LAIRD WB: 000016A4093A5F
Module is Discoverable. Make an SPP connection

--- Connect : 40449
Result: 00000000

Disconnecting...
--- Disconnected : 40449
Stream Functions

StreamGetUartHandle

FUNCTION

Returns the stream handle of the UART.

STREAMGETUARThANDLE(nStreamHandle)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nHandle</td>
<td>BYREF bStreamHandle AS INTEGER Returns the handle of the UART</td>
</tr>
</tbody>
</table>

Interactive Command: No

Related Commands: STREAMBRIDGE, STREAMUNBRIDGE, STREAMGETSPPHANDLE

See example for StreamBridge

STREAMGETUARThANDLE is an extension function.

StreamGetSPPHandle

FUNCTION

Get the stream handle of an SPP connection.

STREAMGETSPPHANDLE(nHandle, nStreamHandle)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nHandle</td>
<td>BYVAL nHandle AS INTEGER The handle of the SPP connection to use</td>
</tr>
<tr>
<td>nHandle</td>
<td>BYREF nStreamHandle AS INTEGER The handle of the stream port</td>
</tr>
</tbody>
</table>

Interactive Command: No

Related Commands: STREAMGETUARThANDLE, STREAMBRIDGE, STREAMUNBRIDGE

See example for StreamBridge

STREAMGETSPPHANDLE is an extension function.

StreamBridge

FUNCTION

Bridges two stream connections together.

STREAMBRIDGE(nHandleOne, nHandleTwo, nHandle)
Returns

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

Arguments:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nHandleOne</td>
<td>BYVAL nHandleOne AS INTEGER</td>
</tr>
<tr>
<td>nHandleTwo</td>
<td>BYVAL nHandleTwo AS INTEGER</td>
</tr>
<tr>
<td>nHandleB</td>
<td>BYREF nHandle AS INTEGER</td>
</tr>
</tbody>
</table>

Interactive Command

No

Related Commands

STREAMGETUARThANDLE, STREAMUNBRIDGE, STREAMGETSPPHANDLE

//Example :: StreamBridge.sb

```vbnet
SUB AssertRC(rc,line)
    IF rc != 0 THEN
        PRINT "Error at line ";line;", code: ";rc;
    ENDIF
ENDSUB

FUNCTION HandlerPairReq()
    //Pair request
    dim BTA$
    rc=BtcGetPairRequestBDAddr(BTA$
    AssertRC(rc, 12)
    PRINT "Pairing requested from device: "; StrHexize$(BTA$
    PRINT "Accepting pair request"
    rc=BtcSendPairResp(1)
    AssertRC(rc, 16)
ENDFUNC

FUNCTION SPPConnect(nHandle, Result)
    //SPP connected
    dim UARTStream, SPPStream
    nSppHandle = nHandle
    PRINT "Connected"n"

    //Bridge to UART
    rc = StreamGetUartHandle(UARTStream)
    AssertRC(rc, 27)
    rc = StreamGetSPPHandle(nSppHandle, SPPStream)
    AssertRC(rc, 29)
    rc = StreamBridge(UARTStream, SPPStream, nHandleB)
    AssertRC(rc, 31)
ENDFUNC

FUNCTION SPPTimeout()
    //SPP connection timeout
    PRINT "Timeout"n"
ENDFUNC

FUNCTION SPPDisconnect(nHandle)
    //SPP disconnection. Remove UART bridge
```c
rc = StreamUnBridge(nHandleB)
AssertRC(rc, 42)
PRINT "Disconnected"
ENDFUNC 1

//Create SPP host connection
rc=BtcDiscoveryConfig(0, 0)
rc=BtcSetConnectable(1)
rc=BtcSetPairable(1)
rc=BtcSavePairings(1)
rc=BtcSetDiscoverable(1, 0)
rc=BtcSppOpen(nSHandleG)

//SPP Events
ONEVENT EVSPPCONN CALL SPPConnect //SPP connected
ONEVENT EVBTC_SPP_CONN_TIMEOUT CALL SPPTimeout //SPP connection timeout
ONEVENT EVSPPDISCON CALL SPPDisconnect //SPP disconnection
ONEVENT EVBTC_PAIR_REQUEST CALL HandlerPairReq //Pair request

WAITEVENT
```

### Expected Output:

```
Connected
Test Data from another WB45
Disconnected
```

STREAMBRIDGE is an extension function.

**StreamUnBridge**

**FUNCTION**

Unbridges a stream connection created using StreamBridge.

**STREAMUNBRIDGE(nHandle)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td>nHandle</td>
<td><strong>BYVAL nHandle AS INTEGER</strong> The handle of the bridged connection to unbridge</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
<tr>
<td>Related Commands</td>
<td>STREAMGETUARTHANDLE, STREAMBRIDGE, STREAMGETSPPHANDLE</td>
</tr>
</tbody>
</table>

See example for [StreamBridge](#)

STREAMUNBRIDGE is an extension function.

### Pairing/Bonding Functions

This section describes the functions related to pairing and bonding manager which manages trusted devices. Pairing is the process of two devices exchanging a link key. This is required each time one of the devices is set pairable and the other device tries to connect (if the two devices are not bonded). If the link key (and other
information including the Bluetooth address of the peer) gets stored in the bonding manager when pairing, the two devices become bonded and do not need to pair again upon subsequent connections.

The bonding manager consists of a rolling database and a persistent database. A link key for a new bond is always stored in the rolling database. When the rolling database is full and a new bond is created, the oldest link key in this database is replaced with the key for the new bond. To prevent a link key from being replaced, it can be moved to the persistent database by calling BtcBondingPersistKey() where it won’t be replaced unless BtcBondingEraseKey() or BtcBondingEraseAll() is called.

Events and Messages

**EVBTC_PAIR_REQUEST**

This event is thrown on a pairing request from another device. See examples given for [EVBTC_PAIR_RESULT](#) and [BtcPair](#).

**EVBTC_PIN_REQUEST**

This event is thrown on a PIN request from another device during pairing. See examples given for [EVBTC_PAIR_RESULT](#) and [BtcPair](#).

**EVBTC_PAIR_RESULT**

This message is thrown after a pairing attempt and comes with one parameter which is the result code. A list of result codes and descriptions can be found [here](#).

```dim rc,mac$,pin$,n$,a$
pin$ = "271192"

//==============================================================================
// Called on a Pairing request from another device
//==============================================================================
function HandlerPairReq()
  rc=BtcGetPairRequestBDAddr(mac$)
  if rc==0 then
    print "\nPairing requested from device: "; StrHexize$(mac$)
    print "\nAccepting pair request"
    rc=BtcSendPairResp(1)
  else
    print "\nErr: "; integer.h'rc
  endif
endfunc 1

//==============================================================================
// Called on a PIN request from another device
//==============================================================================
function HandlerPinReq()
  rc=BtcGetPinRequestBDAddr(mac$)
  if rc==0 then
    print "\nPIN requested from device: "; StrHexize$(mac$)
    print "\nSending PIN response with PIN '271192'"
    rc=BtcSendPINResp(pin$)
  else
    print "\nErr: "; integer.h'rc
  endif
endfunc 1

//==============================================================================
// Called after a pairing attempt
//==============================================================================
```
function HandlerPairRes(nRes)
    if nRes == 0 then
        print "\n--- Successfully paired with device ";StrHexize$(mac$)
    else
        print "\n--- Pairing attempt error: (";integer.h'nRes;")"
    endif
endfunc

rc=BtcSetConnectable(1)
rc=BtcSetDiscoverable(1,60)
rc=BtcSetPairable(1)

rc=BtcGetFriendlyName(n$)
a$ = SysInfo$(4)
print "\n";n$; ": ";StrHexize$(a$)
print "\nModule is Discoverable and Pairable. Pair with the module.\n"

WaitEvent

Expected Output (Legacy Pairing):

LAIRD WB : 000016A4093A5F
Module is Discoverable and Pairable. Pair with the module.

PIN requested from device: 0016A400115E
Sending PIN response with PIN '271192'
--- Successfully paired with device 0016A400115E

Expected Output (Simple Secure Pairing)

LAIRD WB : 000016A4093A5F
Module is Discoverable and Pairable. Pair with the module.

Pairing requested from device: 0016A4093A92
Accepting pair request
--- Successfully paired with device 0016A4093A92

BtcGetPAIRREQUESTBDAddr

FUNCTION
Get the bluetooth address of the device requesting a pairing using Secure Simple Pairing.

BTCGETPAIRREQUESTBDADDR (strBDAddr$)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>strBDAddr$</td>
<td>byREF strBDAddr$ AS STRING On return this string will contain the bluetooth address of the device that the pairing request came from.</td>
</tr>
<tr>
<td>Interactive</td>
<td>No</td>
</tr>
</tbody>
</table>

Embedded Wireless Solutions Support Center:
http://ews-support.lairdtech.com
www.lairdtech.com/wireless

Americas: +1-800-492-2320
Europe: +44-1628-858-940
Hong Kong: +852 2923 0610
### Command

See examples given for [EVBTC_PAIR_RESULT](#) and [BtcPair](#).

BtcGetPINRequestBDAddr is an extension function.

#### BtcGetPINRequestBDAddr

**FUNCTION**

Get the bluetooth address of the device requesting a pairing using Legacy PIN.

[BTCGETPINREQUESTBDADDR](#)(strBDAddr$)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>strBDAddr$</td>
<td>byREF strBDAddr$ AS STRING On return, this string contains the bluetooth address of the device requesting a PIN.</td>
</tr>
<tr>
<td>Interactive</td>
<td>No</td>
</tr>
<tr>
<td>Command</td>
<td>No</td>
</tr>
</tbody>
</table>

See examples given for [EVBTC_PAIR_RESULT](#) and [BtcPair](#).

BtcGetPINRequestBDAddr is an extension function.

#### BtcSendPAIRResp

**FUNCTION**

This function is used to accept or decline a pairing request.

[BTCSENDPAIRRESP](#)(nAccept)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nAccept</td>
<td>byVAL nAccept AS INTEGER 0 Decline 1 Accept</td>
</tr>
<tr>
<td>Interactive</td>
<td>No</td>
</tr>
<tr>
<td>Command</td>
<td>No</td>
</tr>
</tbody>
</table>

BTCSENDPAIRRESP is an extension function. See example given for [EVBTC_PAIR_RESULT](#).

#### BtcSendPINResp

**FUNCTION**

During a pairing procedure, this function responds to a PIN request with a given PIN.

[BTCSENDPINRESP](#)(strPIN$)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code.</th>
</tr>
</thead>
</table>
The most typical value is 0x0000, indicating a successful operation.

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strPIN$</td>
<td>This is the PIN that is used. For example: 1234</td>
</tr>
</tbody>
</table>

Interactive Command: No

See examples given for EVBTC_PAIR_RESULT and BtcPair.

BtcSendPinResp is an extension function.

**BtcSavePairings**

FUNCTION

For subsequent incoming pair requests, this function sets whether or not to bond with devices by storing the relevant information (including the link key and Bluetooth address) in the bonding manager.

**BTCSAVEPAIRINGS**(fSave)

Returns: INTEGER, a result code.

The most typical value is 0x0000, indicating a successful operation.

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fSave</td>
<td>If this flag is: 0 – Pairing information is not stored in the bonding manager 1 – Pairing information is stored in the bonding manager</td>
</tr>
</tbody>
</table>

Interactive Command: No

Dim rc
rc=BtcSavePairings(1)
print "\nrc: "; rc

Expected Output:

0

BTCSAVEPAIRINGS is an extension function.

**BtcPair**

FUNCTION

This function is used to initiate pairing with the device identified by the given Bluetooth address and to specify whether to bond with the device by storing pairing information in the bonding manager. Before using this function, the WB45 must be set Pairable using the function BtcSetPairable().

**BTCPAIR**(strBDAddr$, nSave)

Returns: INTEGER, a result code.

The most typical value is 0x0000, indicating a successful operation.

Argument:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strBDAddr$</td>
<td>The Bluetooth address of the device to pair with. Must be 6 bytes long.</td>
</tr>
</tbody>
</table>
### SmartBASIC Extensions

**User Guide**

#### nSave

This flag sets whether or not to bond.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Do not store pairing information (don’t bond)</td>
</tr>
<tr>
<td>1</td>
<td>Store pairing information (bond)</td>
</tr>
<tr>
<td>2</td>
<td>Use default as specified by BtcSavePairings()</td>
</tr>
</tbody>
</table>

#### Interactive Command

No

---

```basic
dim rc, adr$, n$, m$
#define BOND_WHEN_PAIRING   1

//You will need to change the following defines
#define PIN     "0000"
#define DEV_BT_ADDR  "\94\35\0A\A9\9A\3C"

adr$ = DEV_BT_ADDR
    // adr$ = StrDehexize$(adr$)

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

// Called when there is a pairing request from another device
//------------------------------------------------------------------------------
function HandlerPairReq()
    rc=BtcGetPAIRRequestBDAddr(adr$)
    print "\nPair Req: ";StrHexize$(adr$)
    rc=BtcSendPairResp(1)
    print "\nAccepted, Pairing...
endfunc

// Called on a PIN request from another device
//------------------------------------------------------------------------------
function HandlerPINReq()
    rc=BtcGetPinRequestBDAddr(adr$)
    print "\nPIN Req. Sending pin " + PIN
    rc=BtcSendPinResp(PIN)
endfunc

// Called after a pairing attempt
//------------------------------------------------------------------------------
function HandlerPairRes(res)
    dim i : i=res
```
```basic
print "\n --- Pair Result: ("; integer.h'res; ");StrHexize$(adr$);"\n"
endfunc 0

onevent evbtc_pin_request call HandlerPINReq
	//These two events MUST have handlers registered for them
onevent evbtc_pair_result call HandlerPairRes
onevent evbtc_pair_request call HandlerPairReq

'//get friendly name, print it and the BT address
rc=BtcGetFriendlyName(n$)
m$ = SysInfo$(4)
print n$;" : ";StrHexize$(m$)

'//Set connectable and pairable
rc=BtcSetConnectable(1)
if rc==0 then
    print "\nConnectable"
endif
rc=BtcSetPairable(1)
if rc==0 then
    print "\nPairable"
endif
rc=BtcPair(adr$, BOND_WHEN_PAIRING)
AssertRC(rc,51)

waitevent

Expected Output:

LAIRD WB : 000016A4093A5F
Connectable
Pairable
Initiating Pairing...
Pair Req: 94350AA99A3C
Accepted, Pairing...
--- Pair Result: (00000000) 94350AA99A3C

BTCPAIR is an extension function.

BtcBondingStats

FUNCTION
This function is used to get the classic BT bonding manager database statistics.

BTCBONDINGSTATS (nRolling, nPersistent)

<table>
<thead>
<tr>
<th>Arguments:</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nRolling</strong></td>
<td>The total capacity of the database&lt;br&gt;byREF nRolling AS INTEGER&lt;br&gt;On return, this integer contains the total number of bonds in the rolling database.</td>
</tr>
<tr>
<td><strong>nPersistent</strong></td>
<td>byREF nPersistent AS INTEGER&lt;br&gt;On return, this integer contains the total number of bonds in the persistent database.</td>
</tr>
<tr>
<td>Interactive</td>
<td>No</td>
</tr>
</tbody>
</table>

Embedded Wireless Solutions Support Center:
http://ews-support.lairdtech.com
www.lairdtech.com/wireless

Americas: +1-800-492-2320
Europe: +44-1628-858-940
Hong Kong: +852 2923 0610
Command

<table>
<thead>
<tr>
<th>dim rc, nRoll, nPers</th>
</tr>
</thead>
<tbody>
<tr>
<td>print &quot;\n:Bonding Manager Database Statistics:&quot;</td>
</tr>
<tr>
<td>print &quot;\nCapacity: &quot;,&quot;&quot;, BtcBondingStats(nRoll, nPers)</td>
</tr>
<tr>
<td>print &quot;\nRolling: &quot;,&quot;&quot;,nRoll</td>
</tr>
<tr>
<td>print &quot;\nPersistent: &quot;,nPers</td>
</tr>
</tbody>
</table>

Expected Output:

:Bonding Manager Database Statistics:
Capacity: 16
Rolling: 2
Persistent: 0

BtcBondingEraseKey

FUNCTION

This function is used to erase a link key from the database for the specified BT address.

BTCBONDINGERASEKEY (btaddr$)

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

Arguments:

<table>
<thead>
<tr>
<th>btaddr$</th>
<th>byREF btaddr$ AS STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bluetooth address in big endian. Must be exactly six bytes long.</td>
</tr>
</tbody>
</table>

Interactive Command
No

Expected Output:

Link key for device 008098044E91 erased
BTCBONDINGERASEKEY is an extension function.

**BtcBondingEraseAll**

**FUNCTION**

This function is used to erase all link keys in the database, including both those in the rolling and persistent databases.

**BTCBONDINGERASEALL ()**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```vbscript
dim rc, nRoll, nPers
' //------------------------------------------------------------------------------
' // For debugging
' // --- rc = result code
' // --- ln = line number
' //------------------------------------------------------------------------------
Sub AssertRC(rc,ln)
  if rc!=0 then
    print "\nFail :";integer.h' rc;" at tag ";ln
  else
    print "\nAll link keys in the bonding manager database erased\n"
  endif
EndSub

rc=BtcBondingEraseAll()
AssertRC(rc,17)

print "\n:Bonding Manager Database Statistics:"
print "\nCapacity: ", BtcBondingStats(nRoll, nPers)
print "\nRolling: ",nRoll
print "\nPersistent: ",nPers
```

**Expected Output:**

```
All link keys in the bonding manager database erased

:Bonding Manager Database Statistics:
Capacity: 16
Rolling: 0
Persistent: 0
```

BTCBONDINGERASEALL is an extension function.

**BtcBondingPersistKey**

**FUNCTION**

This function is used to make a link key persistent by transferring it from the rolling database to the persistent database.
BTCBONDINGPERSISTKEY (btaddr$)

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>btaddr$</td>
<td>byREF btaddr$ AS STRING Bluetooth address in big endian. Must be exactly six bytes long.</td>
</tr>
</tbody>
</table>

Interactive Command: No

```
dim rc, BTA$, key$, nRoll, nPers

'//---------------------------------------------------------------------
'// For debugging
'// --- rc = result code
'// --- ln = line number
'//---------------------------------------------------------------------
Sub AssertRC(rc,ln)
  if rc!=0 then
    print "\nFail :";integer.h' rc;" at tag ";ln
  else
    print "\nLink key for device ";StrHexize$(BTA$); " now persistent\n"
  endif
EndSub

'//Make a link key persistent
BTA$="\00\80\98\04\4E\91"
rc=BtcBondingPersistKey(BTA$)
AssertRC(rc,35)

print "\nBonding Manager Database Statistics:
print "\nCapacity: ",",", BtcBondingStats(nRoll, nPers)
print "\nRolling: ",",",nRoll
print "\nPersistent: ",nPers
```

Expected Output:

```
Link key for device 008098044E91 now persistent

:Bonding Manager Database Statistics:
Capacity: 16
Rolling: 3
Persistent: 1
```

BTCBONDINGPERSISTKEY is an extension function.

BtcBondingGetFirst

FUNCTION

This function is used to retrieve details about the first classic Bluetooth bond in the WB45’s database. Information returned includes the key, the type of the key, the database its located in and the target Bluetooth address.

BTCBONDINGGETFIRST (btaddr$, btkey$, keytype, bonddb)
Returns

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

Arguments:

`btaddr$` byREF `btaddr$` AS STRING
Bluetooth address in big endian. Will be exactly six bytes long.

`btkey$` byREF `btkey$` AS STRING
Bluetooth bond key. Will be exactly sixteen bytes long.

`keytype` byREF `keytype` AS INTEGER
Returns the type of the key;

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Combination key</td>
</tr>
<tr>
<td>1</td>
<td>Local unit key</td>
</tr>
<tr>
<td>2</td>
<td>Remote unit key</td>
</tr>
<tr>
<td>3</td>
<td>Debug combination key</td>
</tr>
<tr>
<td>4</td>
<td>Unauthenticated combination key</td>
</tr>
<tr>
<td>5</td>
<td>Authenticated combination key</td>
</tr>
<tr>
<td>6</td>
<td>Changed combination key</td>
</tr>
<tr>
<td>7</td>
<td>Illegal key</td>
</tr>
</tbody>
</table>

`bonddb` byREF `bonddb` AS INTEGER
Which database the key is in;

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Persistent</td>
</tr>
<tr>
<td>1</td>
<td>Rolling</td>
</tr>
</tbody>
</table>

Interactive Command

No

dim rc, Addr$, Key$, Type, DB

rc = BTCBondingGetFirst(Addr$, Key$, Type, DB)

IF (rc == 0) THEN
   PRINT "Address ";STRHEXIZE$(Addr$);", key: ";STRHEXIZE$(Key$);", type:"
   ;Type;" in 
   IF (DB == 1) THEN
      //Rolling
      PRINT "rolling"
   ELSE
      //Persistent
      PRINT "persistent"
   ENDIF
   PRINT " database."
ENDIF

//Get next key
rc = BTCBondingGetNext(Addr$, Key$, Type, DB)

IF (rc == 0) THEN
   //Additional bond(s)
   PRINT "Address ";STRHEXIZE$(Addr$);", key: ";STRHEXIZE$(Key$);", type:"
   ;Type;" in 
   IF (DB == 1) THEN
      //Rolling
      PRINT "rolling"
   ELSE
      //Persistent

Expected Output:

```
Address 0016A406ACCC, key: 0227E51A6F509ED11C4C603AD0E41728, type: 4 in rolling database.
No additional bonds
```

BTCBONDINGGETFIRST is an extension function.

**BtcBondingGetNext**

FUNCTION

This function is used to retrieve details about the next classic Bluetooth bond in the WB45s database (after having used BtcBondingGetFirst). Information returned includes the key, the type of the key, the database its located in and the target Bluetooth address.

**BTCBONDINGGETNEXT (btaddr$, btkey$, keytype, bonddb)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td><em>btaddr</em>$</td>
<td>byREF btaddr$ AS STRING</td>
</tr>
<tr>
<td><em>btkey</em>$</td>
<td>byREF btkey$ AS STRING</td>
</tr>
<tr>
<td><em>keytype</em></td>
<td>byREF keytype AS INTEGER</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>0</td>
<td>Combination key</td>
</tr>
<tr>
<td>1</td>
<td>Local unit key</td>
</tr>
<tr>
<td>2</td>
<td>Remote unit key</td>
</tr>
<tr>
<td>3</td>
<td>Debug combination key</td>
</tr>
<tr>
<td>4</td>
<td>Unauthenticated combination key</td>
</tr>
<tr>
<td>5</td>
<td>Authenticated combination key</td>
</tr>
<tr>
<td>6</td>
<td>Changed combination key</td>
</tr>
<tr>
<td>7</td>
<td>Illegal key</td>
</tr>
<tr>
<td><em>bonddb</em></td>
<td>byREF bonddb AS INTEGER</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>0</td>
<td>Persistent database</td>
</tr>
<tr>
<td>1</td>
<td>Rolling database</td>
</tr>
</tbody>
</table>
See example for \textbf{BtcBondingGetFirst}.

BTCBONDINGGETNEXT is an extension function.

\section*{Miscellaneous Functions}

\subsection*{Events and Messages}

\textit{EVBTC\_DISCOV\_TIMEOUT}

This event is thrown when the module is no longer discoverable. This will be after the time specified with \textbf{BtcSetDiscoverable()}, otherwise it will be after the default value of 60 seconds.

See example given for \textbf{BtcSetDiscoverable}.

\subsection*{BtcGetFriendlyName}

\textbf{FUNCTION}

Get the friendly name of this device as seen by other devices.

\textbf{BTCGETFRIENDLYNAME (name$)}

\begin{itemize}
  \item \textbf{Returns} INTEGER, a result code.
    The most typical value is 0x0000, indicating a successful operation.
  \item \textbf{Arguments:}
    \begin{itemize}
      \item \textit{name$} byREF name$ AS STRING
        On return this string contains the device name
    \end{itemize}
\end{itemize}

\begin{verbatim}
dim rc, name$
rc=BtcGetFriendlyName(name$)
print "\n"; name$
\end{verbatim}

\textbf{Expected Output:}

\begin{verbatim}
Laird WB
\end{verbatim}

BTCGETFRIENDLYNAME is an extension function.

\subsection*{BtcSetFriendlyName}

\textbf{FUNCTION}

Set the friendly name for this module. This name is visible to other Bluetooth Classic devices doing an extended inquiry if they discover the module.

\textbf{BTCSETFRIENDLYNAME (name$)}

\begin{itemize}
  \item \textbf{Returns} INTEGER, a result code.
    The most typical value is 0x0000, indicating a successful operation.
  \item \textbf{Arguments:}
    \begin{itemize}
      \item \textit{name$} AS STRING
        On return this string contains the device name
    \end{itemize}
\end{itemize}

\begin{verbatim}
dim rc, name$
rc=BtcSetFriendlyName(name$)
print "\n"; name$
\end{verbatim}

\textbf{Expected Output:}

\begin{verbatim}
Laird WB
\end{verbatim}

BTCSETFRIENDLYNAME is an extension function.
The new name to set. The maximum allowed length is 31 characters.

Interactive Command | No

dim rc, name$
name$ = "My WB45"
rc=BtcSetFriendlyName(name$)
print "\n"; name$

Expected Output:
My WB45

BtcSetFriendlyName is an extension function.

**BtcDiscoveryConfig**

**FUNCTION**

When a Bluetooth device is discoverable, it listens for inquiries from other Bluetooth devices by performing an inquiry scan. An Inquiry Window and Inquiry Interval are used to optimise power usage:

- Inquiry Interval – The time between inquiry scans.
- Inquiry Window – The duration of the inquiry scan.

This function is used to set the parameters and the discoverability type of this module. If the module is set for General Discoverability, it is seen by devices doing a General Inquiry. If set for Limited Discoverability, the module is only seen by devices doing a Limited Inquiry.

**Note:** Limited Discoverability is not currently supported and will be implemented in future releases of the firmware.

**BTCDISCOVERYCONFIG (nConfigID,nValue)**

**Returns**

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

**Arguments:**

<table>
<thead>
<tr>
<th>nConfigID</th>
<th>byVal nConfigID AS INTEGER. This identifies the value to update as follows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Discoverability type: 0 = General (default) 1 = Limited</td>
</tr>
</tbody>
</table>
| 1         | Inquiry Scan Interval  
Units: Baseband slots (0.625 msec)  
Range: 11.25 msec (0x0012) to 2560 msec (0x1000)  
Default: 640 ms (0x0400)                          |
| 2         | Inquiry Scan Window – Must be less than or equal to the Inquiry Scan interval  
Units: Baseband slots (0.625 msec)  
Range: 11.25 msec (0x0012) to 2560 msec (0x1000)  
Default: 320 ms (0x0200)                          |

**Note:** For all other configID values, the function returns an error.
nValue | byVal nValue AS INTEGER.
The new value to set for the parameter identified by configID.

Interactive Command | No

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

dim rc
rc=BtcDiscoveryConfig(0,0) //general
AssertRC(rc,17)
rc=BtcDiscoveryConfig(1,0x320) //inquiry scan interval of 500ms (0x0320)
AssertRC(rc,19)
rc=BtcDiscoveryConfig(2,0x190) //inquiry scan interval of 250ms (0x0190)
AssertRC(rc,21)

Expected Output:
Discovery Parameter set: line 17
Discovery Parameter set: line 19
Discovery Parameter set: line 21

BTC DISCOVERYCONFIG is an extension function.

BtcSetDiscoverable

FUNCTION
This function sets the module discoverable for the time specified time or not discoverable. It will set the module for the discoverability type specified by BtcDiscoveryConfig().

BtCSETDISCOVERABLE (nEnable, nTimeout)

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

Arguments:

| nEnable | byVal nEnable AS INTEGER
|---------|---------------------------
| 0       | – Not discoverable
| 1       | – Discoverable

| nTimeout | byVal nTimeout AS INTEGER
|----------|-----------------------------
|          | The length of time in seconds that the module is discoverable.
|          | Default: 60 seconds. If nEnable is set to zero (0), this parameter is ignored.

Interactive Command | No
dim rc, n$  
n$ = "My WB45"

function HandlerDiscTimeout()  
   print "\nNo longer discoverable"  
endfunc 0

rc=BtcSetFriendlyName(n$)  

'//Enable discoverability for 10 seconds  
rc=BtcSetDiscoverable(1,10)  
if rc==0 then  
   print "\nDiscoverable for 10 seconds"  
else  
   print "\nFailed: ";integer.h'rc  
endif

onevent evbtc_discov_timeout call HandlerDiscTimeout

waitevent

print "\nExiting..."

Expected Output:

Discoverable for 10 seconds  
No longer discoverable  
Exiting...

BTCSETDISCOVERABLE is an extension function.

**BtcSetConnectable**

**FUNCTION**

This function enables or disables connectivity. It must be enabled in order for incoming connections to work. It must also be enabled if you are enabling pairability as well.

**BTCSETCONNECTABLE(nEnable)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| nEnable | ByVal nEnable AS INTEGER  
|         | 0 – Not connectable  
|         | 1 – Connectable |

| Interactive Command | No |

dim rc

rc=BtcSetConnectable(1)  
if rc==0 then  
   print "\nModule is now connectable"  
endif

See also example for BtcSppWrite().
Expected Output:

Module is now connectable

BTCSETCONNECTABLE is an extension function.

**BtcSetPairable**

**FUNCTION**

This function enables or disables pairability. If set pairable, you will receive a pairing request on outgoing and incoming connections if a bond has not already been established with the device to which you are connecting.

**Note:** The WB45 has to also be set as connectable in order to receive incoming pairing requests.

**BTCSETPAIRABLE**(nEnable)

**Returns**

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

**Arguments:**

<table>
<thead>
<tr>
<th>nEnable</th>
<th>byVal nEnable AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>– Not pairable</td>
</tr>
<tr>
<td>1</td>
<td>– Pairable</td>
</tr>
</tbody>
</table>

**Interactive Command**

No

```vbscript
    dim rc
    rc=BtcSetPairable(1)
    if rc==0 then
        print "\nModule is now pairable"
    endif
```

Expected Output:

Module is now pairable

See also example for **EVBTC_PAIR_RESULT**.

BTCSETPAIRABLE is an extension function.

**BtcGetBDAddrFromHandle**

**FUNCTION**

This function is used to get the Bluetooth address of the remote Bluetooth device given by the connection handle.

**BTCGETBDADDRFROMHANDLE**(connHandle, strBDAddr$)

**Returns**

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

**Arguments:**

<table>
<thead>
<tr>
<th>connHandle</th>
<th>byREF connHandle AS INTEGER</th>
</tr>
</thead>
</table>

Embedded Wireless Solutions Support Center: 50
http://ews-support.lairdtech.com
www.lairdtech.com/wireless

Americas: +1-800-492-2320
Europe: +44-1628-858-940
Hong Kong: +852 2923 0610
Handle of the connection from which to obtain the Bluetooth address

\textbf{strBDAddr$}$ byREF \textbf{strBDAddr$}$ AS STRING

On return, this string contains the Bluetooth address of the device on the other end of the connection

Interactive Command
No

See example for \texttt{BtcGetHandleFromBDAddr}.

\texttt{BTCGETBDADDRFROMHANDLE} is an extension function.

**BtcGetHandleFromBDAddr**

\textbf{FUNCTION}

This function is used to obtain the connection handle of the remote Bluetooth device with the given Bluetooth address.

\texttt{BTCGETHANDLEFROMBDADDR (strBDAddr$, connHandle)}

\textbf{Returns}
\texttt{INTEGER}, a result code.

The most typical value is \texttt{0x0000}, indicating a successful operation.

\textbf{Arguments:}

\texttt{strBDAddr$}$ byREF \texttt{strBDAddr$}$ AS STRING

Bluetooth address of the device on the other end of the connection for which you want to obtain the handle

\texttt{connHandle}$ byREF \texttt{connHandle}$ AS INTEGER

On return, this integer contains the connection handle

Interactive Command
No

```
dim rc, hPort, n$, a$

function HandlerSppCon(hConn, result) as integer
    dim addr$, len
    print "\n--- Connect : ", hConn
    print "\nResult: ", integer.h' result

    rc=BtcGetBDAddrFromHandle(hConn, addr$)
    if rc==0 then
        print "\nConnected to device: "; StrHexize$(addr$)
        dim h
        print "\nConnection Handle obtained from BT Address: "; h
    else
        print "\nError obtaining Bluetooth address: "; integer.h'rc
    endif
endfunc 1

onevent EvSppConn call HandlerSppCon

rc=BtcSetConnectable(1)
rc=BtcSetDiscoverable(1,60)
rc=BtcSppOpen(hPort)
rc=BtcGetFriendlyName(n$)
```
6. BLE EXTENSIONS BUILT-IN ROUTINES

Bluetooth Address

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

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

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

The exact details and format of how the specification requires this to be managed is not relevant for the purpose of how BLE functionality is exposed in this module. Only how various API functions in smartBASIC expect Bluetooth addresses are provided is explained.

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

Address types:

<table>
<thead>
<tr>
<th>Address Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Public</td>
</tr>
<tr>
<td>1</td>
<td>Random Static</td>
</tr>
<tr>
<td>2</td>
<td>Random Private Resolvable</td>
</tr>
<tr>
<td>3</td>
<td>Random Private Non-Resolvable</td>
</tr>
<tr>
<td></td>
<td>All other values are illegal</td>
</tr>
</tbody>
</table>

For example, to specify a public address which has the Bluetooth portion as 112233445566, then the STRING variable shall contain seven octets (00112233445566) and a variable can be initialised using a constant string by escaping as follows:

```plaintext
DIM addr
addr="\001\112\233\445\556"
```

Static random address

01C12233445566 (upper 2 bits of Bluetooth portion == 11)
Resolvable random address | 02412233445566 (upper 2 bits of Bluetooth portion ==01)
Non-resolvable address | 03112233445566 (upper 2 bits of Bluetooth portion ==00)

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

**BleSetAddressType**

**FUNCTION**

This function sets the current address type to be used by the LE radio scan/advert/connection requests. Type 2 and 3 are freshly generated everytime this function is called.

If local IRK not available then no change and an error is returned.

**BLESETADDRESSTYPE(nAddrType)**

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

**Arguments:**

byVal **nAddrType** AS INTEGER.

Specifies the type of the LE address as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Public address, same as Classic.</td>
</tr>
<tr>
<td>1</td>
<td>Random static address, generated first boot.</td>
</tr>
<tr>
<td>2</td>
<td>Random address, resolvable with IRK, generated on call.</td>
</tr>
<tr>
<td>3</td>
<td>Random address, non resolvable, generation on call</td>
</tr>
</tbody>
</table>

**Interactive Command**

No

```
DIM rc
rc = BleSetAddressType(1)
PRINT "\nrc = ";rc
```

**Expected Output:**

```
rc = 0
```

BLESETADDRESSTYPE is an extension function.

**Events and Messages**

**EVBLE_ADV_TIMEOUT**

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

```c
//Example :: EvBle_Adv_Timeout.sb
DIM peerAddr$

//handler to service an advert timeout
FUNCTION HndlrBleAdvTimOut()
    PRINT "\nAdvert stopped via timeout"
ENDFUNC

//start adverts
```

Embedded Wireless Solutions Support Center: http://ews-support.lairdtech.com
www.lairdtech.com/wireless

Americas: +1-800-492-2320
Europe: +44-1628-858-940
Hong Kong: +852 2923 0610
//rc = BleAdvertStart(0,"",100,5000,0)
IF BleAdvertStart(0,peerAddr$,100,2000,0)==0 THEN
    PRINT "\nAdvert Started"
ELSE
    PRINT "\nAdvert not successful"
ENDIF
ONEVENT EVBLE_ADV_TIMEOUT CALL HndlrBleAdvTimOut
WAITEVENT

Expected Output:

Advert Started
Advert stopped via timeout

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

EVBLE_ADV_REPORT
This event is thrown when an advert report is received whether successfully cached or not. See example for BleScanGetAdvReport.

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

EVBLE_SCAN_TIMEOUT
This event is thrown when a BLE scanning procedure initiated by the BleScanStart() function times out. See example for BLESCANSTART.

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

The message contains two parameters:

- msgID – Identifies what event was triggered
- msgCtx – Conveys some context data associated with that event.

The smartBASIC application must register a handler function which takes two integer arguments to be able to receive and process this message.

Note: The messaging subsystem, unlike the event subsystem, has a queue associated with it and, unless that queue is full, pends all messages until they are handled. Only messages that have handlers
associated with them are inserted into the queue. This prevents messages that will not get handled from filling that queue. The following table lists the triggers and associated context parameters.

<table>
<thead>
<tr>
<th>MsgID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A BLE connection is established and msgCtx is the connection handle.</td>
</tr>
<tr>
<td>1</td>
<td>A BLE disconnection event and msgCtx identifies the handle.</td>
</tr>
<tr>
<td>4</td>
<td>A BLE Service Error. The second parameter contains the error code.</td>
</tr>
<tr>
<td>9</td>
<td>Pairing in progress and displayed Passkey supplied in msgCtx.</td>
</tr>
<tr>
<td>10</td>
<td>A new bond has been successfully created.</td>
</tr>
<tr>
<td>11</td>
<td>Pairing in progress and authentication key requested. msgCtx is key type.</td>
</tr>
<tr>
<td>14</td>
<td>Connection parameters update and msgCtx is the conn handle.</td>
</tr>
<tr>
<td>15</td>
<td>Connection parameters update fail and msgCtx is the conn handle.</td>
</tr>
<tr>
<td>16</td>
<td>Connected to a bonded master and msgCtx is the conn handle.</td>
</tr>
<tr>
<td>17</td>
<td>A new pairing has replaced old key for the connection handle specified.</td>
</tr>
<tr>
<td>18</td>
<td>The connection is now encrypted and msgCtx is the conn handle.</td>
</tr>
<tr>
<td>20</td>
<td>The connection is no longer encrypted and msgCtx is the conn handle.</td>
</tr>
<tr>
<td>21</td>
<td>The device name characteristic in the GAP service of the local GATT table has been written by the remote GATT client.</td>
</tr>
</tbody>
</table>

**Note:** Message ID 13 is reserved for future use.

The following is an example of how these messages can be used:

```plaintext
//Example :: EvBleMsg.sb
DIM addr$: addr$=""
DIM rc

//=================================
// This handler is called when there is a BLE message
//=================================
FUNCTION HndlrBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER)
    SELECT nMsgId
    CASE 0
        PRINT "\nBLE Connection ":nCtx
    CASE 1
        PRINT "\nDisconnected":nCtx:"\n"
    CASE 18
        PRINT "\nConnection":nCtx:" is now encrypted"
    CASE 16
        PRINT "\nConnected to a bonded master"
    CASE 17
        PRINT "\nA new pairing has replaced the old key";
    CASE ELSE
        PRINT "\nUnknown Ble Msg"
    ENDSELECT
ENDFUNC

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

Embedded Wireless Solutions Support Center: 55 Americas: +1-800-492-2320
http://ews-support.lairdtech.com Europe: +44-1628-858-940
www.lairdtech.com/wireless Hong Kong: +852 2923 0610
rc=BleAdvertStop()
    PRINT "\nExiting..."
ENDFUNC 0

ONEVENT EVBLEMSG CALL HndlrBleMsg
ONEVENT EVBLE_ADV_TIMEOUT CALL HndlrBleAdvTimOut
ONEVENT EVUARTRX CALL HndlrUartRx

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

Expected Output (When connection made with the module):

Adverts Started
Press any key to exit

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

Expected Output (When no connection made):

Adverts Started
Press any key to exit

Advert stopped via timeout
Exiting...

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

- Connection handle
- The reason for the disconnection.

The reason, for example, can be 0x08 which signifies a link connection supervision timeout which is used in the Proximity Profile.

A full list of Bluetooth HCI result codes for the reason of disconnection is provided in this document [here](#).
WB45 smartBASIC Extensions
User Guide

---

Expected Output:

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

**EVCHARVAL**

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

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

```vbnet
//Example :: EvCharVal.sb
DIM hMyChar,rc,at$,conHndl

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

    //commit service
    rcd=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)

    //initialise char, write/read enabled, accept signed writes
    rcd=BleCharNew(0x0A,BleHandleUuid16(1),BleAttrMetaData(1,1,20,0,rc),0,0)

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

    //commit changes to service
    rcd=BleServiceCommit(hSvc)
```

---

Embedded Wireless Solutions Support Center:

- Americas: +1-800-492-2320
- Europe: +44-1628-858-940
- Hong Kong: +852 2923 0610

http://ews-support.lairdtech.com

www.lairdtech.com/wireless
rc=BleScanRptInit(scRpt$)
// Add 1 service handle to scan report
// rc=BleAdvRptAddUuid16(scRpt$,$0x18EE,-1,-1,-1,-1,-1)
// commit reports to GATT table - adRpt$ is empty
rc=BleAdvRptsCommit(adRpt$,scRpt$)
rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC rc

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

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

FUNCTION HandlerCharVal(BYVAL charHandle, BYVAL offset, BYVAL len)
  DIM s$
  IF charHandle == hMyChar THEN
    PRINT "\n";len;" byte(s) have been written to char value attribute from offset ";offset
    rc=BleCharValueRead(hMyChar,s$)
    PRINT "\nNew Char Value: ";s$
  ENDIF
  CloseConnections()
ENDFUNC

ONEVENT EVCHARVAL CALL HandlerCharVal
ONEVENT EVBLEMSG CALL HndlrBleMsg

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

The characteristic’s value is Hi
Write a new value to the characteristic
--- Connected to client
5 byte(s) have been written to char value attribute from offset 0
New Char Value: Hello
--- Disconnected from client
Exiting...

EVCHARHVC
This event is thrown when a value sent via an indication to a client gets acknowledged. It comes with one parameter:

- The characteristic handle that was returned when the characteristic was registered using the function `BleCharCommit()`

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

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

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

```c
//Example :: EvCharCccd.sb
DIM hMyChar, rc, at$, conHndl
//==============================================================================
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
FUNCTION OnStartup()

DIM rc, hSvc, metaSuccess, at$, attr$, adRpt$, addr$, scRpt$
attr$="Hi"
DIM svcUuid : svcUuid=0x18EE
DIM charUid : charUid = BleHandleUuid16(1)
DIM charMet : charMet = BleAttrMetaData(0,0,20,1,metaSuccess)
DIM hSvcUuid : hSvcUuid = BleHandleUuid16(svcUuid)
DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc)    //CCCD metadata for char

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

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

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

//@commit service to GATT table
rc=BleServiceCommit(hSvc)
```
rc=BleAdvertStart(0,0x7a,20,0,0)
ENDFUNC rc

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

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

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

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

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

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nValue of the characteristic \";hMyChar;\" is: \";at$
    PRINT "\nYou can write to the CCCD characteristic."
    PRINT "\nThe WB45 will then indicate a new characteristic value\n"
    PRINT "\n--- Press any key to exit"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

CloseConnections()

PRINT "\nExiting..."

Expected Output:

Value of the characteristic 1346437121 is: Hi
You can write to the CCCD characteristic.
The WB45 will then indicate a new characteristic value
--- Press any key to exit
--- Connected to client
Indications have been enabled by client
Got confirmation of recent indication
Exiting...

EVCHARSCCD

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

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

The SCCD is used to manage broadcasts of characteristic values.

//Example :: EvCharSccd.sb
DIM hMyChar,rc,chVal$,conHndl

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

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

    //Register characteristic
    attr$=BleAttrRegister(hSvc,1,1,20,rc,adRpt$,attr$)
// initialise broadcast capable, readable, writeable
rc=BleCharNew(0x0B,BleHandleUuid16(1),charMet,0,BleAttrMetadata(1,1,1,0,rc2))

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

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

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

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

// Broadcast characteristic value
FUNCTION PrepAdvReport()
    dim adRpt$, scRpt$, svcDta$

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

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

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

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

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

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

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

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

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

DIM addr$
rc=BleAdvertStart(0,addr$,20,300000,0)
IF rc==0 THEN
  PRINT "\nYou should now see the new characteristic value in the advertisement data"
ENDIF
ELSEIF nMsgID==0 THEN
  PRINT "\n--- Connected to client"
ENDIF
ENDFUNC

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

// CCCD descriptor written handler
FUNCTION HndlrCharSccd(BYVAL charHandle, BYVAL nVal) AS INTEGER

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

// Called after a disconnection
FUNCTION HndlrDiscon(hConn, nRsn) ENDFUNC 1

DIM s$
IF charHandle == hMyChar THEN
  rc=BleCharValueRead(hMyChar,chVal$)
  PRINT "\nNew Char Value: ";chVal$
ENDIF

// New char value handler
FUNCTION HndlrCharVal(BYVAL charHandle, BYVAL offset, BYVAL len) ENDFUNC 1

DIM addr$
Expected Output:

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

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

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

EVCHARDESC

This event is thrown when the client writes to writable descriptor of a characteristic which is not a CCCD or SCCD as they are catered for with their own dedicated messages. It comes with two parameters, the first is the characteristic handle that was returned when the characteristic was registered using the function BleCharCommit() and the second is an index into an opaque array of handles managed inside the characteristic handle. Both parameters are supplied as-is as the first two parameters to the function BleCharDescRead().

//Example :: EvCharDesc.sb
DIM hMyChar, rc, at$, conHndl, hOtherDescr

// Initialise and instantiate service, characteristic, start adverts

FUNCTION OnStartup$(())
    DIM rc, hSvc, at$, adRpt$, addr$, scRpt$, hOtherDescr, attr$, attr2$, rc2
    attr$="Hi"

DIM charMet : charMet = BleAttrMetaData(1,0,20,0,rc)

//Commit svc with handle 'hSvcUuid'
rc=BleServiceNew(1,BleHandleUuid16(0x18EE),hSvc)

//initialise characteristic - readable
rc=BleCharNew(0x02,BleHandleUuid16(1),charMet,0,0)

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

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

//commit service to GATT table
rc=BleServiceCommit(hSvc)
rc=BleAdvertStart(0,addr$,20,300000,0)

ENDFUNC attr$

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

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

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

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

Other Descriptor Value: my char desc
Write a new value
--- Press any key to exit
--- Connected to client
New value for descriptor 0 with uuid FE012901 is hello

EVNOTIFYBUF

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

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

Note: When sending data using Indications, this event is not thrown because those messages have to be confirmed by the client which results in an `EVCHARHVC` message to the smartBASIC application. Likewise, writes which are acknowledged also do not consume these buffers.
rc=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
// initialise char, write/read enabled, accept signed writes, notifiable
rc=BleCharNew(0x12,BleHandleUuid16(1),BleAttrMetaData(1,0,20,0,rc),mdCccd,0)
// commit char initialised above, with initial value "hi" to service 'hMyChar'
rc=BleCharCommit(hSvc,attr$,hMyChar)
// commit changes to service
rc=BleServiceCommit(hSvc)
rc=BleScanRptInit(scRpt$)
// Add 1 service handle to scan report
rc=BleAdvRptAddUuid16(scRpt$,0x18EE,-1,-1,-1,-1)
// commit reports to GATT table - adRpt$ is empty
rc=BleAdvRptsCommit(adRpt$,scRpt$)
rc=BleAdvertStart(0,addr$,50,0,0)
ENDFUNC rc

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

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

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

FUNCTION HndlrNtfyBuf()
   sendData()
ENDFUNC 0

FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
DIM value$, tx$
IF charHandle==hMyChar THEN
  IF nVal THEN
    PRINT " : Notifications have been enabled by client"
    ntfyEnabled=1
    tx$="Hello"
    rc=BleCharValueNotify(hMyChar,tx$)
  ELSE
    PRINT " : Notifications have been disabled by client"
    ntfyEnabled=0
  ENDIF
ELSE
  PRINT " : This is for some other characteristic"
ENDIF
ENDFUNC

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

IF OnStartup()==0 THEN
  rc = BleCharValueRead(hMyChar,at$)
  PRINT " : You can connect and write to the CCCD characteristic."
  PRINT " : The WB45 will then send you data until buffer is full"
ELSE
  PRINT " : Failure OnStartup"
ENDIF
WAITEVENT

CloseConnections()
PRINT " : Exiting..."
use the command AT I 2008. Although this function can accept any value between 10 and -20, the actual transmit power is determined by the internal power table which supports -20, -16, -12, -8, -4, 0, 4 and 8 dBm, when a value is set the highest transmit power that is less than or equal to the desired power is used. SYSINFO(2008) and AT I 2008 will return the power level set, and does not reflect the transmit power level of the radio itself.

**BLETXPOWERSET(nTxPower)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments:</strong></td>
<td></td>
</tr>
<tr>
<td>nTxPower</td>
<td>byVal nTxPower AS INTEGER. Specifies the new transmit power in dBm units to be used for all subsequent tx packets. The actual value is determined by the radios internal power table.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

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

**Expected Output:**

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

BLETXPOWERSET is an extension function.

**BleGetConnHandleFromAddr**

**FUNCTION**

This function is used to get the connection handle from a specified Bluetooth address.
**BLEGETCONNHANDLEFROMADDR(macAddrBE$, nConnHandle)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments:</strong></td>
<td></td>
</tr>
</tbody>
</table>
| **macAddrBE$** | byRef *macAddrBE$ AS STRING.*  
The Bluetooth address of the connected remote device. |
| **nConnHandle** | byRef *nConnHandle AS INTEGER.*  
Returned connection handle. |
| **Interactive Command** | No |

```
DIM rc, periphAddr$

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

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

'//This handler will be called when an advert is received  
FUNCTION HndlrAdvRpt()  
  DIM advData$, nDiscarded, nRssi  
  '//Read an advert report and connect to the sender  
  rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)  
  rc=BleScanStop()
  '//Connect to device with MAC address obtained above with 5s connection timeout,  
  '//20ms min connection interval, 75 max, 5 second supervision timeout.  
  rc=BleConnect(periphAddr$, 5000, 20000, 75000, 5000000)  
  IF rc==0 THEN  
    PRINT "n--- Connecting"
  ELSE  
    PRINT "nError: "; INTEGER.H'rc
  ENDIF
ENDFUNC

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

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

**BleGetAddrFromConnHandle**

**FUNCTION**

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

**BLEGETADDRFROMCONNHANDLE(nConnHandle, macAddrBE$)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nConnHandle</td>
<td>byRef nConnHandle AS INTEGER. Connection handle from which to get Bluetooth address</td>
</tr>
<tr>
<td>macAddrBE$</td>
<td>byRef macAddrBE$ AS STRING. Returned Bluetooth address.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```plaintext
DIM rc, periphAddr$

'//Scan indefinitely
rc=BleScanStart(0, 0)
IF rc==0 THEN
 PRINT "\nScanning"
ELSE
 PRINT "\nError: "; INTEGER.H'rc
ENDIF

'//This handler will be called when an advert is received
FUNCTION HndlrAdvRpt()
  DIM advData$, nDiscarded, nRssi
  //Read an advert report and connect to the sender
  rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)
  rc=BleScanStop()
```
'//Connect to device with MAC address obtained above with 5s connection timeout,
'//20ms min connection interval, 75 max, 5 second supervision timeout.
rc=BleConnect (periphAddr$, 5000, 20000, 75000, 5000000)
IF rc==0 THEN
    PRINT "\n--- Connecting"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF
ENDFUNC 1

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

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

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

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

Expected Output:

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

BLEGETADDRFROMCONNHANDLE is an extension function.

Advertising Functions

This section describes all the advertising-related routines.

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

Each AD record consists of up to three fields:
• Field 1 – One octet in length and indicates the number of octets that follow it that belong to that record.
• Field 2 – One octet in length and is a tag value which identifies the type of payload that starts at the next octet. Hence the payload data is ‘length – 1’.
• Field 3 – A special NULL AD record that consists of one field (the length field) when it contains only the 00 value.

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

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

**BleAdvertStart**

**FUNCTION**

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

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

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

**Note:** Whitelist functionality is currently not supported and will be implemented in future releases of the firmware.

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

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

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation. If a 0x6A01 resultcode is received, it implies a whitelist has been enabled but the Flags AD in the advertising report is set for Limited and/or General Discoverability. The solution is to resubmit a new advert report which is made up so that the nFlags argument to BleAdvRptInit() function is 0. The BT 4.0 spec disallows discoverability when a whitelist is enabled during advertisement see Volume 3, Sections 9.2.3.2 and 9.2.4.2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td><strong>nAdvType</strong></td>
<td>byVal nAdvType AS INTEGER. Specifies the advertisement type as follows:</td>
</tr>
<tr>
<td>0</td>
<td>ADV_IND</td>
</tr>
<tr>
<td>1</td>
<td>ADV_DIRECT_IND</td>
</tr>
<tr>
<td>2</td>
<td>ADV_SCAN_IND</td>
</tr>
<tr>
<td>3</td>
<td>ADV_NONCONN_IND</td>
</tr>
<tr>
<td><strong>peerAddr$</strong></td>
<td>byRef peerAddr$ AS STRING. It can be an empty string that is omitted if the advertisement type is not ADV_DIRECT_IND. This is only required when nAdvType == 1. When not empty, a valid address string is exactly</td>
</tr>
</tbody>
</table>

**Embedded Wireless Solutions Support Center:**
- Americas: +1-800-492-2320
- Europe: +44-1628-858-940
- Hong Kong: +852 2923 0610

[http://ews-support.lairdtech.com](http://ews-support.lairdtech.com)
[www.lairdtech.com/wireless](http://www.lairdtech.com/wireless)
seven octets long (for example: \x00\x11\x22\x33\x44\x55\x66) where the first octet is the address type and the rest of the six octets is the usual Bluetooth address in big endian format (so the most significant octet of the address is at offset 1), whether public or random.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Public</td>
</tr>
<tr>
<td>1</td>
<td>Random Static</td>
</tr>
<tr>
<td>2</td>
<td>Random Private Resolvable</td>
</tr>
<tr>
<td>3</td>
<td>Random Private Non-Resolvable</td>
</tr>
</tbody>
</table>

All other values are illegal.

**nAdvInterval**

ByVal nAdvInterval AS INTEGER.

The interval between two advertisement events (in milliseconds). An advertisement event consists of a total of three packets being transmitted in the three advertising channels. The range of this interval is between 20 and 10240 milliseconds.

**nAdvTimeout**

ByVal nAdvTimeout AS INTEGER.

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

A value of 0 means disable the timeout, but note that if limited advert modes was specified in BleAdvRptInit() then this function fails. When the advert type specified is ADV_DIRECT_IND, the timeout is automatically set to 1280 ms as per the Bluetooth Specification.

**WARNING:** To save power, do not mistakenly set this to e.g. 100ms.

**nFilterPolicy**

ByVal nFilterPolicy AS INTEGER.

Specifies the filter policy for the whitelist as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Filter Policy – Any</td>
</tr>
<tr>
<td>1</td>
<td>Filter Policy – Filter scan request; allow connection request from any</td>
</tr>
<tr>
<td>2</td>
<td>Filter Policy – Filter connection request; allow scan request from any</td>
</tr>
</tbody>
</table>

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

//Example :: BleAdvertStart.sb
DIM addr$ : addr$=""

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

//The advertising interval is set to 25 milliseconds. The module will stop
//advertising after 60000 ms (1 minute)
IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "{nAdverts Started"
    PRINT "{nIf you search for bluetooth devices on your device, you should see
'Laird WB45'"
ELSE
    PRINT "{n\n
ENDIF

ONEVENT EVBLE_ADV_TIMEOUT CALL HndlrBlrAdvTimOut
WAITEVENT
Expected Output:

Adverts Started
If you search for bluetooth devices on your device, you should see 'Laird WB45'
Advert stopped via timeout
Exiting...

BLEADVERTSTART is an extension function.

BleAdvertStop
FUNCTION
This function causes the BLE module to stop advertising.

BLEADVERTSTOP ()

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Interactive</td>
<td>No</td>
</tr>
</tbody>
</table>

BLEADVERTSTOP is an extension function.

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

The following lists the default values for the parameters:

<table>
<thead>
<tr>
<th>Advert Channel Mask</th>
<th>Bit field detailing the channels to advertise on.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note:</td>
<td>Set channel mask Bit 0 to enable advert channel 0, Bit 1 to enable advert channel 1, and Bit 2 to enable advert channel 2.</td>
</tr>
</tbody>
</table>

BLEADVERTCONFIG (configID,configValue)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>configID</td>
<td>byVal configID AS INTEGER. This identifies the value to update as follows:</td>
</tr>
<tr>
<td></td>
<td>0 Unused</td>
</tr>
<tr>
<td></td>
<td>1 Unused</td>
</tr>
<tr>
<td></td>
<td>2 Unused</td>
</tr>
<tr>
<td></td>
<td>3 Advert Channel Mask</td>
</tr>
<tr>
<td>configValue</td>
<td>byVal configValue AS INTEGER. This contains the new value to set in the parameters indentified by configID.</td>
</tr>
<tr>
<td>Interactive</td>
<td>No</td>
</tr>
</tbody>
</table>
Command

BLEADVERTCONFIG is an extension function.

BleAdvRptInit

FUNCTION

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

This report is for use with advertisement packets.

BLEADVPTSCOMMIT(advRpt$, nFlagsAD, nAdvAppearance, nMaxDevName)

Returns

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

Arguments:

advRpt$ byRef advRpt$ AS STRING.
This contains an advertisement report.

nFlagsAD byVal nFlagsAD AS INTEGER.
Specifies the flags AD bits where bit 0 is set for limited discoverability and bit 1 is set for general discoverability. Bit 2 will be forced to 1 and bits 3 & 4 will be forced to 0. Bits 3 to 7 are reserved for future use by the BT SIG and must be set to 0.

nAdvAppearance byVal nAdvAppearance AS INTEGER.
Determines whether the appearance advert should be added or omitted as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Omit appearance advert</td>
</tr>
<tr>
<td>1</td>
<td>Add appearance advert as specified in the GAP service which is supplied via the BleGapSvcInit() function</td>
</tr>
</tbody>
</table>

nMaxDevName byVal nMaxDevName AS INTEGER.
The n leftmost characters of the device name specified in the GAP service. If this value is set to zero (0) then the device name is not included.

Interactive Command

No

//Example :: BleAdvRptInit.sb
DIM advRpt$ : advRpt$=""
DIM discovMode : discovMode=0
DIM advAppearance : advAppearance = 1
DIM maxDevName : maxDevName = 10

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

Expected Output:

Advert report initialised

BLEADVPTSCOMMIT is an extension function.

BleScanRptInit

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

This report is for use with SCAN_RESPONSE packets.

**BLESCANRPTINIT(scanRpt)**

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

Arguments:

- `scanRpt` byRef scanRpt AS STRING.
  This contains a scan report.

Interactive Command: No

//Example :: BleScanRptInit.sb

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

Expected Output:

Scan report initialised

**BLESCANRPTINIT** is an extension function.

**BleAdvRptGetSpace**

FUNCTION

This function returns the free space in the advert advRpt$

**BLEADVRPTGETSPACE(advRpt$)**

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

Arguments:

- `advRpt$` byRef advRpt$ AS STRING.
  This contains an advert/scan report.

Interactive Command: No

```vbnet
dim rc, s$, dn$
rc=BleScanRptInit(s$)
dn$ = BleGetDeviceName$()
'//Add device name to scan report
rc=BleAdvRptAppendAD(s$,0x09,dn$)
print "\nFree space in scan report: "; BleAdvRptGetSpace(s$); " bytes"
```

Expected Output:

Free space in scan report: 18 bytes

**BLESCANRPTINIT** is an extension function.
BLEAdvRptAddUuid16

FUNCTION

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

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

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

Arguments:

- **advRpt**: byRef AdvRpt AS STRING. The advert report onto which the 16-bit uuids AD record is added.
- **Uuid1**: byVal uuid1 AS INTEGER. UUID in the range 0 to FFFF; if the value is outside that range, it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
- **Uuid2**: byVal uuid2 AS INTEGER. UUID in the range 0 to FFFF; if the value is outside that range, it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
- **Uuid3**: byVal uuid3 AS INTEGER. UUID in the range 0 to FFFF; if the value is outside that range, it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
- **Uuid4**: byVal uuid4 AS INTEGER. UUID in the range 0 to FFFF; if the value is outside that range, it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
- **Uuid5**: byVal uuid5 AS INTEGER. UUID in the range 0 to FFFF; if the value is outside that range, it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
- **Uuid6**: byVal uuid6 AS INTEGER. UUID in the range 0 to FFFF; if the value is outside that range, it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.

Interactive Command: No

//Example :: BleAdvAddUuid16.sb
DIM advRpt$, rc
DIM discovMode : discovMode=0
DIM advAppearance : advAppearance = 1
DIM maxDevName : maxDevName = 10
rc = BleAdvRptInit(advRpt$, discovMode, advAppearance, maxDevName)

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

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

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

Expected Output:

| UUID Service List AD added |

BLEADVRPTADDUUID16 is an extension function.
**BleAdvRptAddUuid128**

**FUNCTION**

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

**BLEADVRIPTADDDUUID128 (advRpt, nUuidHandle)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>advRpt</td>
<td>byRef AdvRpt AS STRING. The advert report into which the 128-bit UUID AD record is to be added.</td>
</tr>
<tr>
<td>nUuidHandle</td>
<td>byVal nUuidHandle AS INTEGER This is handle to a 128-bit UUID which was obtained using a function such as BleHandleUuid128() or some other function which returns one.</td>
</tr>
</tbody>
</table>

**Interactive Command**

```basic
//Example :: BleAdvAddUuid128.sb
DIM uuid$, hUuidCustom
DIM tx$, scRpt$, adRpt$, addr$, hndl
scRpt$=""
PRINT BleScanRptInit(scRpt$)

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

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

**Expected Output:**

```
00000
```

BLEADVRIPTADDDUUID128 is an extension function.

**BleAdvRptAppendAD**

**FUNCTION**

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

**BLEADVRIPTAPPENDAD (advRpt, nTag, stData$)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>advRpt</td>
<td>byRef AdvRpt AS STRING.</td>
</tr>
</tbody>
</table>
The advert report onto which the AD record is to be appended.

### nTag

**byVal nTag AS INTEGER**

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

### stData$

**byRef stData$ AS STRING**

This is an octet string which can be 0 bytes long. The maximum length is governed by the space available in AdvRpt, a maximum of 31 bytes long.

### Interactive Command

No

---

```plaintext
//Example :: BleAdvRptAppendAD.sb
DIM scnRpt$,ad$
ad$="\01\02\03\04"
PRINT BleScanRptInit(scnRpt$)
IF BleAdvRptAppendAD(scnRpt$,0x31,ad$)==0 THEN //6 bytes will be used up in the report
   PRINT "\nAD with data \\
";ad$;
ENDIF
```

**Expected Output:**

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

BLEADVRPRTAPPENDAD is an extension function.

### BleAdvRptsCommit

**FUNCTION**

This function is used to commit one or both advert reports. If the string is empty then that report type is not updated. Both strings can be empty and in that case this call will have no effect.

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

**BLEADVRPRTSCOMMIT(advRpt, scanRpt)**

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

**Arguments:**

<table>
<thead>
<tr>
<th>advRpt</th>
<th>byRef advRpt AS STRING.</th>
</tr>
</thead>
<tbody>
<tr>
<td>scanRpt</td>
<td>byRef scanRpt AS STRING.</td>
</tr>
</tbody>
</table>

**Interactive Command**  No

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

**Tip:**  You can commit advert reports to update your advertisement data while advertising.
// Example :: BleAdvRptsCommit.sb
DIM advRpt$ : advRpt$=""
DIM scRpt$ : scRpt$=""
DIM discovMode : discovMode = 0
DIM advApprnce : advApprnce = 1
DIM maxDevName : maxDevName = 10
PRINT BleAdvRptInit(advRpt$, discovMode, advApprnce, maxDevName)
PRINT BleAdvRptAddUuid16(advRpt$, 0x180F, 0x180A, -1, -1, -1, -1)
PRINT BleAdvRptsCommit(advRpt$, scRpt$)
// Only the advert report will be updated.

Expected Output:

000

BLEADVRPTSCOMMIT is an extension function.

**Scanning Functions**

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

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

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

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

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

For more information about adverts see the section Advertising Functions.

**BleScanStart**

**FUNCTION**

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

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVBLE_SCAN_TIMEOUT</td>
<td>End of scanning</td>
</tr>
<tr>
<td>EVBLE_ADV_REPORT</td>
<td>Advert report received</td>
</tr>
<tr>
<td>EVBLE_FAST_PAGED</td>
<td>Peripheral inviting a connection to this module</td>
</tr>
</tbody>
</table>
- **EVBLE_ADV_REPORT** – Received when an advert has been successfully cached in a ring buffer. The handler should call the function `BleScanGetAdvReport()` repeatedly to read all the advert reports that have been cached until the cache is empty, otherwise there is a risk that advert reports will be discarded. The output parameter `nDiscarded` returns the number of discarded reports, if any.

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

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

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

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

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

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

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

---

### BLESCANSTART (scanTimeoutMs, nFilterHandle)

**Returns**

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

**Arguments**

- **byVAL scanTimeoutMs AS INTEGER.**
  The length of time in milliseconds the scan for adverts lasts. If the timer times out then the event EVBLE_SCAN_TIMEOUT is thrown to the `smartBASIC` application. Valid range is 0 to 65535000 milliseconds (about 18 hours). If 0 is supplied, a timer is not started and scanning can only be stopped by calling either `BleScanAbort()` or `BleScanStop()`.

- **byVAL nFilterHandle AS INTEGER**
  This must be zero (0) to specify no filtering of adverts.
  **Note:** In this current firmware version, this is only a placeholder.

**Interactive Command**

No

---

```plaintext
//Example :: BleScanStart.sb
DIM rc

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

**FUNCTION**

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

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

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

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

**BLESCANABORT()**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```plaintext
//Example :: BleScanAbort.sb
DIM rc, startTick

'//Scan for 20 seconds with no filtering
rc = BleScanStart(20000, 0)
```
`IF rc==0 THEN
    PRINT "\nScanning"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF

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

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

Expected Output:

Scanning
Aborting scan
Scan aborted

BLESCANABORT is an extension function.

**BleScanStop**

**FUNCTION**

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

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

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

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

<table>
<thead>
<tr>
<th>BLESCANSTOP()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Returns</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
</tr>
<tr>
<td><strong>Interactive Command</strong></td>
</tr>
</tbody>
</table>

//Example :: BleScanStop.sb
DIM rc, startTick
BLESCANSTOP is an extension function.

**BleScanFlush**

**FUNCTION**

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

**BLESCANFlush()**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```
DIM rc, startTick

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

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

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

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

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

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

Expected Output:

Scanning
Aborting scan
Scan aborted
Scan results flushed.

BLESCANFLUSH is an extension function.

BleScanConfig

FUNCTION

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

The following lists the default values for the parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan Interval</td>
<td>80 milliseconds</td>
</tr>
<tr>
<td>Scan Window</td>
<td>40 milliseconds</td>
</tr>
<tr>
<td>Scan Type (Active/Passive)</td>
<td>Active</td>
</tr>
<tr>
<td>Minimum Reports in Cache</td>
<td>4</td>
</tr>
</tbody>
</table>

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

BLESCANCONFIG (configID,configValue)

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

Arguments:

<table>
<thead>
<tr>
<th>configID</th>
<th>VAL configID AS INTEGER.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Scan Interval in milliseconds (range 0..10240)</td>
</tr>
<tr>
<td>1</td>
<td>Scan Window in milliseconds (range 0..10240)</td>
</tr>
<tr>
<td>2</td>
<td>Scan Type (0=Passive, 1=Active)</td>
</tr>
<tr>
<td>3</td>
<td>Advert Report Cache Size</td>
</tr>
</tbody>
</table>

For all other configID values the function returns an error.
**configValue** byVal **configValue** AS INTEGER.
This contains the new value to set in the parameters indentified by configID.

<table>
<thead>
<tr>
<th>Interactive Command</th>
<th>Yes</th>
</tr>
</thead>
</table>

```
//Example :: BleScanConfig.sb
DIM rc, startTick
PRINT "\nScan Interval: "; SysInfo(2150) //get current scan interval
PRINT "\nScan Window: "; SysInfo(2151) //get current scan window
PRINT "\nScan Type: ";
IF SysInfo(2152)==0 THEN //get current scan type
    PRINT "Passive"
ELSE
    PRINT "Active"
ENDIF
PRINT "\nReport Cache Size: "; SysInfo(2153) //get report cache size
PRINT "\n--- New Parameters:"
rc = BleScanConfig(0, 100) //set scan interval to 100
rc = BleScanConfig(1, 50) //set scan window to 50
rc = BleScanConfig(2, 0) //set scan type to passive
rc = BleScanConfig(3, 3) //set report cache size
PRINT "\n--- New Parameters:"
PRINT "\nScan Interval: "; SysInfo(2150) //get current scan interval
PRINT "\nScan Window: "; SysInfo(2151) //get current scan window
PRINT "\nScan Type: ";
IF SysInfo(2152)==0 THEN //get current scan type
    PRINT "Passive"
ELSE
    PRINT "Active"
ENDIF
PRINT "\nReport Cache Size: "; SysInfo(2153) //get report cache size
```

Expected Output:

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

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

BLESCANCONFIG is an extension function.

**BleScanGetAdvReport**

**FUNCTION**

When a scan is in progress after having called BleScanStart() for each advert report, the information is cached in a queue buffer and an EVBLE_ADV_REPORT event is thrown to the smartBASIC application.
This function is used by the smartBASIC application to extract it from the queue for further processing in the handler for the EVBLE_ADV_REPORT event.

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

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

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

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>periphAddr$</td>
<td>byREF periphAddr$ AS STRING On return, this parameter is updated with the address of the peripheral that sent the advert.</td>
</tr>
<tr>
<td>advData$</td>
<td>byREF advData $ AS STRING On return, this parameter is updated with the data payload of the advert which consists of multiple AD elements.</td>
</tr>
<tr>
<td>nDiscarded</td>
<td>byREF nDiscarded AS INTEGER On return, this parameter is updated with the number of adverts that were discarded because there was no space in the internal queue.</td>
</tr>
<tr>
<td>nRssi</td>
<td>byREF nRssi AS INTEGER On return, this parameter is updated with the RSSI as reported by the stack for that advert. <strong>Note:</strong> This is NOT a value that is sent by the peripheral but a value that is calculated by the receiver in this module.</td>
</tr>
</tbody>
</table>

**Interactive Command**
No

**Note:** This code snippet was tested with another WB45 running the iBeacon app (see in smartBASIC_Sample_Apps folder) on peripheral firmware.

```plaintext
//Example :: BleScanGetAdvReport.sb
DIM rc

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

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

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

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

    '//Read all cached advert reports
    rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)

    WHILE (rc == 0)
        PRINT "\n\nPeer Address: "; StrHexize$(periphAddr$
        PRINT "\nAdvert Data: ";StrHexize$(advData$
        PRINT "\nNo. Discarded Adverts: ";nDiscarded
        PRINT "\nRSSI: ";nRssi

        rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)
    ENDWHILE

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

ONEVENT EVBLE_SCAN_TIMEOUT CALL HndlrScanTO
ONEVENT EVBLE_ADV_REPORT CALL HndlrAdvRpt

WAITEVENT

Expected Output:

<table>
<thead>
<tr>
<th>Scanning</th>
<th>Peer Address: 01D8CFCF14498D</th>
<th>Advert Data: 0201061AFF4C000215E2C56DB5DFFB48D2B060D0F5A71096E012345678C4</th>
<th>No. Discarded Adverts: 0</th>
<th>RSSI: -97</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Peer Address: 01D8CFCF14498D</td>
<td>Advert Data: 0201061AFF4C000215E2C56DB5DFFB48D2B060D0F5A71096E012345678C4</td>
<td>No. Discarded Adverts: 0</td>
<td>RSSI: -97</td>
</tr>
<tr>
<td></td>
<td>--- No more adverts in cache</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Peer Address: 01D8CFCF14498D</td>
<td>Advert Data: 0201061AFF4C000215E2C56DB5DFFB48D2B060D0F5A71096E012345678C4</td>
<td>No. Discarded Adverts: 0</td>
<td>RSSI: -92</td>
</tr>
<tr>
<td></td>
<td>--- No more adverts in cache</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BLESCANGETADVREPORT is an extension function.

**BleGetADbyIndex**

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

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

**BLEGETADBYINDEX (nIndex, rptData$, nADTag, ADval$)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| nIndex | byVAL nIndex AS INTEGER  
This is a zero-based index of the AD element that is copied into the output data parameter ADval$. |
| rptData$ | byREF rptData$ AS STRING.  
This parameter is a string that contains concatenated AD elements which were either constructed for an outgoing advert or were received in a scan (depends on module variant). |
| nADTag | byREF nADTag AS INTEGER  
When the nth index is found, the single byte tag value for that AD element is returned in this parameter |
| ADval$ | byREF ADval$ AS STRING  
When the nth index is found, the data excluding single byte the tag value for that AD element is returned in this parameter. |

**Interactive Command** No

```plaintext
//Example :: BleAdvGetADbyIndex.sb
DIM rc, ad1$, ad2$, fullAD$, nADTag, ADval$

`//AD with length = 6 bytes, tag = 0xDD
ad1$="06\DDD\12\23\34\55"

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

fullAD$ = ad1$ + ad2$
PRINT "\n\n"; Strhexize$(fullAD$);
rc=BleGetADbyIndex(0, fullAD$, nADTag, ADval$)
IF rc==0 THEN
   PRINT "\nFirst AD element with tag 0x"; INTEGER.H'nADTag ;" is ";StrHexize$(ADval$)
ELSE
   PRINT "\nError reading AD: "; INTEGER.H'rc
ENDIF
rc=BleGetADbyIndex(1, fullAD$, nADTag, ADval$)
IF rc==0 THEN
   PRINT "\nSecond AD element with tag 0x"; INTEGER.H'nADTag ;" is ";StrHexize$(ADval$)
ELSE
   PRINT "\nError reading AD: "; INTEGER.H'rc
ENDIF

`//Will fail because there are only 2 AD elements
```
BLEGETADBYINDEX is an extension function.

**BleGetADbyTag**

**FUNCTION**

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

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

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

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td><code>rptData$</code></td>
<td><code>byREF</code> <code>rptData$</code> <code>AS</code> <code>STRING</code>. This parameter is a string that contains concatenated AD elements which were either constructed for an outgoing advert or were received in a scan (depends on module variant).</td>
</tr>
<tr>
<td><code>nADTag</code></td>
<td><code>byVAL</code> <code>nADTag</code> <code>AS</code> <code>INTEGER</code>. This parameter specifies the single byte tag value for the AD element that is to returned in the ADval$ parameter. Only the first instance can be catered for. If multiple instances are suspected, then use BleAdvADbyIndex() to extract it.</td>
</tr>
<tr>
<td><code>ADval$</code></td>
<td><code>byREF</code> <code>ADval$</code> <code>AS</code> <code>STRING</code>. When the nth index is found, the data excluding single byte the tag value for that AT element is returned in this parameter.</td>
</tr>
</tbody>
</table>

**Interactive Command**

No

```plaintext
//Example :: BleAdvGetADbyIndex.sb
DIM rc, ad1$, ad2$, fullAD$, nADTag, ADval$

'//AD with length = 6 bytes, tag = 0xDD
ad1$="06\DD\12\22\33\44\55"

'//AD with length = 7 bytes, tag = 0xDA
ad2$="07\EE\AA\BB\CC\DD\EE\FF"
```
FULLAD$ = ad1$ + ad2$
PRINT "\n\n"; Strhexize$(fullAD$);"\n"

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

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

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

Expected Output:

06DD112233445507EEAABBCCDDEEFF
AD element with tag 0x000000DD is 1122334455
AD element with tag 0x000000EE is AABBCDDEEFF
Error reading AD: 00006060

BLEGETADBYTAG is an extension function.

BleScanGetPagerAddr
FUNCTION

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

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

BLESCANGETPAGERADDR (periphAddr$, nRssi)

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

Arguments:

periphAddr$ byREF periphAddr$ AS STRING
On return, this parameter is updated with the address of the peripheral that sent the advert.
**WB45 smartBASIC Extensions**

**User Guide**

| Interactive Command | No |

**byREF nRssi AS INTEGER**

On return, this parameter is updated with the RSSI as reported by the stack for that advert. **Note:** This is NOT a value that is sent by the peripheral but a value that is calculated by the receiver in this module.

```plaintext
//Example :: BleScanGetPagerAddr.sb
DIM rc

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

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

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

'//This handler will be called when an advert is received requesting a connection to this module
FUNCTION HndlrFastPaged()
   DIM periphAddr$, nRssi
   rc = BleScanGetPagerAddr(periphAddr$, nRssi)
   PRINT "\nAdvert received from peripheral "; StrHexize$(periphAddr$); " with RSSI ";nRssi
   PRINT "\nrequesting a connection to this module"
   rc = BleScanStop()
ENDFUNC 0

ONEVENT EVBLE_SCAN_TIMEOUT CALL HndlrScanTO
ONEVENT EVBLE_FAST_PAGED CALL HndlrFastPaged
WAITEVENT
```

**Expected Output:**

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

BLESCANGETPAGERADDR is an extension function.

**Connection Functions**

This section describes all the connection manager-related routines.

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

**Events and Messages**

*Embedded Wireless Solutions Support Center:*

- **93 Americas:** +1-800-492-2320
- **Europe:** +44-1628-858-940
- **Hong Kong:** +852 2923 0610

[http://ews-support.lairdtech.com](http://ews-support.lairdtech.com)

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

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

**BleConnect**

**FUNCTION**

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

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

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

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

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

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

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

**Interactive Command** No

//Example :: BleConnect.sb
DIM rc, periphAddr$
'//Scan indefinitely
rc=BleScanStart(0, 0)

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

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

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

    '//Connect to device with Bluetooth address obtained above with 5s connection timeout,
    '//20ms min connection interval, 75 max, 5 second supervision timeout.
rc=BleConnect(periphAddr$, 5000, 20000, 75000, 5000000)

    IF rc==0 THEN
        PRINT "\n--- Connecting"
    ELSE
        PRINT "\nError: "; INTEGER.H'rc
    ENDIF
ENDFUNC 1

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

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

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

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

WAITEVENT
BLECONNECT is an extension function.

**BleConnectCancel**

**FUNCTION**

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

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

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

**BLECONNECTCANCEL ()**

- **Returns**: INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
- **Arguments**: None
- **Interactive Command**: No

```plaintext
//Example :: BleConnectCancel.sb
DIM rc, periphAddr$

'//Scan indefinitely
rc=BleScanStart(0, 0)
IF rc==0 THEN
  PRINT "\nScanning"
ELSE
  PRINT "\nError: "; INTEGER.H'rc
ENDIF

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

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

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

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

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

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

ONEVENT EVBLE_ADV_REPORT CALL HndlrAdvRpt

WAITEVENT

Expected Output:

<table>
<thead>
<tr>
<th>Scanning</th>
</tr>
</thead>
<tbody>
<tr>
<td>--- Connecting</td>
</tr>
<tr>
<td>Cancel</td>
</tr>
<tr>
<td>--- Connection attempt cancelled</td>
</tr>
</tbody>
</table>

BLECONNECTCANCEL is an extension function.

**BleConnectConfig**

**FUNCTION**

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

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

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

The following are the default settings for these parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-link Connection Interval Periodicity</td>
<td>30 milliseconds</td>
</tr>
<tr>
<td>Scan Interval</td>
<td>120 milliseconds</td>
</tr>
<tr>
<td>Scan Window</td>
<td>60 milliseconds</td>
</tr>
<tr>
<td>Slave Latency</td>
<td>0</td>
</tr>
</tbody>
</table>

**Note:**

- The Scan Window and Interval are multiple integers of the periodicity (although not required to be). The scanning has a 50% duty cycle. The 50% duty cycle attempts to ensure that connection events for existing connections are missed as infrequently as possible.
The Scan Window and Interval are internally stored in units of 0.625 milliseconds slots so reading back via SYSINFO() does not accurately return the value you set.

**BLECONNECTCONFIG (configID,configValue)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td><strong>byVal configID</strong> AS INTEGER. The following are the values to update:</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>For all other configID values, the function returns an error.</td>
</tr>
<tr>
<td></td>
<td><strong>byVal configValue</strong> AS INTEGER. This contains the new value to set in the parameters indentified by configID.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```plaintext
//Example :: BleConnectConfig.sb
DIM rc, startTick

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

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

**Expected Output:**

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

Setting new parameters...
```
--- New Parameters:
Conn Scan Interval: 60ms
Conn Scan Window: 12ms
Conn slave latency: 3
ML Conn Interval Periodicity: 30ms

BLECONNECTCONFIG is an extension function.

BleDisconnect

FUNCTION

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

When the disconnection is complete, a EVBLEMSG message with msgId = 1 and context containing the handle is thrown to the smartBASIC runtime engine.

BLEDISCONNECT (nConnHandle)

<table>
<thead>
<tr>
<th>Arguments:</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>nConnHandle</td>
<td>INTEGER, ( n ) result code. The most typical value is 0x0000, indicating a successful operation</td>
</tr>
</tbody>
</table>

Interactive Command: No

//Example :: BleDisconnect.sb
DIM addr$: addr$=""
DIM rc

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

ONEVENT EVBLEMSG CALL HndlrBleMsg

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

Expected Output:

Adverts Started

New Connection 35800
BLEDISCONNECT is an extension function.

**BleSetCurConnParms**

**FUNCTION**

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

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

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

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
</table>

**Arguments:**

| nConnHandle | byVal nConnHandle AS INTEGER. Specifies the handle of the connection that must have the connection parameters changed. |
|-------------|-----------------------------------------------------------------------------------------------------------------
| nMinIntUs   | byVal nMinIntUs AS INTEGER. The minimum acceptable connection interval in microseconds. |
| nMaxIntUs   | byVal nMaxIntUs AS INTEGER. The maximum acceptable connection interval in microseconds. |
| nSuprToutUs | byVal nSuprToutUs AS INTEGER. The link supervision timeout for the connection in microseconds. It should be greater than the slave latency times that granted the connection interval. |
| nSlaveLatency | byVal nSlaveLatency AS INTEGER. The number of connection interval polls that the peripheral may ignore. This times the connection interval shall not be greater than the link supervision timeout. |

**Interactive Command:** No

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

For example, a keyboard could set the connection interval to 1000 msec and slave latency to 0. In this case, key presses are reported to the central device once per second, a poor user experience. Instead, the connection interval can be set to 50 msec, for example, and slave latency to 19. If there are no key presses, the power use is the same as before because ((19+1) * 50) equals 1000. When a key is pressed, the peripheral knows that the central device will poll within 50 msec, so it can send that keypress with a latency of 50 msec. A connection interval of 50 and slave latency of 19 means the slave is allowed to NOT acknowledge a poll for up to 19 poll messages from the central device.

```//Example :: BleSetCurConnParms.sb DIM rc DIM addr$ : addr$=""```
FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER

DIM intrvl,sprvTo,sLat

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

ONEVENT EVBLEMSG CALL HandlerBleMsg

IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "\nAdverts Started\n"
    PRINT "\nMake a connection to the WB45"
ELSE
    PRINT "\nAdvertisement not successful"
ENDIF

WAITEVENT

Expected Output (Unsuccessful Negotiation):

Adverts Started
Make a connection to the WB45
--- New Connection : 1352
Conn Interval 7500
Conn Supervision Timeout 7000000
Conn Slave Latency 0
Request new parameters
??? Conn Parm Negotiation FAILED
--- Disconnected : 1352

Expected Output (Successful Negotiation):
Adverts Started

Make a connection to the WB45
--- New Connection : 134
  Conn Interval    30000
  Conn Supervision Timeout  720000
  Conn Slave Latency      0

Request new parameters

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

Note:  The first set of parameters differ depending on your central device.

BLESETCURCONNPARMS is an extension function.

BleGetCurConnParms

FUNCTION

This function gets the current connection parameters for the connection identified by the connection handle. Given there are 3 connection parameters, the function takes three variables by reference so that the function can return the values in those variables.

BLEGETCURCONNPARMS (nConnHandle, nIntervalUs, nSuprToutUs, nSlaveLatency)

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nConnHandle</td>
<td>ByVal nConnHandle AS INTEGER. Specifies the handle of the connection to read the connection parameters of</td>
</tr>
<tr>
<td>nIntervalUs</td>
<td>byRef nIntervalUs AS INTEGER. The current connection interval in microseconds</td>
</tr>
<tr>
<td>nSuprToutUs</td>
<td>byRef nSuprToutUs AS INTEGER. The current link supervision timeout in microseconds for the connection.</td>
</tr>
<tr>
<td>nSlaveLatency</td>
<td>byRef nSlaveLatency AS INTEGER. The current number of connection interval polls that the peripheral may ignore. This value multiplied by the connection interval will not be greater than the link supervision timeout.</td>
</tr>
</tbody>
</table>

Note: See Note on Slave Latency.

Interactive Command
No

See previous example
BLEGETCURCONNPARMS is an extension function.

BleConnMngrUpdCfg

FUNCTION

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

BLECONNMNGRUPDCFG (nConnUpdateFirstDelay, nConnUpdateNextDelay, nConnUpdateMaxRetry)
Returns: INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.

Arguments:

- **nConnUpdateFirstDelay**: ByVal nConnUpdateFirstDelay AS INTEGER.
  In milliseconds 100 to 32000
- **nConnUpdateNextDelay**: ByVal nConnUpdateNextDelay AS INTEGER.
  In milliseconds 100 to 32000
- **nConnUpdateMaxRetry**: ByVal nConnUpdateMaxRetry AS INTEGER.
  In milliseconds 0 to 60000

Interactive Command: No

```vbs
Dim rc
#define CONN_UPD_FIRST_DELAY 500
#define CONN_UPD_NEXT_DELAY 800
#define CONN_UPD_MAX_RETRY 800
rc=bleConnMngrUpdCfg(CONN_UPD_FIRST_DELAY, CONN_UPD_NEXT_DELAY, CONN_UPD_MAX_RETRY)
If rc == 0 Then
    Print "\Connection manager successfully initialised"
Else
    Print "\Error: "; integer.h'rc
EndIf
```

Expected Output:

```
Connection manager successfully initialised
```

BLECONNMNGRPDCFG is an extension function.

**Security Manager Functions**

This section describes routines which manage all aspects of BLE security such as IO capabilities, Passkey exchange, OOB data, and bonding requirements.

**Events and Messages**

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

<table>
<thead>
<tr>
<th>MsgId</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Pairing in progress and display Passkey supplied in msgCtx.</td>
</tr>
<tr>
<td>10</td>
<td>A new bond has been successfully created</td>
</tr>
<tr>
<td>11</td>
<td>Pairing in progress and authentication key requested. Type of key is in msgCtx. msgCtx is 1 for passkey_type which is a number in the range 0 to 999999 and 2 for OOB key which is a 16 byte key.</td>
</tr>
<tr>
<td>23</td>
<td>OOB Data availability request, reply with BleSecMngrOobAvailable()</td>
</tr>
</tbody>
</table>

To submit a passkey, use the function **BLESECMNGRPASSKEY**.

**BleSecMngrJustWorksConf**
FUNCTION
This function is used to set the default action for when a pairing is in process and the I/O Capability is set to Just Works.

BLESECNMGJUSTWORKSCONF(nJustWorksConf)

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

Arguments:
byVal nJustWorksConf AS INTEGER.
If set to 0, pairing just works without confirmation. If set to 1, when pairing is in progress, you get an EVBLEMSG event with ID 11 and key type 0. In this case you accept or decline the pairing request with BleAcceptPairing().

Interactive Command
No

See example for BlePair().
BLESECNMGJUSTWORKSCONF is an extension function.

BleSecMngrOobPref

FUNCTION
This function is used to set a flag to indicate to the peer during a pairing that OOB pairing is preferred.

BLESECNMGROOBPREF(nOobPreferred)

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

Arguments:
byVal nJustWorksConf AS INTEGER.
If set to 0, we do not have OOB data available. If set to 1, OOB data is available. If set to 2, prompt me for OOB data availability.

Interactive Command
No

//Example :: BleSecMngrOobPref.sb

dim rc  
rc = BleSecMngrOobPref(1)  
IF (rc == 0) THEN  
   PRINT "OOB Pairing preference has been set."  
ENDIF

Expected Output:

OOB Pairing preference has been set.

BLESECNMGROOBPREF is an extension function.

BleSecMngrOobAvailable

FUNCTION
This function is used indicate that OOB data is available for the requested connection.
BLESECROOBAVAILABLE(connHandle, nOobAvail)

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connHandle</td>
<td>byVal connHandle AS INTEGER. The connection handle as received via the EVBLEMSG event with msgId set to 0.</td>
</tr>
<tr>
<td>nOobAvail</td>
<td>byVal nOobAvail AS INTEGER. If set to 0, we do not have OOB data available. If set to 1, OOB data is available.</td>
</tr>
</tbody>
</table>

Interactive Command: No

BLESECROOBAVAILABLE is an extension function.

BleAcceptParing

FUNCTION

This function is used to accept or decline a just works pairing request from the peer device at the other end of the connection with the specified handle. This function should, in most cases, be called in a EVBLEMSG handler when the nMsgID is 11 – Authentication Key Requested and the Key Type is 0.

Note: As part of the Bluetooth specification, a master may not use this function until the slave device has used it otherwise an invalid state error is returned.

BLEACCEPTPAIRING(nConnHandle, nAccept)

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nConnHandle</td>
<td>byVal nConnHandle AS INTEGER. The handle of the connection for which you are accepting or rejecting a pairing request.</td>
</tr>
<tr>
<td>nAccept</td>
<td>byVal nAccept AS INTEGER. Set to 0 to reject the pairing request, set to 1 to accept the pairing request.</td>
</tr>
</tbody>
</table>

Interactive Command: No

See example for BlePair().

BLEACCEPTPAIRING is an extension function.

BleSecMngrPasskey

FUNCTION

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

BLESECROOBPASSKEY(connHandle, nPassKey)

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connHandle</td>
<td>byVal connHandle AS INTEGER. The connection handle as received via the EVBLEMSG event with msgId set to 0.</td>
</tr>
<tr>
<td>nPassKey</td>
<td>byVal nPassKey AS INTEGER. The passkey to submit to the stack. Submit a value outside the range 0 to 999999 to reject the pairing.</td>
</tr>
</tbody>
</table>
Interactive Command | No
--- | ---

//Example :: BleSecMngrPasskey.sb

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

'// Called when data arrives through the UART - PIN
FUNCTION HandlerUartRxPIN()
    i = UartReadMatch(pin$,13)
    if i !=0 then
        pin$ = StrSplitLeft$(pin$,i-1)
        if strcmp(pin$,"quit")==0 || strcmp(pin$,"exit")==0 then
            rc=BleDisconnect(connHandle)
            exitfunc 0
        endif
        elseif BleSecMngrPassKey(connHandle,StrValDec(pin$))==0 then
            print "\nPasskey: ";pin$
            OnEvent EVUARTRX disable
        endif
        pin$=""
    endif
ENDFUNC 1

FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
SELECT nMsgId
    CASE 0
        connHandle = nCtx
        PRINT \n--- Ble Connection, ",nCtx
    CASE 1
        PRINT \n--- Disconnected ";nCtx;"
        EXITFUNC 0
    CASE 10
        PRINT \n--- New bond
    CASE 11
        PRINT \n+++ Auth Key Request, type=";nCtx
        PRINT \nEnter the pass key and Press Enter:\n" onevent evuartrx call HandlerUartRxPIN
    CASE 17
        print \nNew pairing/bond has replaced old key"
    CASE ELSE
ENDSELECT
ENDFUNC 1

ONEVENT EVBLEMSG CALL HandlerBleMsg

rc=BleSecMngrIoCap(2) //Set i/o capability - Keyboard Only (authenticated pairing)
IF BleAdvertStart(0,addr$,25,0,0)==0 THEN
    PRINT \nAdverts Started\n
    PRINT \nPair with the module"
ELSE
    PRINT \nAdvertisement not successful"
ENDIF
WAITEVENT
**Expected Output:**

```
Adverts Started

Pair with the module
--- Ble Connection, 2782
+++ Auth Key Request, type=1
Enter the pass key and Press Enter:
904096

Passkey: 904096
--- New bond
--- Disconnected 2782
```

BLESECMMNGRPASSKEY is an extension function.

**BleSecMngrOOBkey**

**FUNCTION**

This function submits an OOB (Out Of Band) key to the underlying stack during a pairing procedure when prompted by the EVBLEMSG with msgId set to 11 and the key type nCtx is 2, OOB. See Events & Messages.

**BLESECMMNGRPASSKEY(connHandle, nPassKey)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments:</strong></td>
<td></td>
</tr>
<tr>
<td>connHandle</td>
<td>byVal connHandle  AS INTEGER. This is the connection handle as received via the EVBLEMSG event with msgId set to 0.</td>
</tr>
<tr>
<td>oobKey$</td>
<td>byRef oobKey$  AS STRING. This is the OOB key to submit to the stack. Submit a 16 byte string, or a string of a different length to reject the request.</td>
</tr>
</tbody>
</table>

**Interactive Command**

```
DIM rc, connHandle
DIM addr$ : addr$=""
DIM oob$ : oob$ = "\11\22\33\44\55\66\77\88\99\00\aa\cc\bb\dd\ee\ff"
#define OOB_KEY     2

FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
SELECT nMsgId
CASE 0
    connHandle = nCtx
    PRINT "\nBle Connection ",nCtx
CASE 1
    PRINT "\nDisconnected ";nCtx;"\n"
    EXITFUNC 0
CASE 10
    PRINT "\n--- New bond"
CASE 11
    PRINT "\n+++ Auth Key Request, type=",nCtx
    if nCtx == OOB_KEY then
        rc=BleSecMngrOobKey(connHandle,oob$)
        print "\nOOB Key ";StrHexize$(oob$);" was used"
```

endif

CASE ELSE
   PRINT "\nUnknown Ble Msg"
ENDSELECT
ENDFUNC

ONEVENT EVBLEMSG CALL HandlerBleMsg

IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
   PRINT "\nAdverts Started\n"
   PRINT "\nMake a connection to the WB45"
ELSE
   PRINT "\nAdvertisement not successful"
ENDIF

WAITEVENT

Expected Output:

Adverts Started
Make a connection to the WB45
Ble Connection, 1655
+++ Auth Key Request, type=2
OOB Key 112234455667789911AAACBBDEEFF was used
--- New bond
Disconnected 1655

BLESECNMGRPASSKEY is an extension function.

**BleSecMngrKeySizes**

**FUNCTION**

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

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

**BLESECNMGRKEYSIZE** *(nMinKeysize, nMaxKeysize)*

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td>nMinKeysize</td>
<td>byVal nMinKeysize AS INTEGER. The minimum key size. The range of this value is from 7 to 16.</td>
</tr>
<tr>
<td>nMaxKeysize</td>
<td>byVal nMaxKeysize AS INTEGER. The maximum key size. The range of this value is from nMinKeysize to 16.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

//Example :: BleSecMngrKeySizes.sb
PRINT BleSecMngrKeySizes(8,15)
BLESECrmngrkeysizes is an extension function.

**BleSecMngrIoCap**

**FUNCTION**

This function sets the user I/O capability for subsequent pairings and is used to determine if the pairing is authenticated. This is related to Simple Secure Pairing as described in the following whitepapers:


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

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

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

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

**BLESECMNGROICAP** *(nIoCap)*

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td><strong>byVal nIoCap AS INTEGER.</strong> The user I/O capability for all subsequent pairings.</td>
</tr>
</tbody>
</table>

**Interactive Command**: No

//Example :: BleSecMngrIoCap.sb
PRINT BleSecMngrIoCap(1)

**Expected Output:**

0

See also examples for [BleSecMngrPasskey()](#) and [BlePair()](#).

BLESECrmngrbondreq is an extension function.

**BleSecMngrBondReq**

**FUNCTION**
This function is used to enable or disable bonding when pairing. If enabled, and if your application requires pairing, a peer device only needs to pair with this module once. If disabled, the device needs to pair every time it connects to the module.

**BLESECNMGRBONDREQ (nBondReq)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td>byVal nBondReq AS INTEGER.</td>
</tr>
<tr>
<td>nBondReq</td>
<td>0 Disable</td>
</tr>
<tr>
<td></td>
<td>1 Enable</td>
</tr>
</tbody>
</table>

Interactive Command: No

// Example :: BleSecMngrBondReq.sb
IF BleSecMngrBondReq(0) == 0 THEN
  PRINT "\nBonding disabled"
ENDIF

Expected Output:

```
Bonding disabled
```

BLESECNMGRBONDREQ is an extension function.

**BlePair**

**FUNCTION**

This routine is used to induce the module to pair with the peer and to specify whether to bond with the peer by storing pairing information in the bonding manager. This function is likely to be used if a write attempt to an attribute fails with a status code such as 0x105. See EvAttrWrite and EvAttrRead.

**BLEPAIR (hConn, nSave)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td>byRef hConn AS INTEGER. This is the connection handle provided in the EVBLEMSG(0) message which informs the stack that a connection had been established.</td>
</tr>
<tr>
<td>hConn</td>
<td></td>
</tr>
<tr>
<td>byVal nSave AS INTEGER This flag sets whether or not to bond.</td>
<td></td>
</tr>
<tr>
<td>nSave</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Interactive Command: No

dim rc, pr$, hC, hDesc
dim s$ : s$ = "\02\00"    //value to write to cccd to enable indications

// This example app was tested with a WB45 running the health thermometer sensor sample app which requires bonding.
// It connects, tries to read from the temperature characteristic and then initiates a bonding procedure when it fails.
```c
#define GATT_SERVER_ADDRESS "\01\F6\36\27\A6\0B\EA"
#define AUTHENTICATION_REQUIRED 0x0105
#define SERVICE_UUID 0x1809
#define CHAR_UUID 0x2a1c
#define DESC_UUID 0x2902

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

//'-- This handler is called when there is a significant BLE event
function HndlrBleMsg(byval nMsgId as integer, byval nCtx as integer) select nMsgId
    case 0
        hC = nCtx
        print "\nConnected, Finding Temp Measurement Char"
        rc=BleGattcFindDesc(nCtx, BleHandleUuid16(SERVICE_UUID), 0, BleHandleUuid16(CHAR_UUID), 0, BleHandleUuid16(DESC_UUID), 0)
        AssertRC(rc,35)
    case 1
        print "\n\n--- Disconnected"
    case 10
        print "\nNew bond created"
        print "\nAttempting to enable indications again"
        rc=BleGattcWrite(hC, hDesc, s$
        AssertRC(rc,58)
    case 11
        print "\nPair request: Accepting"
        rc=BleAcceptPairing(hC,1)
        AssertRC(rc,52)
        print "\nPairing in progress"
    case 17
        print "\nNew pairing/bond has replaced old key"
    case 18
        print "\nConnection now encrypted"
    case else
        endselect
endfunc

//'-- Called after BleGattcFindDesc returns success
//'-- Called after BleGattcFindDesc returns success
function HndlrFindDesc(hConn, hD) if hD==0 then
    print "\nCCCD not found"
    exitfunc 0
endif
```
hDesc = hD
    print "\nTemp Measurement Char CCCD Found. Attempting to enable indications"
    rc=BleGattcWrite(hConn, hDesc, s$)
    AssertRC(rc,58)
endfunc 1

'// Called after BleGattcRead returns success '////////////////////////////////////////////////////////////////////////////////
function HndlrAttrWriteExit(hConn, hAttr, nSts)
endfunc 0

'// Called after BleGattcRead returns success '////////////////////////////////////////////////////////////////////////////////
function HndlrAttrWrite(hConn, hAttr, nSts)
if nSts == 0 then
    print "\nIndications enabled"
    print "\nDisabling indications"
    s$ = "\00\00"
    rc=BleGattcWrite(hC, hDesc, s$)
onevent evattrwrite call HndlrAttrWriteExit
exitfunc 1

elseif nSts == AUTHENTICATION_REQUIRED then
    print "\n\nAuthentication required."
    //bond with the peer
    rc=BlePair(hConn, 1)
    AssertRC(rc,75)
    print " Bonding..."
endif
endfunc 1

//*****************************************************************************/
// Equivalent to main() in C
//*****************************************************************************/
rc=BleSecMngrIoCap(0) //set io capability to just works
rc=BleSecMngrJustWorksConf(1) //module will wait for confirmation (EVBLEMSG 11)
before just works pairing
rc=BleGattcOpen(0,0)
pr$ = GATT_SERVER_ADDRESS
rc=BleConnect(pr$, 10000, 25, 100, 3000000)
AssertRC(rc,91)

// Enable synchronous event handlers '////////////////////////////////////////////////////////////////////////////////
onevent evblemsg    call HndlrBleMsg
onevent evfinddesc call HndlrFindDesc
onevent evattrwrite call HndlrAttrWrite

waitevent

print "\nExiting..."
Connected, Finding Temp Measurement Char
Temp Measurement Char CCCD Found. Attempting to enable indications

Authentication required. Bonding...
Pair request: Accepting
Pairing in progress
Connection now encrypted
New bond created

Attempting to enable indications again
Indications enabled
Disabling indications
Exiting...

BLEPAIR is an extension function.

**BleEncryptConnection**

**FUNCTION**

This function is used to encrypt a BLE connection with a device that the module has previously bonded with (the device is present in the bonding manager).

**BLEENCRIPTCONNECTION(nConnHandle, nLtkMinSize, nMitmRequired)**

**Returns**

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

**Arguments:**

- **nConnHandle**
  - byVal nConnHandle AS INTEGER. The handle of the connection which is obtained from an EVBLEMSG message with ID 0 indicating that a connection had been established.

- **nLtkMinSize**
  - byVal nLtkMinSize AS INTEGER. The minimum long term key size which must be in the range 7-16.

- **nMitmRequired**
  - byVal nMitmRequired AS INTEGER. Set to 1 if MITM protection is required, 0 if not required.

**Interactive Command** No

```plaintext
dim rc, pr$, hC, hDesc
#define GATT_SERVER_ADDRESS "\01\F6\36\27\A6\0B\EA"

// This example app was tested with a WB45 running the health thermometer sensor sample app which the module had previously bonded with.

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

'//-----------------------------------------------
'// This handler is called when there is a significant BLE event
'//-----------------------------------------------
```

Embedded Wireless Solutions Support Center: 113
http://ews-support.lairdtech.com
www.lairdtech.com/wireless
**function** HndlBleMsg(byval nMsgId as integer, byval nCtx as integer)

    select nMsgId
    case 0
        hC = nCtx
        print "\nConnected"
        rc=BleEncryptConnection(hC, 16, 0)
        if rc==0 then
            print "\nEncrypting connection"
        else
            AssertRC(rc,28)
        endif
    case 1
        print "\n\n--- Disconnected"
        exitfunc 0
    case 10
        print "\nNew bond created"
    case 11
        print "\nPair request: Accepting"
        rc=BleAcceptPairing(hC,1)
        AssertRC(rc,52)
        print "\nPairing in progress"
    case 17
        print "\nNew pairing/bond has replaced old key"
    case 18
        print "\nConnection now encrypted"
        rc=BleDisconnect(hC)
    case else
    endselect
endfunc 1

rc=BleSecMngrIoCap(0) //set io capability to just works
rc=BleSecMngrJustWorksConf(0) //module will not wait for confirmation (EVBLEMSG 11)
before just works pairing

pr$ = GATT_SERVER_ADDRESS
rc=BleConnect(pr$, 10000, 25, 100, 30000000)
AssertRC(rc,91)

onevent evblemsg  call HndlBleMsg

waitevent

print "\nExiting..."

**Expected Output:**

Connected
Encrypting connection
Connection now encrypted
--- Disconnected
Exiting...

**BLEENCRIPTCONNECTION** is an extension function.

**GATT Server Functions**

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

A GATT table is a collection of adopted or custom services which, in turn, are a collection of adopted or custom characteristics. By definition, an adopted service cannot contain custom characteristics but the reverse is possible where a custom service can include both adopted and custom characteristics.

Descriptions of services and characteristics are available in the Bluetooth Specification v4.0 or newer. Because these descriptions are concise and difficult to understand, the following section attempts to familiarise you with these concepts using the smartBASIC programming environment perspective.

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

You can consider a service as a carrier bag to hold a group of related characteristics together where the printing on the carrier bag is a UUID. From a smartBASIC developer’s perspective, a set of characteristics is what you need to manage and the concept of service is only required at GATT table creation time.

A GATT table can have many services, each containing one or more characteristics. The difference between services and characteristics is expedited using an identification number called a UUID (Universally Unique Identifier) which is a 128-bit (16-byte) number. Adopted services or characteristics have a 16-bit (2-byte) shorthand identifier (which is an offset plus a base 128-bit UUID defined and reserved by the Bluetooth SIG); custom service or characteristics have the full 128-bit UUID. The logic behind this is that a 16-bit UUID implies that a specification has been published by the Bluetooth SIG whereas using a 128-bit UUID does NOT require any central authority to maintain a register of those UUIDs or specifications describing them.

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

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

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

How unique is a GUID?

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

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

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

---

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

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

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

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

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

A service is an abstraction of some collectivised functionality which, if broken down further, would cease to provide the intended behaviour. Two examples in the BLE domain that have been adopted by the Bluetooth SIG are Blood Pressure Service and Heart Rate Service. Each have sub-components that map to characteristics.

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

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

These services are also assigned 16-bit UUIDs (value 0x18xx) and are referenced in some of the smartBASIC API functions described in this section.

A list of all adopted characteristics is found at: http://developer.bluetooth.org/GATT/characteristics/Pages/CharacteristicsHome.aspx. You should note that these descriptors are also assigned 16-bit UUIDs (value 0x2Axx) and are referenced in some of the API functions described in this section. Custom characteristics have 128-bit (16-byte) UUIDs and API functions are provided to handle those.

Note: If you intend to create a custom service or characteristic and adopt the recommendation of a single 16-byte base UUID so that the service can be identified using a 2-byte UUID, then allocate a 16-bit value which is not going to coincide with any adopted values to minimise confusion. Selecting a similar value is possible and legal given that the base UUID is different.
The remainder of this introduction focuses on the specifics of how to create and manage a GATT table from a perspective of the smartBASIC API functions in the module.

Recall that a service was described as a carrier bag that groups related characteristics together and a characteristic is a data container (pot). Therefore, a remote GATT client looking at the server which is presented in your GATT table, sees multiple carrier bags each containing one or more pots of data.

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

From the smartBASIC application developer’s logical perspective, a GATT table looks nothing like the table that is presented in most BLE literature. Instead, the GATT table is simply a collection of char_handles that reference the characteristics (data containers) which have been registered with the underlying GATT table in the BLE stack.

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

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

With this simple model in mind, an overview of how the smartBASIC functions are used to register services and characteristics is illustrated in the flowchart on the right and sample code follows on the next page.
//Example :: ServicesAndCharacteristics.sb

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

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

DIM hChar11 // handles for characteristic 1 of Service 1
DIM hChar21 // handles for characteristic 2 of Service 1
DIM hChar12 // handles for characteristic 1 of Service 2
DIM hUuidS1 // handles for uuid of Service 1
DIM hUuidS2 // handles for uuid of Service 2
DIM hUuidC11 // handles for uuid of characteristic 1 in Service 1
DIM hUuidC12 // handles for uuid of characteristic 2 in Service 1
DIM hUuidC21 // handles for uuid of characteristic 1 in Service 2

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

//---Register Characteristic 1 in Service 1
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,10,0,rc)
mdCccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
mdScccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
chProp = BLE_CHAR_PROPERTIES_READ + BLE_CHAR_PROPERTIES_WRITE
hUuidC11 = BleHandleUuid16(0x2A37)
rc = BleCharNew(chProp, hUuidC11, mdAttr, mdCccd, mdScccd)
attr$ = "00"
rc = BleCharCommit(hSvc, attr$, hChar21)
rc = BleServiceCommit(hSvc)

//---Register Characteristic 2 in Service 1
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,10,0,rc)
mdCccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
mdScccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
chProp = BLE_CHAR_PROPERTIES_INDICATE
hUuidC12 = BleHandleUuid16(0x2A39)
rc = BleCharNew(chProp, hUuidC12, mdAttr, mdCccd, mdScccd)
attr$ = "00\00"
rc = BleCharCommit(hSvc, attr$, hChar11)
rc = BleServiceCommit(hSvc)

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

//---Register Characteristic 1 in Service 2
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_NONE,BLE_ATTR_ACCESS_NONE,10,0,rc)
mdCccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
mdScccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
chProp = BLE_CHAR_PROPERTIES_NOTIFY
hUuidC12 = BleHandleUuid16(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,2,0,rc)
rc = BleCharNew(chProp, hUuidC21, mdAttr, mdCccd, mdScccd)
attr$ = "00\00"
rc = BleCharCommit(hSvc, attr$, hChar12)
rc = BleServiceCommit(hSvc)
WB45 smartBASIC Extensions
User Guide

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

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

//------------------------------------------------------------------------------
// To deal with writes from a GATT client into characteristic 1 of Service 1
// which has the handle hChar11
//------------------------------------------------------------------------------
FUNCTION HandlerCharVal(Byval hChar As Integer) As Integer
DIM attr$
IF hChar == hChar11 THEN
   rc = BleCharValueRead(hChar11, attr$)
   PRINT "Svc1/Char1 has been written with = "; attr$
ENDIF
ENDFUNC 1

//enable characteristic value write handler
OnEvent EVCHARVAL call HandlerCharVal
WAITEVENT

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

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

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

//------------------------------------------------------------------------------
// indicate a value for characteristic 2 in service 1
//------------------------------------------------------------------------------
FUNCTION HandlerCharHvc(Byval hChar As Integer) As Integer
IF hChar == hChar12 THEN
   PRINT "Svc1/Char2 indicate has been confirmed"
ENDIF
ENDFUNC 1

//enable characteristic value indication confirm handler
OnEvent EVCHARHVC CALL HandlerCharHvc
attr$="somevalue"
rc = BleCharValueIndicate(hChar12, attr$)

The rest of this section details all the smartBASIC functions that help create that framework.
Events and Messages

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

BleGapSvcInit

FUNCTION

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

The GAP service contains five characteristics as listed at the following site:

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

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation</th>
</tr>
</thead>
</table>

Arguments:

<table>
<thead>
<tr>
<th>deviceName</th>
<th>byRef deviceName AS STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The name of the device (such as Laird_Thermometer) to store in the Device Name characteristic of the GAP service.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> When an advert report is created using BLEADVRPINIT(), this field is read from the service and an attempt is made to append it in the Device Name AD. If the name is too long, that function fails to initialise the advert report and a default name is transmitted. We recommend that the device name submitted in this call be as short as possible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nameWritable</th>
<th>byVal nameWritable AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If non-zero, the peer device is allowed to write the device name. Some profiles allow this to be made optional.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nAppearance</th>
<th>byVal nAppearance AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Field lists the external appearance of the device and updates the Appearance characteristic of the GAP service. Possible values: org.bluetooth.characteristic.gap.appearance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nMinConnInterval</th>
<th>byVal nMinConnInterval AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The preferred minimum connection interval, updates the ‘Peripheral Preferred Connection Parameters’ characteristic of the GAP service. Range is between 7500 and 4000000 microseconds (rounded to the nearest 1250 microseconds). This must be smaller than nMaxConnInterval.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nMaxConnInterval</th>
<th>byVal nMaxConnInterval AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The preferred maximum connection interval, updates the ‘Peripheral Preferred Connection Parameters’ characteristic of the GAP service. Range is between 7500 and 4000000 microseconds (rounded to the nearest 1250 microseconds). This must be larger than nMinConnInterval.</td>
</tr>
</tbody>
</table>
**nSupervisionTimeout**

byVal nSupervisionTimeout AS INTEGER

The preferred link supervision timeout and updates the ‘Peripheral Preferred Connectivity Parameters’ characteristic of the GAP service. Range is between 100000 to 32000000 microseconds (rounded to the nearest 10000 microseconds).

**nSlaveLatency**

byVal nSlaveLatency AS INTEGER

The preferred slave latency is the number of communication intervals that a slave may ignore without losing the connection and updates the ‘Peripheral Preferred Connectivity Parameters’ characteristic of the GAP service. This value must be smaller than (nSupervisionTimeout/ nMaxConnInterval) -1. i.e. nSlaveLatency < (nSupervisionTimeout / nMaxConnInterval) -1

---

**Interactive Command**

No

---

```plaintext
//Example :: BleGapSvcInit.sb

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

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

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

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

```

Expected Output:

Success

BLEGAPSCVINIT is an extension function.

**BleGetDeviceName$**

**FUNCTION**

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

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

**BLEGEDTDEVICENAME$ ()**

**Returns**

STRING, the current device name in the local GATT table. It is the same as that supplied in BleGapSvcInit() if the ‘nameWritable’ parameter was 0, otherwise it can be different. EVBLEMSG event is thrown with ‘msgid’ == 21 when the GATT client writes a new value.
**BleGetDeviceName$** is an extension function.

**BleSvcRegDevInfo**

**FUNCTION**

This function is used to register the Device Information service with the GATT server. The Device Information service contains nine characteristics as listed at the following website:


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

**BLESVCREGDEVINFO**

<table>
<thead>
<tr>
<th>Arguments:</th>
</tr>
</thead>
</table>
| **manfName$** | byVal manfName$ AS STRING  
The device manufacturer. Can be set empty to omit submission. |
| **modelNum$** | byVal modelNum$ AS STRING  
The device model number. Can be set empty to omit submission. |
| **serialNum$** | byVal serialNum$ AS STRING  
The device serial number. Can be set empty to omit submission. |
| **hwRev$** | byVal hwRev$ AS STRING  
The device hardware revision string. Can be set empty to omit submission. |
| **swRev$** | byVal swRev$ AS STRING  
The device software revision string. Can be set empty to omit submission. |

**Returns**: INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
### ByVal sysId$ AS STRING

The device system ID as defined in the specifications. Can be set empty to omit submission. Otherwise it shall be a string exactly eight octets long, where:

- **Byte 0..4 := Manufacturer Identifier**
- **Byte 5..7 := Organisationally Unique Identifier**

If the string is one character long and contains @, the system ID is created from the Bluetooth address if (and only if) an IEEE public address is set. If the address is the random static variety, this characteristic is omitted.

### ByVal regDataList$ AS STRING

The device’s regulatory certification data list as defined in the specification. It can be set as an empty string to omit submission.

### ByVal pnpId$ AS STRING

The device’s plug and play ID as defined in the specification. Can be set empty to omit submission. Otherwise, it shall be exactly 7 octets long, where:

- **Byte 0 := Vendor Id Source**
- **Byte 1,2 := Vendor Id (Byte 1 is LSB)**
- **Byte 3,4 := Product Id (Byte 3 is LSB)**
- **Byte 5,6 := Product Version (Byte 5 is LSB)**

---

```vbnet
//Example :: BleSvcRegDevInfo.sb

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

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

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

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

Success

BLESVCREGDEVINFO is an extension function.

**BleHandleUuid16**

**FUNCTION**

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

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

The returned handle is treated by the developer as an opaque entity and no further logic is based on the bit content, apart from all zeros which represent an invalid UUID handle.

**BLEHANDLEUUID16 (nUid16)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a nonzero handle shorthand for the UUID. Zero is an invalid UUID handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td><strong>nUid16</strong></td>
<td>ByVal nUid16 AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>nUid16 is first bitwise ANDed with 0xFFFF and the result is treated as an offset into the Bluetooth SIG 128 bit base UUID</td>
</tr>
</tbody>
</table>

**Interactive Command**

No

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

Expected Output:

Handle for HRS Uuid is FE01180D (-33482739)

BLEHANDLEUUID16 is an extension function.

**BleHandleUuid128**

**FUNCTION**

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

The base UUID is created by taking the 16-byte input string and setting bytes 12 and 13 to zero after extracting those bytes and storing them in the handle object. The handle also contains an index into an array of these 16-byte base UUIDs which are managed opaquely in the underlying stack.
The returned handle shall be treated by the developer as an opaque entity and no further logic shall be based on the bit content. However, note that a string of zeroes represents an invalid UUID handle.

**Note:** Ensure that you use a 16-byte UUID that has been generated using a random number generator with sufficient entropy to minimise duplication and that the first byte of the array is the most significant byte of the UUID.

### BLEHANDLEUUID128 (stUuid$)

**Returns:** INTEGER, A handle representing the shorthand UUID.

If zero, which is an invalid UUID handle, there is either no spare RAM memory to save the 16-byte base or more than 253 custom base UUIDs have been registered.

**Arguments:**

<table>
<thead>
<tr>
<th>byRef stUuid$ AS STRING</th>
</tr>
</thead>
</table>

Any 16-byte string that was generated using a UUID generation utility that has enough entropy to ensure that it is random. The first byte of the string is the MSB of the UUID (big endian format).

**Interactive Command:** No

//Example :: BleHandleUuid128.sb
DIM uuid$, hUuidCustom

//create a custom uuid for my ble widget
uuid$ = "ced9d91366924a1287d56f2764762b2a"

uuid$ = StrDehexize$(uuid$)

hUuidCustom = BLEHANDLEUUID128 (uuid$)

IF hUuidCustom == 0 THEN
    PRINT "Failed to create a handle"
ELSE
    PRINT "Handle for custom Uuid is "; integer$(hUuidCustom); "("; hUuidCustom; ")"
ENDIF

// hUuidCustom now references an object which points to
// a base uuid = ced9d91366924a1287d56f2764762b2a (note 0's in byte position 2/3)
// and an offset = 0xd913

**Expected Output:**

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

BLEHANDLEUUID128 is an extension function.

### BleHandleUuidSibling

**FUNCTION**

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

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

**BLEHANDLEUUIDSIBLING (nUuidHandle, nUuid16)**

**Returns:** INTEGER, a handle representing the shorthand UUID and can be zero which is an invalid UUID handle, if nUuidHandle is an invalid handle in the first place.

**Arguments:**

<table>
<thead>
<tr>
<th>nUuidHandle, nUuid16</th>
</tr>
</thead>
</table>

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

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

**nUuidHandle**
- **byVal nUuidHandle AS INTEGER**
  - A handle that was previously created using either `BleHandleUuid16()` or `BleHandleUuid128()`.

**nUuid16**
- **byVal nUuid16 AS INTEGER**
  - A UUID value in the range 0 to 65535 which is treated as an offset into the 128-bit base UUID referenced by `nUuidHandle`.

### Interactive Command

```vbnet
//Example :: BleHandleUuidSibling.ab
DIM uuid$ ,hUuid1, hUuid2   //hUuid2 will have the same base uuid as hUuid1

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

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

**Expected Output:**

```
Handle for custom Uuid is FC03D913 (-66856685)
Handle for custom sibling Uuid is FC031234 (-66907596)
BLEHANDLEUUIDSIBLING is an extension function
```

### BleServiceNew

**FUNCTION**

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

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

**Note:** When a GATT client queries a GATT server for services over a BLE connection, it only receives a list of PRIMARY services. SECONDARY services are a mechanism for multiple PRIMARY services to reference...
single instances of shared characteristics that are collected in a SECONDARY service. This referencing is expedited within the definition of a service using the concept of INCLUDED SERVICE which is an attribute that is grouped with the PRIMARY service definition. An Included Service is expedited using the function BleSvcAddInclueSvc() which is described immediately after this function.

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

BLESERVICENEW (nSvcType, nUuidHandle, hService)

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

<table>
<thead>
<tr>
<th>nSvcType</th>
<th>byVal nSvcType AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is zero for a SECONDARY service and 1 for a PRIMARY service. All other values are reserved for future use and result in this function failing with an appropriate result code.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nUuidHandle</th>
<th>byVal nUuidHandle AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is a handle to a 16-bit or 128-bit UUID that identifies the type of service function provided by all the characteristics collected under it. It has been pre-created using one of the three functions: BleHandleUuid16(), BleHandleUuid128(), or BleHandleUuidSibling()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hService</th>
<th>byRef hService AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If the service attribute is created in the GATT table, then this contains a composite handle which references the actual attribute handle. This is then subsequently used when adding characteristics to the GATT table. If the function fails to install the service attribute for any reason, this variable will contain 0 and the returned result code will be non-zero.</td>
</tr>
</tbody>
</table>

Interactive Command | No |

```c
//Example :: BleServiceNew.sb

#define BLE_SERVICE_SECONDARY 0
#define BLE_SERVICE_PRIMARY 1

//Create a Health Thermometer PRIMARY service attribute which has a uuid of 0x1809

DIM hHtsSvc //composite handle for hts primary service
DIM hUuidHT : hUuidHT = BleHandleUuid16(0x1809) //HT Svc UUID Handle

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

//Create a Battery PRIMARY service attribute which has a uuid of 0x180F

DIM hBatSvc //composite handle for battery primary service
DIM hUuidBatt : hUuidBatt = BleHandleUuid16(0x180F) //Batt Svc UUID Handle

IF BleServiceNew(BLE_SERVICE_PRIMARY,hUuidBatt,hBatSvc)==0 THEN
    PRINT "Battery Service attribute written to GATT table"
ELSE
    PRINT "Service Commit Failed"
ENDIF
```
BLESERVICENEW is an extension function.

**BleServiceCommit**

This function in the WB45 is used to commit a defined service using BleServiceNew() to the GATT table and should be called after the last characteristic/description has been created/commited for that service.

**BLESERVICECOMMIT (hService)**

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

Arguments:

- **hService** byVal hService AS INTEGER
  This handle is returned from BleServiceNew().

Interactive Command: No

See example for **BleCharCommit()**.

BleServiceCommit is an extension function.

**BleSvcAddIncludeSvc**

**FUNCTION**

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

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

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

This mechanism is provided to allow for a single set of characteristics to be shared by multiple primary services. This is most relevant if a characteristic is defined so that it can have only one instance in a GATT table but needs to be offered in multiple PRIMARY services. A typical implementation, where a characteristic is part of many PRIMARY services, installs that characteristic in a SECONDARY service (see **BleSvcCommit()**) and then uses the function defined in this section to add it to all the PRIMARY services that want to have that characteristic as part of their group.
It is possible to include a service which is also a PRIMARY or SECONDARY service, which in turn can include further PRIMARY or SECONDARY services. The only restriction to nested includes is that there cannot be recursion.

**Note:** If a service has INCLUDED services, then they are installed in the GATT table immediately after a service is created using `BleSvcCommit()` and before `BleCharCommit()`. The BT 4.0 specification mandates that any ‘included service’ attribute be present before any characteristic attributes within a particular service group declaration.

### `BleSvcAddIncludeSvc (hService)`

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

**Arguments:**

<table>
<thead>
<tr>
<th>hService</th>
<th>byVal hService  AS INTEGER</th>
</tr>
</thead>
</table>
| This argument contains a handle that was previously created using the function `BleSvcCommit()`.

**Interactive Command**  
No

```c
//Example :: BleSvcAddIncludeSvc.sb
#define BLE_SERVICE_SECONDARY                           0
#define BLE_SERVICE_PRIMARY                             1

//----------------------------------------------------------------------------
//Create a Battery SECONDARY service attribute which has a uuid of 0x180F
//----------------------------------------------------------------------------
dim hBatSvc  //composite handle for battery primary service
dim rc       //or we could have reused nHtsSvc
DIM charMet : charMet = BleAttrMetaData(1,1,10,1,metaSuccess)
DIM s$ : s$ = "Hello"  //initial value of char in Battery Service
DIM hBatChar
rc = BleServiceNew(BLE_SERVICE_SECONDARY, BleHandleUuid16(0x180F), hBatSvc)
rc = BleCharNew(3,BleHandleUuid16(0x2A1C),charMet,0,0)
rc = BleCharCommit(hBatSvc, s$,hBatChar)
rc = BleServiceCommit(hBatSvc)

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

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

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

`BleSvcAddIncludeSvc` is an extension function.

### `BleAttrMetaData`  

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

This function allows the creation of a 32-bit integer (an opaque object) which defines those properties and is then submitted along with other information to add the attribute to the GATT table.

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

However, when adding characteristics, which consists of a minimum of two attributes, one similar in function as the aforementioned service attribute and the other the actual data container, then properties for the value attribute must be specified. Here, properties refers to properties for the attribute, not properties for the characteristic container as a whole.

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

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

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

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

Consider a metadata object as a note to define how an attribute behaves; the GATT table manager needs this before it is added. Some attributes have those ‘notes’ specified by the BT specification; if this is the case, none need to be provided to the GATT table manager.

This function helps write that metadata.

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

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a 32-bit opaque data object to be used in subsequent calls when adding Characteristics to a GATT table.</th>
</tr>
</thead>
</table>

**Arguments:**
**byVal nReadRights AS INTEGER**
This specifies the read rights and shall have one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No access</td>
</tr>
<tr>
<td>1</td>
<td>Open</td>
</tr>
<tr>
<td>2</td>
<td>Encrypted with No Man-In-The-Middle (MITM) protection</td>
</tr>
<tr>
<td>3</td>
<td>Encrypted with Man-In-The-Middle (MITM) protection</td>
</tr>
<tr>
<td>4</td>
<td>Signed with No Man-In-The-Middle (MITM) protection (not available)</td>
</tr>
<tr>
<td>5</td>
<td>Signed with Man-In-The-Middle (MITM) protection (not available)</td>
</tr>
</tbody>
</table>

**Note:** In early releases of the firmware, 4 and 5 are not available.

**byVal nWriteRights AS INTEGER**
This specifies the write rights and shall have one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No access</td>
</tr>
<tr>
<td>1</td>
<td>Open</td>
</tr>
<tr>
<td>2</td>
<td>Encrypted with No Man-In-The-Middle (MITM) protection</td>
</tr>
<tr>
<td>3</td>
<td>Encrypted with Man-In-The-Middle (MITM) protection</td>
</tr>
<tr>
<td>4</td>
<td>Signed with No Man-In-The-Middle (MITM) protection (not available)</td>
</tr>
<tr>
<td>5</td>
<td>Signed with Man-In-The-Middle (MITM) protection (not available)</td>
</tr>
</tbody>
</table>

**Note:** In early releases of the firmware, 4 and 5 are not available.

**byVal nMaxDataLen AS INTEGER**
This specifies the maximum data length of the VALUE attribute. Range is from 1 to 512 bytes according to the BT specification; the stack implemented in the module may limit it for early versions. At the time of writing the limit is **20 bytes**.

**byVal fIsVariableLen AS INTEGER**
Set this to non-zero only if you want the attribute to automatically shorten its length according to the number of bytes written by the client.

For example, if the initial length is 2 and the client writes only 1 byte, then if this is 0, only the first byte gets updated and the rest remain unchanged. If this parameter is set to 1, then when a single byte is written the attribute shortens its length to accommodate. If the client tries to write more bytes than the initial maximum length, then the client receives an error response.

**byRef resCode AS INTEGER**
This variable is updated with a result code which is 0 if a metadata object was successfully returned by this call. Any other value implies a metadata object did not get created.

---

**Interactive Command**

```plaintext
//Example :: BleAttrMetadata.sb

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

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

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

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

Expected Output:
Success

BLEATTRMETADATA is an extension function.

BleCharNew

FUNCTION

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

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

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

Returns

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

Arguments:

<table>
<thead>
<tr>
<th>nCharProps</th>
<th>byVal nCharProps AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This variable contains a bit mask to specify the following high level properties for the characteristic that is added to the GATT table:</td>
</tr>
<tr>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>0</td>
<td>Broadcast capable (SCCD descriptor must be present)</td>
</tr>
<tr>
<td>1</td>
<td>Can be read by the client</td>
</tr>
<tr>
<td>2</td>
<td>Can be written by the client without a response</td>
</tr>
<tr>
<td>3</td>
<td>Can be written</td>
</tr>
<tr>
<td>4</td>
<td>Can be notifiable (CCCD descriptor must be present)</td>
</tr>
<tr>
<td>5</td>
<td>Can be indicatable (CCCD descriptor must be present)</td>
</tr>
<tr>
<td>6</td>
<td>Can accept signed writes</td>
</tr>
<tr>
<td>7</td>
<td>Reliable writes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nUuidHandle</th>
<th>byVal nUuidHandle AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This specifies the UUID that is allocated to the characteristic, either 16 or 128 bits. This variable is a handle, pre-created using one of the following functions: BleHandleUuid16(), BleHandleUuid128(), BleHandleUuidSibling().</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mdVal</th>
<th>byVal mdVal AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is the mandatory metadata used to define the properties of the Value attribute that is created in the characteristic and is pre-created with help from function BleAttrMetadata().</td>
</tr>
</tbody>
</table>
This is an optional metadata that is used to define the properties of the CCCD descriptor attribute that is created in the characteristic and is pre-created using the help of the function BleAttrMetadata() or set to 0 if CCCD is not to be created. If nCharProps specifies that the characteristic is notifiable or indicatable and this value contains 0, this function aborts with an appropriate result code.

This is an optional metadata that is used to define the properties of the SCCD descriptor attribute that is created in the characteristic and is pre-created using the help of the function BleAttrMetadata() or set to 0 if SCCD is not to be created. If nCharProps specifies that the characteristic is broadcastable and this value contains 0, this function aborts with an appropriate result code.

---

// Example :: BleCharNew.sb

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

//Create a new char:
//--- Indicatable, not Broadcastable (so mdCccd is included, but not mdSccd)
//--- Can be read, not written (shown in mdVal as well)

IF BleCharNew(0x22,charUuid,mdVal,mdCccd,0)==0 THEN
  PRINT "\nNew Characteristic created"
ELSE
  PRINT "\nFailed"
ENDIF

Expected Output:

New Characteristic created

BLECHARNEW is an extension function.

**BleCharDescUserDesc**

**FUNCTION**

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

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

**BLECHARDESCUSERDESC(usertDesc$, mdUser)**

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

**Arguments:**
userDesc$ byRef userDesc$ AS STRING
The user description string with which to initialize the descriptor. If the length of the string exceeds the maximum length of an attribute then this function aborts with an error result code.

mdUser byVal mdUser AS INTEGER
This is a mandatory metadata that defines the properties of the User Description Descriptor attribute created in the characteristic and pre-created using the help of BleAttrMetadata(). If the write rights are set to 1 or greater, the attribute is marked as writable and the client is able to provide a user description that overwrites the one provided in this call.

Interactive Command
No

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

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

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

Expected Output:
Char created and User Description 'A description' added

BLECHARDESCUSERDESC is an extension function.

BleCharDescPrstnFrmt

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

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

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

“One of the goals for the Generic Attribute Profile was to enable generic clients. A generic client is defined as a device that can read the values of a characteristic and display them to the user without understanding what they mean. . . . The most important aspect that denotes if a characteristic can be used by a generic client is the Characteristic Presentation Format descriptor. If this exists, it’s possible for the generic client to display its value, and it is safe to read this value.”
BLECHARDESCPRSTNFRMT (nFormat,nExponent,nUnit,nNameSpace,nNSdesc)

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

Arguments:

- **nFormat**
  - ByVal **nFormat** AS INTEGER
  - Valid range 0 to 255.
  - The format specifies how the data in the Value attribute is structured. A list of valid values for this argument is found at [http://developer.bluetooth.org/GATT/Pages/FormatTypes.aspx](http://developer.bluetooth.org/GATT/Pages/FormatTypes.aspx) and the enumeration is described in the BT 4.0 spec, section 3.3.3.5.2.
  - The following is the enumeration list at the time of writing:
    - 0x00: RFU
    - 0x01: boolean
    - 0x02: 2bit
    - 0x03: nibble
    - 0x04: unit8
    - 0x05: uint12
    - 0x06: uint16
    - 0x07: uint24
    - 0x08: uint32
    - 0x09: uint48
    - 0x0A: uint64
    - 0x0B: uint128
    - 0x0C: sint8
    - 0x0D: sint12
    - 0x0E: sint16
    - 0x0F: sint24
    - 0x10: sint32
    - 0x11: sint48
    - 0x12: sint64
    - 0x13: sint128
    - 0x14: float32
    - 0x15: float64
    - 0x16: SFLOAT
    - 0x17: FLOAT
    - 0x18: uint16
    - 0x19: utf8s
    - 0x1A: utf16s
    - 0x1B: struct
    - 0x1C-0xFF: RFU

- **nExponent**
  - ByVal **nExponent** AS INTEGER
  - This value is used with integer data types given by the enumeration in nFormat to further qualify the value so that the actual value is: `actual value = Characteristic Value * 10 to the power of nExponent`.
  - Valid range -128 to 127.

- **nUnit**
  - ByVal **nUnit** AS INTEGER
  - This value is a 16-bit UUID used as an enumeration to specify the units which are listed in the Assigned Numbers document published by the Bluetooth SIG, found at: [http://developer.bluetooth.org/GATT/units/Pages/default.aspx](http://developer.bluetooth.org/GATT/units/Pages/default.aspx)
  - Valid range 0 to 65535.

- **nNameSpace**
  - ByVal **nNameSpace** AS INTEGER
  - The value identifies the organization, defined in the Assigned Numbers document published by the Bluetooth SIG, found at: [https://developer.bluetooth.org/GATT/Pages/GATTNamespaceDescriptors.aspx](https://developer.bluetooth.org/GATT/Pages/GATTNamespaceDescriptors.aspx)
  - Valid range 0 to 255.

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

Interactive Command: No

//Example :: BleCharDescPrstnFrmt.sb
## BLECHARDESCPRSTNFRMT

BLECHARDESCPRSTNFRMT is an extension function.

**BleCharDescAdd**

**FUNCTION**

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

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

**BLECHARDESCADD (nUuid16, attr$, mdDesc)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>

- **nUuid16** `byVal nUuid16 AS INTEGER`
  - This is a value in the range 0x2905 to 0x2999
  - **Note:** This is the actual UUID value, NOT the handle.
  - The highest value at the time of writing is 0x2908, defined for the Report Reference Descriptor.
attr$ byRef attr$ AS STRING
This is the data that is saved in the Descriptor’s attribute

mdDesc byVal n AS INTEGER
This is mandatory metadata that is used to define the properties of the Descriptor attribute
that is created in the Characteristic and was pre-created using the help of the function
BleAttrMetadata(). If the write rights are set to 1 or greater, then the attribute is marked as
writable and the client is able to modify the attribute value.

Interactive Command No

//Example :: BleCharDescAdd.sb

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

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

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

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

//++++
//Add the other Descriptor 0x29XX -- second one
//++++
attr$="some_value2"
rc=rc+BleCharDescAdd(0x2906,attr$,mdChrDsc)

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

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

Expected Output:
Other descriptors added successfully
BLECHARDESCADD is an extension function.

**BleCharCommit**

**FUNCTION**

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

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

The characteristic handle that is returned references an internal opaque object that is a linked list of all the attribute handles in the characteristic which by definition implies that there is a minimum of 1 (for the characteristic value attribute) and more as appropriate. For example, if the characteristic’s property specified is notifiable then a single CCCD attribute also exists.

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

**BLECHARCOMMIT (hService, attr$, charHandle)**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hService</strong></td>
<td>byVal hService AS INTEGER</td>
<td>This is the handle of the service to which the characteristic belongs, which in turn was created using the function BleSvcCommit().</td>
</tr>
<tr>
<td><strong>attr$</strong></td>
<td>byRef attr$ AS STRING</td>
<td>This string contains the initial value of the value attribute in the characteristic. The content of this string is copied into the GATT table and the variable can be reused after this function returns.</td>
</tr>
<tr>
<td><strong>charHandle</strong></td>
<td>byRef charHandle AS INTEGER</td>
<td>The composite handle for the newly created characteristic is returned in this argument. It is zero if the function fails with a non-zero result code. This handle is then used as an argument in subsequent function calls to perform read/write actions, so it is must be placed in a global smartBASIC variable. When a significant event occurs as a result of action by a remote client, an event message is sent to the application which can be serviced using a handler. That message contains a handle field corresponding to this composite characteristic handle. Standard procedure is to select on that value to determine for which characteristic the message is intended.</td>
</tr>
</tbody>
</table>

See event messages: EVCHARHVC, EVCHARVAL, EVCHARCCCD, EVCHARSCCD, EVCHARDESC.

**Interactive Command**

No

// Example :: BleCharCommit.sb
```
#DEFINE BLE_SERVICE_SECONDARY 0
#DEFINE BLE_SERVICE_PRIMARY 1

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

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

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

// Commit the characteristics with some initial data
attr$="hello\00world\64"
IF BleCharCommit(hHtsSvc,attr$,hHtsMeas)==0 THEN
  PRINT 
    char
    Characteristic Commited
ELSE
  PRINT 
    char
    Failed
ENDIF
rc=BleServiceCommit(hHtsSvc)

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

**Expected Output:**

```
Characteristic Commited
```

**BLECHARCOMMIT** is an extension function.

**BleCharValueRead**

**FUNCTION**

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

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

**BLECHARVALUEREAD (charHandle,attr$)**

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

**Arguments:**

- **charHandle** ByVal charHandle AS INTEGER
  This is the handle to the characteristic whose value must be read which was returned when
BleCharCommit() was called.

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

Interactive Command
No

//Example :: BleCharValueRead.sb
DIM hMyChar, rc, conHndl

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

FUNCTION HndlrChar(BYVAL chrHndl, BYVAL offset, BYVAL len)
    dim s$
    IF chrHndl == hMyChar THEN
        PRINT "\n\nNew Char Value: ";s$
    ENDIF
rc=BleCharValueRead(hMyChar,s$)
PRINT "\nNew Char Value: ";s$
ENDFUNC 0

FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nMsg1, BYVAL nCnt)
    conHndl=nCnt
ENDFUNC 1

IF OnStartup()=0 THEN
    DIM at$ : rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic value attribute: ";at$\nConnect to WB45 and send a new value\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

ONEVENT EVCHARVAL CALL HndlrChar
ONEVENT EVBLEMSG CALL HndlrBleMsg
WAITEVENT
PRINT "\nExiting..."

Expected Output:

Characteristic value attribute: Hi
Connect to WB45 and send a new value

New characteristic value: Laird
Exiting...

BLECHARVALUEREAD is an extension function.

**BleCharValueWrite**

**FUNCTION**

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

**BLECHARVALUEWRITE (charHandle,attr$)**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charHandle</td>
<td>byVal charHandle AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>This is the handle to the characteristic whose value must be updated which</td>
</tr>
<tr>
<td></td>
<td>was returned when BleCharCommit() was called.</td>
</tr>
<tr>
<td>attr$</td>
<td>byRef attr$ AS STRING</td>
</tr>
<tr>
<td></td>
<td>String variable, contains new value to write to the characteristic.</td>
</tr>
</tbody>
</table>

**Interactive Command**

No

//Example :: BleCharValueWrite.sb
DIM hMyChar,rc

// Initialise and instantiate service, characteristic,
// Initialise char, write/read enabled, accept signed writes
rc=BleCharNew(0x4A,BleHandleUuid16(1), BleAttrMetaData(1, 1, 20, 0), 0, 0)
rc=BleCharCommit(hSvc, attr$, hMyChar)
rc = BleServiceCommit(hSvc)
ENDFUNC
// Uart Rx handler - write input to characteristic
FUNCTION HndlrUartRx()
    TimerStart(0,10,0)
ENDFUNC 1

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

IF OnStartup()==0 THEN
    DIM at$ : rc = BleCharValueRead(hMyChar,at$)
    PRINT \nCharacteristic value attribute: ";at$"\n    ELSE
        PRINT \nFailure OnStartup
    ENDIF
ONEVENT EVUARTRX CALL HndlrUartRx
ONEVENT EVTMR0 CALL HndlrTmr0
WAITEVENT
PRINT \nExiting...

Expected Output:
Characteristic value attribute: Hi
Send a new value
Laird
New characteristic value: Laird
Exiting...

BLECHARVALUEWRITE is an extension function.

BleCharValueNotify

FUNCTION

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

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

BLECHARVALUENOTIFY (charHandle,attr$)

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

Arguments:
**charHandle**

*byVal charHandle AS INTEGER*

This is the handle to the characteristic whose value must be updated which is returned when BleCharCommit() is called.

**attr$**

*byRef attr$ AS STRING*

String variable containing new value to write to the characteristic and then send as a notification to the client. If there is no connection, this function fails with an appropriate result code.

<table>
<thead>
<tr>
<th>Interactive Command</th>
<th>No</th>
</tr>
</thead>
</table>

//Example :: BleCharValueNotify.sb
DIM hMyChar, rc, at$, conHndl

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

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

// Close connections so that we can run another app without problems
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
conHndl=nCtx
IF nMsgID=1 THEN
PRINT "\n\n--- Disconnected from client"
EXITFUNC 0
ELSEIF nMsgID=0 THEN
PRINT "\n--- Connected to client"
ENDIF
ENDFUNC 1
// CCCD descriptor written handler
//============================================================================
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
  DIM value$  
  IF charHandle==hMyChar THEN
    PRINT "\nCCCD Val: ";nVal
    IF nVal THEN
      PRINT " : Notifications have been enabled by client"
      value$="hello"
      IF BleCharValueNotify(hMyChar,value$)!=0 THEN
        PRINT "\nFailed to notify new value ";INTEGER.H'r
      ELSE
        PRINT "\nSuccessful notification of new value"
        EXITFUNC 0
      ENDIF
    ELSE
      PRINT " : Notifications have been disabled by client"
    ENDIF
    ELSE
      PRINT "\nThis is for some other characteristic"
  ENDIF
ENDFUNC

ONEVENT EVBLEMSG CALL HndlrBleMsg
ONEVENT EVCHARCCCD CALL HndlrCharCccd

IF OnStartup()==0 THEN
  rc = BleCharValueRead(hMyChar,at$)
  PRINT "\nCharacteristic Value: ";at$
  PRINT "\nYou can connect and write to the CCCD characteristic."
  PRINT "\nThe WB45 will then notify your device of a new characteristic value"
ELSE
  PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

CloseConnections()
PRINT "\nExiting..."

Expected Output:

Characteristic Value: Hi
You can connect and write to the CCCD characteristic.
The WB45 will then notify your device of a new characteristic value
--- Connected to client
CCCD Val: 0 : Notifications have been disabled by client
CCCD Val: 1 : Notifications have been enabled by client
Successful notification of new value
Exiting...

BLECHARVALUENOTIFY is an extension function.

BleCharValueIndicate
FUNCTION

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

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

**BLECHARVALUEINDICATE (charHandle,attr$)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| charHandle | byVal charHandle AS INTEGER  
This is the handle to the characteristic whose value must be updated which is returned when BleCharCommit() was called. |
| attr$ | byRef attr$ AS STRING  
String variable containing new value to write to the characteristic and then to send as a notification to the client. If there is no connection, this function fails with an appropriate result code. |

Interactive Command  
No

```
//Example :: BleCharValueIndicate.sb
DIM hMyChar,rc,at$,conHndl

//==============================================================================
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
FUNCTION OnStartup()
  DIM rc, hSvc, at$, attr$, adRpt$, addr$, scRpt$
  attr$="Hi"
  DIM mdCccd : mdCccd = BlesAttrMetadata(1,1,2,0,rc) //CCCD metadata for char
  rc=BleServiceNew(1,BleHandleUuid16(0x18EE),hSvc)
  //initialise char, write/read enabled, accept signed writes, notifiable
  rc=BleCharNew(0x22,BleHandleUuid16(1),BleAttrMetaData(1,0,20,0,rc),mdCccd,0)
  //commit char initialised above, with initial value "hi" to service 'hMyChar'
  rc=BleCharCommit(hSvc,attr$,hMyChar)
  //commit changes to service
  rc=BleServiceCommit(hSvc)
  rc=BleScanRptInit(scRpt$)
  //Add 1 service handle to scan report
  rc=BleAdvRptAddUuid16(scRpt$),(scRpt$),0x18EE,-1,-1,-1,-1)
  //commit reports to GATT table - adRpt$ is empty
  rc=BleAdvRptsCommit(adRpt$,scRpt$)
  rc=BleAdvRptStart(0,addr$,50,0,0)
ENDFUNC
rc

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

Embedded Wireless Solutions Support Center:  145  Americas: +1-800-492-2320
http://ews-support.lairdtech.com  Europe: +44-1628-858-940
www.lairdtech.com/wireless  Hong Kong: +852 2923 0610
PRINT "\n--- Connected to client"
ENDIF
ENDFUNC

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

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

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

IF OnStartup()==0 THEN
rc = BleCharValueRead(hMyChar,at$)
PRINT "\nCharacteristic Value: ";at$
PRINT "\nYou can connect and write to the CCCD characteristic."
PRINT "\nThe WB45 will then indicate a new characteristic value"
ELSE
PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
rc=BleDisconnect(conHndl)
rc=BleAdvertStop()
PRINT "\nExiting..."
Characteristic Value: Hi
You can connect and write to the CCCD characteristic.
The WB45 will then indicate a new characteristic value

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

Got confirmation of recent indication
Exiting...

BLECHARVALUEINDICATE is an extension function.

**BleCharDescRead**

**FUNCTION**

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

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

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

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| charHandle       | byVal charHandle AS INTEGER
| This is the handle to the characteristic whose descriptor must be read which is returned when BleCharCommit() is called and is been supplied in the EVCHARDESC event message. |
| nDescHandle      | byVal nDescHandle AS INTEGER
| This is an index into an opaque array of descriptor handles inside the charHandle and is supplied as the second parameter in the EVCHARDESC event message. |
| nOffset          | byVal nOffset AS INTEGER
| This is the offset into the descriptor attribute from which the data should be read and copied into attr$. |
| nLength          | byVal nLength AS INTEGER
| This is the number of bytes to read from the descriptor attribute from offset nOffset and copied into attr$. |
| nDescUuidHandle  | byRef nDescUuidHandle AS INTEGER
| On exit, this is updated with the uuid handle of the descriptor that got updated. |
| attr$            | byRef attr$ AS STRING
| On exit, this string variable contains the new value from the characteristic descriptor. |

Interactive Command

No

//Example :: BleCharDescRead.sb
DIM rc,conHndl,hMyChar

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

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

    // Commit characteristic
    attr$ = "00"  // no initial alert
    rc = BleCharCommit(hSvc, attr$, hMyChar)

    // Add 1 char handle to scan report
    rc = BleAdvRptAddUuid16(scRpt$, 0x2AFF, -1, -1, -1, -1, -1)
    // commit reports to GATT table - adRpt$ is empty
    rc = BleAdvRptsCommit(adRpt$, scRpt$)
    rc = BleAdvertStart(0, addr$, 200, 0, 0)
ENDSUB

SUB CloseConnections()
    rc = BleDisconnect(conHndl)
    rc = BleAdvertStop()
ENDSUB

FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl = nCtx
ENDFUNC

FUNCTION HandlerCharDesc(BYVAL hChar AS INTEGER, BYVAL hDesc AS INTEGER)
    DIM instnc, nUuid, a$, offset, duid
    IF hChar == hMyChar THEN
        rc = BleCharDescRead(hChar, hDesc, 0, 20, duid, a$)
        IF rc == 0 THEN
            PRINT "\nRead 20 bytes from index "; offset; " in new char value."
            PRINT "\n  :New Descriptor Data: "; StrHexize$(a$);
            PRINT "\n  :Length=":StrLen(a$)
            PRINT "\n  :Descriptor UUID "; integer.h' duid
        ELSE
            PRINT "\nCould not access the uuid"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC
//install a handler for writes to characteristic values
ONEVENT EVCHARDESC CALL HandlerCharDesc
ONEVENT EVBLEMSG CALL HndlrBleMsg

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

//wait for events and messages
WAITEVENT

CloseConnections()
PRINT "\nExiting..."

Expected Output:

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

BLECHARDESCREAD is an extension function.

GATT Client Functions

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

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

To interact with a remote GATT server, it is important to have a good understanding of how it is constructed. It is best to see it as a table consisting of many rows and three visible columns (handle, type, value) and at least one more invisible column whose content affects access to the data column.

<table>
<thead>
<tr>
<th>16 bit Handle</th>
<th>Type (16 or 128 bit)</th>
<th>Value (1 to 512 bytes)</th>
<th>Permissions</th>
</tr>
</thead>
</table>

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

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

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

When a row with Type = 0x2803 (a characteristic) is encountered, then the next row contains the value for that characteristic; afterwards, there may be zero or more descriptors.
This means each characteristic consists of at least two rows in the table; and if descriptors exist for that characteristic, then a single row per descriptor.

<table>
<thead>
<tr>
<th>Handle</th>
<th>Type</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>0x2800</td>
<td>UUID of the Service</td>
<td>Primary Service 1 Start</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x2803</td>
<td>Properties, Value Handle, Value UUID1</td>
<td>Characteristic 1 Start</td>
</tr>
<tr>
<td>0x0003</td>
<td></td>
<td>Value UUID1</td>
<td>Value : 1 to 512 bytes Actual data</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x2803</td>
<td>Properties, Value Handle, Value UUID2</td>
<td>Characteristic 2 Start</td>
</tr>
<tr>
<td>0x0005</td>
<td></td>
<td>Value UUID2</td>
<td>Value : 1 to 512 bytes Actual data</td>
</tr>
<tr>
<td>0x0006</td>
<td>0x2902</td>
<td>Value</td>
<td>Descriptor 1 (CCCD)</td>
</tr>
<tr>
<td>0x0007</td>
<td>0x2903</td>
<td>Value</td>
<td>Descriptor 2 (SCCD)</td>
</tr>
<tr>
<td>0x0008</td>
<td>0x2800</td>
<td>UUID of the Service</td>
<td>Primary Service 2 Start</td>
</tr>
<tr>
<td>0x0009</td>
<td>0x2803</td>
<td>Properties, Value Handle, Value UUID3</td>
<td>Characteristic 1 Start</td>
</tr>
<tr>
<td>0x000A</td>
<td>Value UUID3</td>
<td>Value : 1 to 512 bytes</td>
<td>Actual data</td>
</tr>
<tr>
<td>0x000B</td>
<td>0x2800</td>
<td>UUID of the Service</td>
<td>Primary Service 3 Start</td>
</tr>
<tr>
<td>0x000C</td>
<td>0x2803</td>
<td>Properties, Value Handle, Value UUID3</td>
<td>Characteristic 3 Start</td>
</tr>
<tr>
<td>0x000D</td>
<td>Value UUID3</td>
<td>Value : 1 to 512 bytes</td>
<td>Actual data</td>
</tr>
<tr>
<td>0x000E</td>
<td>0x2902</td>
<td>Value</td>
<td>Descriptor 1 (CCCD)</td>
</tr>
<tr>
<td>0x000F</td>
<td>0x2903</td>
<td>Value</td>
<td>Descriptor 2 (SCCD)</td>
</tr>
<tr>
<td>0x0010</td>
<td>0x2904</td>
<td>Value (presentation format data)</td>
<td>Descriptor 3</td>
</tr>
<tr>
<td>0x0011</td>
<td>0x2906</td>
<td>Value (valid range)</td>
<td>Descriptor 4 (Range)</td>
</tr>
</tbody>
</table>

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

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

Likewise, each characteristic starts with a row with Type=0x2803 and all rows following it (up to a row with type = 0x2800/2801/2803) are considered belonging to that characteristic. For example, the characteristic at row with handle = 0x0009 has the mandatory value row and then two descriptors.

The Bluetooth specification allows for multiple instances of the same service or characteristics or descriptors and they are differentiated by the unique handle. This ensures no ambiguity.

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

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

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

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

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

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

Events and Messages
The nature of GATT client operation consists of multiple queries and acting on the responses. Because the connection intervals are slower than the CPU speed, responses can arrive many 10s of milliseconds after the procedure is triggered; these are delivered to an application using an event or message. Since these event/messages are tightly coupled with the appropriate commands, all but one is described when the command that triggers them is described.

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

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

- Connection Handle

```vbnet
//Example :: EVGATTCTOUT.sb
//
DIM rc,conHndl
//==============================================================================
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGATTcOpen(0,0) : ENDIF
ENDFUNC rc
```
// Ble event handler

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

FUNCTION HandlerGATTcTout(cHndl) AS INTEGER
    PRINT "\nEVGATTCTOUT connHandle=",cHndl
ENDFUNC

// Main() equivalent

ONEVENT EVBLEMSG CALL HndlrBleMsg
OnEvent EVGATTCTOUT call HandlerGATTcTout

rc = OnStartup()
WAITEVENT

Expected Output:

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

BleGattcOpen

FUNCTION

This function is used to initialise the GATT client functionality for immediate use so that appropriate buffers for caching GATT responses are created in the heap memory. About 300 bytes of RAM is required by the GATT client manager; given that a majority of WB45 use cases do not use it, the sacrifice of 300 bytes is not worth the permanent allocation of memory.

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

Use SYSINFO(2019) to obtain the actual default size and SYSINFO(2020) to obtain the maximum allowed. The same information can be obtained in interactive mode using the commands AT I 2019 and 2020 respectively.
Note: When the ring buffer for the notifiable and indicatable characteristics is full, then any new messages are discarded and, depending on the flags parameter, the indicates are or are not confirmed.

This function is safe to call when the GATT client manager is already open. However, in that case, the parameters are ignored and existing values are retained. Existing GATTc client operations are not interrupted. It is recommended that this function NOT be called when in a connection.

BLEGATTCOPEN (nNotifyBufLen, nFlags)

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

Arguments:

*nNotifyBufLen* byVal nNotifyBufLen AS INTEGER
This is the size of the ring buffer used for incoming notifiable and indicatable characteristic data. Set to 0 to use the default size.

*nFlags* byVal nFlags AS INTEGER
Bit 0 – Set to 1 to disable automatic indication confirmations. If the buffer is full then the Handle Value Confirmation is only sent when BleGattcNotifyRead() is called to read the ring buffer.
Bit 1..31 – Reserved for future use and must be set to 0s.

Interactive Command
No

//Example :: BleGattcOpen.sb
DIM rc
//open the GATT client with default notify/indicate ring buffer size
rc = BleGATTCOpen(0, 0)
IF rc == 0 THEN
    PRINT "\nGATT Client is now open"
ENDIF

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

Expected Output:

GATT Client is now open
GATT Client is still open, because already open

BLEGATTCOPEN is an extension function.

BleGattcClose

SUBROUTINE
This function is used to close the GATT client manager and is safe to call if it is already closed. It is recommended that this function NOT be called when in a connection.

BLEGATTCLOSE ()
Command

//Example :: BleGattcClose.sb

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

Expected Output:

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

BLEGATTCCLOSE is an extension subroutine.

BleDiscServiceFirst / BleDiscServiceNext

FUNCTIONS

This pair of functions is used to scan the remote GATT server for all primary services with the help of the EVDISCPRIMSVC message event. When called, a handler for the event message must be registered as the discovered primary service information is passed back in that message.

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

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

EVDISCPRIMSVC event message

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

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

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

BLEDISCSERVICEFIRST (connHandle,startAttrHandle,uuidHandle)

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

Register a handler for the EVDISCPRIMSVC event message

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

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

**Returns**
INTEGER, a result code. The typical value is 0x0000, indicating a successful operation. This means an EVDISCPRIMSVC event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message is NOT thrown.

**Arguments:**

**connHandle**
*byVal nConnHandle AS INTEGER*
This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.

**startAttrHandle**
*byVal startAttrHandle AS INTEGER*
This is the attribute handle from where the scan for primary services will be started and you can typically set it to 0 to ensure that the entire remote GATT Server is scanned.

**uuidHandle**
*byVal uuidHandle AS INTEGER*
Set this to 0 if you want to scan for any service, otherwise this value will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

**Interactive Command** No

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

**Returns**
INTEGER, a result code.
The typical value is 0x0000, indicating a successful operation and it means an EVDISCPRIMSVC event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message is not thrown.

**Arguments:**

**connHandle**
*byVal nConnHandle AS INTEGER*
This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.

**Interactive Command** No

// Example :: BleDiscServiceFirst.Next.sb

//
Remote server has 5 primary services with 16 bit UUIDs and 3 with 128 bit UUIDs. 3 of the 16 bit UUIDs are the same value 0xDEAD and 2 of the 128 bit UUIDs are also the same 112233445566778899AABBCCDDEEFF.

Server created using BleGATTcTblDiscPrimSvc.sub invoked in _OpenMcp.scr using Nordic USB Dongle PC10000.

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

// Initialise and instantiate service, characteristic, start advert.

```vbnet
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc = BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc = BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc = BleAdvRptsCommit(adRpt$, scRpt$) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc
```

// Close connections so that we can run another app without problems.

```vbnet
SUB CloseConnections()
    rc = BleDisconnect(conHndl)
    rc = BleAdvertStop()
ENDSUB
```

// Ble event handler.

```vbnet
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl = nCtx
    IF nMsgId == 1 THEN
        PRINT "\n\nDisconnected"
        EXITFUNC 0
    ELSEIF nMsgId == 0 THEN
        PRINT "\n\nConnected, so scan remote GATT Table for ALL services"
        rc = BleDiscServiceFirst(conHndl, 0, 0)
        IF rc==0 THEN
            //HandlerPrimSvc() will exit with 0 when operation is complete
            WAITEVENT
            PRINT "\nScan for service with uuid = 0xDEAD"
            uHndl = BleHandleUuid16(0xDEAD)
            rc = BleDiscServiceFirst(conHndl, 0, uHndl)
            IF rc==0 THEN
                //HandlerPrimSvc() will exit with 0 when operation is complete
                WAITEVENT
                uu$ = "112233445566778899AABBCCDDEFF00"
                PRINT "\nScan for service with custom uuid "; uu$
                uu$ = StrDehexize$(uu$)
                uHndl = BleHandleUuid128(uu$)
                rc = BleDiscServiceFirst(conHndl, 0, uHndl)
                IF rc==0 THEN
                    //HandlerPrimSvc() will exit with 0 when operation is complete
                    WAITEVENT
                    PRINT "\nScan for service with custom uuid "; uu$
                    uu$ = StrDehexize$(uu$)
                    uHndl = BleHandleUuid128(uu$)
                    rc = BleDiscServiceFirst(conHndl, 0, uHndl)
                    IF rc==0 THEN
                        //HandlerPrimSvc() will exit with 0 when operation is complete
                        WAITEVENT
                        uu$ = "112233445566778899AABBCCDDEFF00"
                        PRINT "\nScan for service with custom uuid "; uu$
WAITEVENT
ENDIF
ENDIF
ENDIF
CloseConnections()
ENDIF
ENDFUNC

//==============================================================================
// EVDISCPRIMSVC event handler
//==============================================================================
FUNCTION HandlerPrimSvc(cHndl, svcUuid, sHndl, eHndl) AS INTEGER
PRINT "\nEVDISCPRIMSVC :
PRINT " cHndl=";cHndl
PRINT " svcUuid=";integer.h svcUuid
PRINT " sHndl=";sHndl
PRINT " eHndl=";eHndl
IF sHndl == 0 THEN
  PRINT "\nScan complete"
  EXITFUNC 0
ELSE
  rc = BleDiscServiceNext(cHndl)
  IF rc != 0 THEN
    PRINT "\nScan abort"
    EXITFUNC 0
  ENDIF
ENDIF
endfunc 1

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

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "11223445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

uuid$ = "1122DEAD5566778899AABBCCDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

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

Expected Output:

Advertising, and GATT Client is open
- Connected, so scan remote GATT Table for ALL services
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01FE01 sHndl=1 eHndl=3
BLEDISCSERVICEFIRST and BLEDISCERVICENEXT are both extension functions.

**BleDiscCharFirst / BleDiscCharNext**

**FUNCTIONS**

These pair of functions are used to scan the remote GATT server for characteristics in a service with the help of the EVDISCCHAR message event. When called, a handler for the event message must be registered because the discovered characteristics information is passed back in that message.

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

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

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

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

**EVDISCCHAR event message**

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

- Connection Handle
- Characteristic UUID Handle
- Characteristic properties
- Handle for the value attribute of the characteristic
- Included Service UUID Handle

---

Embedded Wireless Solutions Support Center: 158
http://ews-support.lairdtech.com
www.lairdtech.com/wireless
If no more characteristics were discovered because the end of the table was reached, then all parameters contain zero apart from the Connection Handle.

‘Characteristic Uuid Handle’ contains the UUID of the characteristic and supplied as a handle.

‘Characteristic Properties’ contains the properties of the characteristic and is a bit mask as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set if BROADCAST is enabled</td>
</tr>
<tr>
<td>1</td>
<td>Set if READ is enabled</td>
</tr>
<tr>
<td>2</td>
<td>Set if WRITE_WITHOUT_RESPONSE is enabled</td>
</tr>
<tr>
<td>3</td>
<td>Set if WRITE is enabled</td>
</tr>
<tr>
<td>4</td>
<td>Set if NOTIFY is enabled</td>
</tr>
<tr>
<td>5</td>
<td>Set if INDICATE is enabled</td>
</tr>
<tr>
<td>6</td>
<td>Set if AUTHENTICATED_SIGNED_WRITE is enabled</td>
</tr>
<tr>
<td>7</td>
<td>Set if RELIABLE_WRITE is enabled</td>
</tr>
</tbody>
</table>

‘Handle for the Value Attribute of the Characteristic’ is the handle for the value attribute and is the value to store to keep track of important characteristics in a GATT server for later read/write operations.

‘Included Service Uuid Handle’ is for future use and is always 0.

BLEDISCCHARFIRST (connHandle, charUuidHandle, startAttrHandle, endAttrHandle)

A typical pseudo code for discovering characteristic involves first calling BleDiscCharFirst() with information obtained from a primary services scan and then waiting for the EVDISCCHAR event message and depending on the information returned in that message calling BleDiscCharNext() which in turn will result in another EVDISCCHAR event message and typically is as follows:-

   Register a handler for the EVDISCCHAR event message

   On EVDISCCHAR event message
       If Char Value Handle == 0 then scan is complete
       Else Process information then
           call BleDiscCharNext()
           if BleDiscCharNext() not OK then scan complete

   Call BleDiscCharFirst( --information from EVDISCPRIMSVC )
   If BleDiscCharFirst() ok then Wait for EVDISCCHAR

Returns

INTEGER, a result code.

The typical value is 0x0000, indicating a successful operation and it means an EVDISCCHAR event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCCHAR message is not thrown.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connHandle</td>
<td>ByVal nConnHandle AS INTEGER</td>
<td>This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.</td>
</tr>
<tr>
<td>charUuidHandle</td>
<td>ByVal charUuidHandle AS INTEGER</td>
<td>Set this to 0 if you want to scan for any characteristic in the service, otherwise this value is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().</td>
</tr>
<tr>
<td>startAttrHandle</td>
<td>ByVal startAttrHandle AS INTEGER</td>
<td>This is the attribute handle from where the scan for characteristic is started and is acquired by doing a primary services scan, which returns the start and end handles of services.</td>
</tr>
</tbody>
</table>
**endAttrHandle**

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

---

**Interactive Command**

No

---

**BLEDISCCCHARNEXT** (connHandle)

Calling this assumes that BleDiscCharFirst() has been called at least once to set up the internal characteristics scanning state machine. It scans for the next characteristic.

---

**Returns**

INTEGER, a result code.
The typical value is 0x0000, indicating a successful operation and it means an EVDISCCHAR event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCCHAR message is not thrown.

---

**Arguments:**

- **connHandle**
  
  byVal nConnHandle AS INTEGER
  
  This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.

---

**Interactive Command**

No

---

```plaintext
//Example :: BleDiscCharFirst.Next.sb
//
//Remote server has 1 prim service with 16 bit uuid and 8 characteristics where
// 5 uuids are 16 bit and 3 are 128 bit
// 3 of the 16 bit uuid are the same value 0xDEAD and
// 2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGATTcTblDiscChar.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

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

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

//==============================================================================
// Close connections so that we can run another app without problems
//==============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB
```
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote GATT Table for first service"
        PRINT "\n- and a characteric scan will be initiated in the event"
        rc = BleDiscServiceFirst(conHndl,0,0)
        IF rc==0 THEN
            //wait for start and end handles for first primary service
            WAITEVENT
            PRINT "\n\nScan for characteristic with uuid = 0xDEAD"
            uHndl = BleHandleUuid16(0xDEAD)
            rc = BleDiscCharFirst(conHndl,uHndl,sAttr,eAttr)
            IF rc == 0 THEN
                //HandlerCharDisc() will exit with 0 when operation is complete
                WAITEVENT
                ENDIF
            ENDIF
            ENDIF
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC

FUNCTION HandlerPrimSvc(cHndl,svcUuid,sHndl,eHndl) AS INTEGER
    PRINT "\nEVDISCPRIMSVC :"
    PRINT " cHndl=";cHndl
    PRINT " svcUuid=";integer.h' svcUuid
    PRINT " sHndl=";sHndl
    PRINT " eHndl=";eHndl
    IF sHndl == 0 THEN
        PRINT "\nPrimary Service Scan complete"
        EXITFUNC 0
    ELSE
        PRINT "\nGot first primary service so scan for ALL characteristics"
        sAttr = sHndl
        eAttr = eHndl
        rc = BleDiscCharFirst(conHndl,0,sAttr,eAttr)
        IF rc != 0 THEN
            PRINT "\nScan characteristics failed"
            EXITFUNC 0
        ENDIF
    ENDIF
ENDIF
endfunc 1
function HandlerCharDisc(cHndl, cUuid, cProp, hVal, isUuid) as integer

    print "\nEVDISCCHAR :
    print " cHndl=";cHndl
    print " chUuid=";integer.h' cUuid
    print " Props=";cProp
    print " valHndl=";hVal
    print " ISvcUuid=";isUuid
    IF hVal == 0 THEN
        PRINT "\nCharacteristic Scan complete"
        EXITFUNC 0
    ELSE
        rc = BleDiscCharNext(conHndl)
        IF rc != 0 THEN
            PRINT "\nCharacteristics scan abort"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc

//_Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN

uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128 (uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128 (uuid$)

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

WAITEVENT

PRINT "\nExiting..."

Expected Output:

Advertising, and GATT Client is open

- Connected, so scan remote GATT Table for first service
- and a characeristic scan will be initiated in the event
EVDISCPRIMSVCE : cHndl=3549 svcUuid=FE01FE02 sHndl=1 eHndl=17
Got first primary service so scan for ALL characteristics
EVDISCCHAR : cHndl=3549 chUuid=FE01FE02 Props=2 valHndl=3 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=5 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=7 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FB04BEEF Props=2 valHndl=9 ISvcUuid=0
BLEDISCCHARFIRST and BLEDISCCHARNEXT are both extension functions.

BleDiscDescFirst /BleDiscDescNext

FUNCTIONS

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

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

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

Depending on the size of the remote GATT server table and the connection interval, the scan of all descriptors may take many 100s of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

EVDISCDESC event message

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

- Connection Handle
- Descriptor Uuid Handle
- Handle for the Descriptor in the remote GATT Table

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

‘Descriptor Uuid Handle’ contains the UUID of the descriptor and is supplied as a handle.

‘Handle for the Descriptor in the remote GATT Table’ is the handle for the descriptor as well as the value to store to keep track of important characteristics in a GATT server for later read/write operations.
BLEDISCDESCFIRST (connHandle, descUuidHandle, charValHandle)

A typical pseudo code for discovering descriptors involves first calling BleDiscDescFirst() with information obtained from a characteristics scan and then waiting for the EVDISCDESC event message. Depending on the information returned in that message, calling BleDiscDescNext() results in another EVDISCDESC event message and typically is as follows:

1. Register a handler for the EVDISCDESC event message
2. On EVDISCDESC event message
   - If Descriptor Handle == 0 then scan is complete
   - Else Process information then
     - call BleDiscDescNext()
       - if BleDiscDescNext() not OK then scan complete
3. Call BleDiscDescFirst() --information from EVDISCCHAR
4. If BleDiscDescFirst() ok then Wait for EVDISCDESC

Returns

INTEGER, a result code.
The typical value is 0x0000, indicating a successful operation and it means an EVDISCDESC event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCDESC message is not thrown.

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>connHandle</td>
<td>ByVal nConnHandle AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.</td>
</tr>
<tr>
<td>descUuidHandle</td>
<td>ByVal descUuidHandle AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>Set this to 0 if you want to scan for any descriptor in the characteristic, otherwise this value is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().</td>
</tr>
<tr>
<td>charValHandle</td>
<td>ByVal charValHandle AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>This is the value attribute handle of the characteristic on which the descriptor scan is to be performed. It will have been acquired from an EVDISCCHAR event.</td>
</tr>
</tbody>
</table>

Interactive Command

No

BLEDISCDESCNEXT (connHandle)

Calling this assumes that BleDiscCharFirst() has been called at least once to set up the internal characteristics scanning state machine and that BleDiscDescFirst() has been called at least once to start the descriptor discovery process.

Returns

INTEGER, a result code.
The typical value is 0x0000, indicating a successful operation and it means an EVDISCDESC event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCDESC message is not thrown.

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>connHandle</td>
<td>ByVal nConnHandle AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.</td>
</tr>
</tbody>
</table>

Interactive

No
// Example :: BleDiscDescFirst.Next.sb
//
// Remote server has 1 prim service with 16 bit uuid and 1 characteristics
// which contains 8 descriptors, that are ...
// 5 uuids are 16 bit and 3 are 128 bit
// 3 of the 16 bit uuid are the same value 0xDEAD and
// 2 of the 128 bit uuids are also the same 11233456678899AABBCCDDEEFF
//
// Server created using BleGATTcTblDiscDesc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc, at$, conHndl, uHndl, uuid$, sAttr, eAttr, cValAttr

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

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

//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgId==1 THEN
        PRINT "\n\nDisconnected"
        EXITFUNC 0
    ELSEIF nMsgId==0 THEN
        PRINT "\n\nConnected, so scan remote GATT Table for first service"
        PRINT "\n\nand a characteristic scan will be initiated in the event"
        rc = BleDiscServiceFirst(conHndl, 0, 0)
        IF rc==0 THEN
            //wait for start and end handles for first primary service
            WAITEVENT
            PRINT "\n\nScan for descriptors with uuid = 0xDEAD"
            uHndl = BleHandleUuid16(0xDEAD)
            rc = BleDiscDescFirst(conHndl, uHndl, cValAttr)
        ENDIF
    ENDIF
ENDFUNCTION HndlrBleMsg()}
uu$ = StrDehexize$(uu$)
uHndl = BleHandleUuid128(uu$)
rc = BleDiscDescFirst(conHndl, uHndl, cValAttr)
    IF rc==0 THEN
        //HandlerDescDisc() will exit with 0 when operation is complete
        WAITEVENT
    ENDIF
ENDIF
ENDIF
CloseConnections()
ENDFUNC

//==============================================================================
// EVDISCPRIMSVC event handler
//==============================================================================
FUNCTION HandlerPrimSvc(cHndl, svcUuid, sHndl, eHndl) AS INTEGER
    PRINT "\nEVDISCPRIMSVC :"
    PRINT " cHndl=";cHndl
    PRINT " svcUuid=";integer.h' svcUuid
    PRINT " sHndl=";sHndl
    PRINT " eHndl=";eHndl
    IF sHndl == 0 THEN
        PRINT "\nPrimary Service Scan complete"
        EXITFUNC 0
    ELSE
        PRINT "\nGot first primary service so scan for ALL characteristics"
        sAttr = sHndl
eAttr = eHndl
        rc = BleDiscCharFirst(conHndl, 0, sAttr, eAttr)
        IF rc != 0 THEN
            PRINT "\nScan characteristics failed"
            EXITFUNC 0
        ENDIF
    ENDIF
ENDIF
endfunc 1

//==============================================================================
// EVDISCCHAR event handler
//==============================================================================
FUNCTION HandlerCharDisc(cHndl, cUuid, cProp, hVal, isUuid) AS INTEGER
    PRINT "\nEVDISCCHAR :"
    PRINT " cHndl=";cHndl
    PRINT " chUuid=";integer.h' cUuid
    PRINT " Props=";cProp
    PRINT " valHndl=";hVal
    PRINT " ISvcUuid=";isUuid
    IF hVal == 0 THEN
        PRINT "\nCharacteristic Scan complete"
        EXITFUNC 0
    ELSE
    PRINT "\nGot first characteristic service at handle ";hVal
    PRINT "\nScan for ALL Descs"
    cValAttr = hVal
    rc = BleDiscDescFirst(conHndl, 0, cValAttr)
    IF rc != 0 THEN
        PRINT "\nScan descriptors failed"
        EXITFUNC 0
    ENDIF
ENDIF
endfunc 1
'//==============================================================================
// EVDISCDESC event handler
//==============================================================================
function HandlerDescDisc(cHndl, cUuid, hndl) as integer
    print "\nEVDISCDESC"
    print " cHndl=":cHndl
    print " dscUuid=":integer.h' cUuid
    print " dscHndl=":hndl
    IF hndl == 0 THEN
        PRINT "\nDescriptor Scan complete"
        EXITFUNC 0
    ELSE
        rc = BleDiscDescNext(cHndl)
        IF rc != 0 THEN
            PRINT "\nDescriptor scan abort"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc

//==============================================================================
// Main() equivalent
//==============================================================================
ONEVENT EVBLEMSG CALL HndlrBleMsg
OnEvent EVDISCPRIMSVC call HandlerPrimSvc
OnEvent EVDISCCHAR call HandlerCharDisc
OnEvent EVDISCDESC call HandlerDescDisc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "1122345566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

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

WAITEVENT PRINT "\nExiting..."

Expected Output:

Advertising, and GATT Client is open
- Connected, so scan remote GATT Table for first service
- and a characteristic scan will be initiated in the event
EVDISCPRIMSVC : cHndl=3790 svcUuid=FE01FE02 sHndl=1 eHndl=11
Got first primary service so scan for ALL characteristics
EVDISCCHAR : cHndl=3790 chUuid=FE01FC21 Props=2 valHndl=3 ISvcUuid=0
Got first characteristic service at handle 3
Scan for ALL Descs
EVDISCDESC cHndl=3790 dscUuid=FE01FD21 dscHndl=4
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=5
BLEDISCDESCFIRST and BLEDISCDESCNEXT are both extension functions.

**BleGattcFindChar**

**FUNCTION**

This function facilitates an efficient way of locating the details of a characteristic if the UUID is known along with the UUID of the service containing it. The results are delivered in an EVFINDCHAR event message. If the GATT server table has multiple instances of the same service/characteristic combination then this function works because, in addition to the UUID handles to be searched for, it also accepts instance parameters which are indexed from 0. This means the fourth instance of a characteristic with the same UUID in the third instance of a service with the same UUID is located with index values 3 and 2 respectively.

Given that the results are returned in an event message, a handler must be registered for the EVFINDCHAR event.

Depending on the size of the remote GATT server table and the connection interval, the search of the characteristic may take many 100s of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

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

**EVFINDCHAR event message**

This event message is thrown if BleGATTcFindChar() returns a success. The message contains the following INTEGER parameters:

- Connection Handle
- Characteristic Properties
- Handle for the Value Attribute of the Characteristic
- Included Service Uuid Handle
If the specified instance of the service/characteristic is not present in the remote GATT server table, then all parameters contain zero apart from the Connection Handle.

‘Characteristic Properties’ contains the properties of the characteristic and is a bit mask as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set if BROADCAST is enabled</td>
</tr>
<tr>
<td>1</td>
<td>Set if READ is enabled</td>
</tr>
<tr>
<td>2</td>
<td>Set if WRITE_WITHOUT_RESPONSE is enabled</td>
</tr>
<tr>
<td>3</td>
<td>Set if WRITE is enabled</td>
</tr>
<tr>
<td>4</td>
<td>Set if NOTIFY is enabled</td>
</tr>
<tr>
<td>5</td>
<td>Set if INDICATE is enabled</td>
</tr>
<tr>
<td>6</td>
<td>Set if AUTHENTICATED_SIGNED_WRITE is enabled</td>
</tr>
<tr>
<td>7</td>
<td>Set if RELIABLE_WRITE is enabled</td>
</tr>
<tr>
<td>15</td>
<td>Set if the characteristic has extended properties</td>
</tr>
</tbody>
</table>

‘Handle for the Value Attribute of the Characteristic’ is the handle for the value attribute and is the value to store to keep track of important characteristics in a GATT server for later read/write operations.

‘Included Service Uuid Handle’ is for future use and is always 0.

BLEGATTCFINDCHAR (connHandle, svcUidHandle, svcIndex, charUidHandle, charIndex)

A typical pseudo code for finding a characteristic involves calling BleGATTcFindChar() which in turn will result in the EVFINDCHAR event message and typically is as follows:-

1. Register a handler for the EVFINDCHAR event message
2. On EVFINDCHAR event message
   - If Char Value Handle == 0 then
     - Characteristic not found
   - Else
     - Characteristic has been found
3. Call BleGATTcFindChar()
4. If BleGATTcFindChar () ok then Wait for EVFINDCHAR

Returns

INTEGER, a result code.
The typical value is 0x0000, indicating a successful operation and it means an EVFINDCHAR event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVFINDCHAR message is not thrown.

Arguments:

- **connHandle**
  - ByVal nConnHandle AS INTEGER
  - This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.

- **svcUidHandle**
  - ByVal svcUidHandle AS INTEGER
  - Set this to the service UUID handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

- **svcIndex**
  - ByVal svcIndex AS INTEGER
  - This is the instance of the service to look for with the UUID handle svcUidHandle, where 0 is the first instance, 1 is the second, and so on.

- **charUidHandle**
  - ByVal charUidHandle AS INTEGER
  - Set this to the characteristic UUID handle which is generated either by BleHandleUuid16()
or BleHandleUuid128() or BleHandleUuidSibling() for identifying a specific instance of a characteristic.

**charIndex**

(`${charIndex}`) as `INTEGER` is the instance of the characteristic to look for with the UUID handle `charUuidHndl`, where 0 is the first instance, 1 is the second, and so on.

<table>
<thead>
<tr>
<th>Interactive Command</th>
<th>No</th>
</tr>
</thead>
</table>

```plaintext
// Example :: BleGATTcFindChar.sb
// Remote server has 5 primary services with 16 bit uuid and 3 with 128 bit uuids
// 3 of the 16 bit uuid are the same value OxDEAD and
// 2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
// Server created using BleGATTcTblFindChar.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

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

// Initialise and instantiate service, characteristic, start adverts
FUNCTION OnStartup()
  DIM rc, adRpt$, addr$, scRpt$
  rc=BleAdvRptInit(adRpt$, 2, 0, 10)
  IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
  IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$, scRpt$) : ENDIF
  IF rc==0 THEN : rc=BleAdvertStart(0, addr$, 50, 0, 0) : ENDIF
  IF rc==0 THEN : rc = BleGattcOpen(0, 0) : ENDIF
ENDFUNC rc

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

// Ble event handler
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
  DIM uu$, uHndS, uHndC, conHndl=nCtx
  IF nMsgID==1 THEN
    PRINT "\n\n- Disconnected"
    EXITFUNC 0
  ELSEIF nMsgID==0 THEN
    PRINT "\n- Connected, so scan remote GATT Table for an instance of char"
    uHndS = BleHandleUuid16(0xDEAD)
    uu$ = "112233445566778899AABBCCDDEEFF00"
    uu$ = StrDehexize$(uu$)
    uHndC = BleHandleUuid128(uu$)
    sIdx = 2
    cIdx = 1 //valHandle will be 32
  ELSE
    PRINT "\n\n- Other BLE message received"
    EXITFUNC nMsgId
  ENDIF
ENDFUNCTION HndlrBleMsg
```

Embedded Wireless Solutions Support Center: 170  Americas: +1-800-492-2320
http://ews-support.lairdtech.com  Europe: +44-1628-858-940
www.lairdtech.com/wireless  Hong Kong: +852 2923 0610
rc = BleGattcFindChar(conHndl,uHndS,sIdx,uHndC,cIdx)
IF rc==0 THEN
    //BleDiscCharFirst() will exit with 0 when operation is complete
    WAITEVENT
ENDIF
sIdx = 1
cIdx = 3 //does not exist
rc = BleGattcFindChar(conHndl,uHndS,sIdx,uHndC,cIdx)
IF rc==0 THEN
    //BleDiscCharFirst() will exit with 0 when operation is complete
    WAITEVENT
ENDIF
CloseConnections()
ENDIF
ENDFUNC 1

//===============================================
//===============================================
function HandlerFindChar(cHndl,cProp,hVal,isUuid) as integer
    print "\nEVFINDCHAR 
    print " cHndl=";cHndl
    print " Props=";cProp
    print " valHndl=";hVal
    IF hVal == 0 THEN
        PRINT "\nDid NOT find the characteristic"
    ELSE
        PRINT "\nFound the characteristic at handle ";hVal
        PRINT "\nSvc Idx=";sIdx;" Char Idx=";cIdx
    ENDIF
endfunc 0

//================================================
// Main() equivalent
//================================================
ONEVENT EVBLEMSG  CALL HndlrBleMsg
OnEvent EVFINDCHAR  call HandlerFindChar

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

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

Expected Output:
Advertising, and GATT Client is open
BLEGATTCFINDCHAR is an extension function.

**BleGattcFindDesc**

**FUNCTION**

This function facilitates an efficient way of locating the details of a descriptor if the UUID is known along with the UUID of the service and the UUID of the characteristic containing it. The results are delivered in a EVFINDDESC event message. If the GATT server table has multiple instances of the same service/characteristic/descriptor combination then this function works because, in addition to the UUID handles to be searched for, it accepts instance parameters which are indexed from 0. This means that the second instance of a descriptor in the fourth instance of a characteristic with the same UUID in the third instance of a service with the same UUID is located with index values 1, 3, and 2 respectively.

Given that the results are returned in an event message, a handler *must* be registered for the EVFINDDESC event.

Depending on the size of the remote GATT server table and the connection interval, the search of the characteristic may take many 100s of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

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

**EVFINDDESC event message**

This event message is thrown if BleGATTcFindDesc() returned a success. The message contains the following INTEGER parameters:

- Connection Handle
- Handle of the Descriptor

If the specified instance of the service/characteristic/descriptor is not present in the remote GATT server table, then all parameters contain zero apart from the Connection Handle.

‘Handle of the Descriptor’ is the handle for the descriptor and is the value to store to keep track of important descriptors in a GATT server for later read/write operations – for example CCCD’s to enable notifications and/or indications.

**BLEGATTCFINDDESC (connHndl, svcUuHndl, svcIdx, charUuHndl, charIdx, descUuHndl, descrIdx)**

A typical pseudo code for finding a descriptor involves calling BleGATTcFindDesc() which in turn results in the EVFINDDESC event message and typically is as follows:

Register a handler for the EVFINDDESC event message

On EVFINDDESC event message
If Descriptor Handle == 0 then
    Descriptor not found
Else
    Descriptor has been found

Call BleGATTcFindDesc()
If BleGATTcFindDesc() ok then Wait for EVFINDDESC

Returns
INTEGER, a result code.
The typical value is 0x0000, indicating a successful operation and it means an EVFINDDESC event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVFINDDESC message is not thrown.

Arguments:

**connHndl**
byVal connHndl AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.

**svcUuHndl**
byVal svcUuHndl AS INTEGER
Set this to the service UUID handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

**svcIdx**
byVal svcIdx AS INTEGER
This is the instance of the service to look for with the UUID handle svcUuHndl, where 0 is the first instance, 1 is the second, and so on.

**charUuHndl**
byVal charUuHndl AS INTEGER
Set this to the characteristic UUID handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

**charIdx**
byVal charIdx AS INTEGER
This is the instance of the characteristic to look for with the UUID handle charUuHndl, where 0 is the first instance, 1 is the second, and so on.

**descUuHndl**
byVal descUuHndl AS INTEGER
Set this to the descriptor uuid handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

**descIdx**
byVal descIdx AS INTEGER
This is the instance of the descriptor to look for with the UUID handle descUuHndl, where 0 is the first instance, 1 is the second, and so on.

Interactive Command
No

//Example :: BleGATTcFindDesc.sb
//Remote server has 5 prim services with 16 bit uuid and 3 with 128 bit uuids
// 3 of the 16 bit uuid are the same value 0xDEAD and
// 2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
// Server created using BleGATTcTblFindDesc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000
DIM rc,at$,connHndl,uHndl,uuid$,sIdx,cIdx,dIdx

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

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

FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$,uHndS,uHndC,uHndD
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote GATT Table for ALL services"
        uHndS = BleHandleUuid16(0xDEAD)
        uu$ = "112233445566778899AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndC = BleHandleUuid128(uu$)
        uu$ = "1122C0DE5566778899AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndD = BleHandleUuid128(uu$)
        sIdx = 2
        cIdx = 1
        dIdx = 1 // handle will be 37
        rc = BleGattcFindDesc(conHndl,uHndS,sIdx,uHndC,cIdx,uHndD,dIdx)
        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF
        sIdx = 1
        cIdx = 3
        dIdx = 4 //does not exist
        rc = BleGattcFindDesc(conHndl,uHndS,sIdx,uHndC,cIdx,uHndD,dIdx)
        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC
function HandlerFindDesc(hndl, dscHndl) as integer
  print "\nEVFINDDESC "
  print " cHndl="; hndl
  print " dscHndl="; dscHndl
  IF hndl == 0 THEN
    PRINT "\nDid NOT find the descriptor"
  ELSE
    PRINT "\nFound the descriptor at handle "; hndl
    PRINT "\nSvc Idx="; sIdx; " Char Idx="; cIdx; " desc Idx="; dIdx
  ENDIF
endfunc

//==================================
// Main() equivalent
//==================================
ONEVENT EVBLEMSG CALL HndlrBleMsg
OnEvent EVFINDDESC call HandlerFindDesc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "1122344556677889AAABCCDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD556677889AAABCCDEFFE00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

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

WAITEVENT
PRINT "\nExiting..."

Expected Output:

Advertising, and GATT Client is open
- Connected, so scan remote GATT Table for ALL services
  EVFINDDESC cHndl=1106 dscHndl=37
  Found the descriptor at handle 37
    Svc Idx=2 Char Idx=1 desc Idx=1
  EVFINDDESC cHndl=1106 dscHndl=0
  Did NOT find the descriptor
- Disconnected
Exiting...

BLEGATTFINDDESC is an extension function.

BleGattcRead / BleGattcReadData

FUNCTIONS
If the handle for an attribute is known, then these functions are used to read the content of that attribute
from a specified offset in the array of octets in that attribute value.
Given that the success or failure of this read operation is returned in an event message, a handler must be registered for the EVATTRREAD event.

Depending on the connection interval, the read of the attribute may take many 100s of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

BleGattCRead is used to trigger the procedure and BleGattCReadData is used to read the data from the underlying cache when the EVATTRREAD event message is received with a success status.

**EVATTRREAD event message**

This event message is thrown if BleGattCRead() returns a success. The message contains the following INTEGER parameters:

- Connection Handle
- Handle of the Attribute
- GATT status of the read operation.

‘**GATT status of the read operation**’ is one of the following values, where 0 implies the read was successfully expedited and the data can be obtained by calling BlePubGattClientReadData().

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Success</td>
</tr>
<tr>
<td>0x0001</td>
<td>Unknown or not applicable status</td>
</tr>
<tr>
<td>0x0100</td>
<td>ATT Error: Invalid Error Code</td>
</tr>
<tr>
<td>0x0101</td>
<td>ATT Error: Invalid Attribute Handle</td>
</tr>
<tr>
<td>0x0102</td>
<td>ATT Error: Read not permitted</td>
</tr>
<tr>
<td>0x0103</td>
<td>ATT Error: Write not permitted</td>
</tr>
<tr>
<td>0x0104</td>
<td>ATT Error: Used in ATT as Invalid PDU</td>
</tr>
<tr>
<td>0x0105</td>
<td>ATT Error: Authenticated link required</td>
</tr>
<tr>
<td>0x0106</td>
<td>ATT Error: Used in ATT as Request Not Supported</td>
</tr>
<tr>
<td>0x0107</td>
<td>ATT Error: Offset specified was past the end of the attribute</td>
</tr>
<tr>
<td>0x0108</td>
<td>ATT Error: Used in ATT as Insufficient Authorisation</td>
</tr>
<tr>
<td>0x0109</td>
<td>ATT Error: Used in ATT as Prepare Queue Full</td>
</tr>
<tr>
<td>0x010A</td>
<td>ATT Error: Used in ATT as Attribute not found</td>
</tr>
<tr>
<td>0x010B</td>
<td>ATT Error: Attribute cannot be read or written using read/write blob requests</td>
</tr>
<tr>
<td>0x010C</td>
<td>ATT Error: Encryption key size used is insufficient</td>
</tr>
<tr>
<td>0x010D</td>
<td>ATT Error: Invalid value size</td>
</tr>
<tr>
<td>0x010E</td>
<td>ATT Error: Very unlikely error</td>
</tr>
<tr>
<td>0x010F</td>
<td>ATT Error: Encrypted link required</td>
</tr>
<tr>
<td>0x0110</td>
<td>ATT Error: Attribute type is not a supported grouping attribute</td>
</tr>
<tr>
<td>0x0111</td>
<td>ATT Error: Encrypted link required</td>
</tr>
<tr>
<td>0x0112</td>
<td>ATT Error: Reserved for Future Use range #1 begin</td>
</tr>
<tr>
<td>0x0117</td>
<td>ATT Error: Reserved for Future Use range #1 end</td>
</tr>
<tr>
<td>0x0180</td>
<td>ATT Error: Application range begin</td>
</tr>
<tr>
<td>0x019F</td>
<td>ATT Error: Application range end</td>
</tr>
<tr>
<td>0x01A0</td>
<td>ATT Error: Reserved for Future Use range #2 begin</td>
</tr>
<tr>
<td>0x01DF</td>
<td>ATT Error: Reserved for Future Use range #2 end</td>
</tr>
<tr>
<td>0x01EF</td>
<td>ATT Error: Reserved for Future Use range #3 begin</td>
</tr>
<tr>
<td>0x01FC</td>
<td>ATT Error: Reserved for Future Use range #3 end</td>
</tr>
<tr>
<td>0x01FD</td>
<td>ATT Common Profile and Service Error: Client Characteristic Configuration Descriptor is improperly configured</td>
</tr>
<tr>
<td>0x01FE</td>
<td>ATT Common Profile and Service Error: Procedure Already in Progress</td>
</tr>
<tr>
<td>0x01FF</td>
<td>ATT Common Profile and Service Error: Out Of Range</td>
</tr>
</tbody>
</table>
BLEGATTCREAD (connHndl, attrHndl, offset)
A typical pseudo code for reading the content of an attribute calling BleGattcRead() which in turn results in
the EVATTRREAD event message and typically is as follows:

Register a handler for the EVATTRREAD event message

On EVATTRREAD event message
If GATT_Status == 0 then
    BleGattcReadData() //to actually get the data
Else
    Attribute could not be read

Call BleGattcRead()
If BleGattcRead() ok then Wait for EVATTRREAD

Returns INTEGER, a result code.
The typical value is 0x0000, indicating a successful operation and it means an EVATTRREAD
event message is thrown by the smartBASIC runtime engine containing the results. A non-
zero return value implies an EVATTRREAD message is not thrown.

Arguments:

connHndl byVal connHndl AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on
which the remote GATT server can be accessed. This is returned in the EVBLEMSG event
message with msgId == 0 and msgCtx is the connection handle.

attrHndl byVal attrHndl AS INTEGER
Set to the handle of the attribute to read. It is a value in the range 1 to 65535.

offset byVal offset AS INTEGER
This is the offset from which the data in the attribute is to be read.

Interactive Command No

BLEGATTCREADDATA (connHndl, attrHndl, offset, attrData$)
This function is used to collect the data from the underlying cache when the EVATTRREAD event message has
a success GATT status code.

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

Arguments:

connHndl byVal connHndl AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on
which the remote GATT server can be accessed. This is returned in the EVBLEMSG event
message with msgId == 0 and msgCtx is the connection handle.

attrHndl byRef attrHndl AS INTEGER
The handle for the attribute that was read is returned in this variable. It is the same as the
one supplied in BleGATTcRead, but supplied here so that the code can be stateless.

offset byRef offset AS INTEGER
The offset into the attribute data that was read is returned in this variable. It is the same as
the one supplied in BleGATTcRead, but supplied here so that the code can be stateless.

attrData$ byRef attrData$ AS STRING
The attribute data which was read is supplied in this parameter.
//Example :: BleGattcRead.sb
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//Server created using BleGattcTblRead.sub invoked in _OpenMcp.scr
//using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,nOff,atHndl

//==============================================================================
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BlleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BlleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BlleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BlleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BlleGattcOpen(0,0) : ENDIF
ENDFUNC rc

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

//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\nDisconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n\nConnected, so read attibute handle 3"
        atHndl = 3
        nOff = 0
        rc=BlleGattcRead(conHndl,atHndl,nOff)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\nread attibute handle 300 which does not exist"
        atHndl = 300
        nOff = 0
        rc=BlleGattcRead(conHndl,atHndl,nOff)
        IF rc==0 THEN
            WAITEVENT
function HandlerAttrRead(cHndl, aHndl, nSts) as integer
  dim nOfst, nAhndl, at$
  print "\nEVATTRREAD ", cHndl, " attrHndl=", aHndl, " status=", nSts
  if nSts == 0 then
    print "\nAttribute read OK"
    rc = BleGattcReadData(cHndl, nAhndl, nOfst, at$)
    print "\nData = ", StrHexize$(at$)
    print " Offset= ", nOfst
    print " Len=", strlen(at$)
  else
    print "\nFailed to read attribute"
  endif
endfunc

// Main() equivalent
// ==============================================================
ONEVENT EVBLEMSG CALL HndlrBleMsg
OnEvent EVATTRREAD call HandlerAttrRead
IF OnStartup() == 0 THEN
  PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
  PRINT "\nFailure OnStartup"
ENDIF
WAITEVENT
PRINT "\nExiting..."

Expected Output:

- Connected, so read attribute handle 3
  EVATTRREAD cHndl=2960 attrHndl=3 status=00000000
  Attribute read OK
  Data = 00000000 Offset= 0 Len=4
  handle = 3
  read attribute handle 300 which does not exist
  EVATTRREAD cHndl=2960 attrHndl=300 status=00000101
  Failed to read attribute

- Disconnected
  Exiting...

BLEGATTCREAD and BLEGATTREADDATA are extension functions.

BleGattcWrite

Embedded Wireless Solutions Support Center: 179 Americas: +1-800-492-2320
http://ews-support.lairdtech.com Europe: +44-1628-858-940
www.lairdtech.com/wireless Hong Kong: +852 2923 0610
FUNCTION

If the handle for an attribute is known then this function is used to write into an attribute starting at offset 0. The acknowledgement is returned via a EVATTRWRITE event message.

Given that the success or failure of this write operation is returned in an event message, a handler must be registered for the EVATTRWRITE event.

Depending on the connection interval, the write to the attribute may take many 100s of milliseconds. While this is in progress, it is safe to do other non-GATT related operations such as servicing sensors and displays or any of the onboard peripherals.

**EVATTRWRITE event message**

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

- Connection Handle
- Handle of the Attribute
- GATT status of the write operation.

‘GATT status of the write operation’ is one of the following values, where 0 implies the write was successfully expedited.

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Success</td>
</tr>
<tr>
<td>0x0001</td>
<td>Unknown or not applicable status</td>
</tr>
<tr>
<td>0x0100</td>
<td>ATT Error: Invalid Error Code</td>
</tr>
<tr>
<td>0x0101</td>
<td>ATT Error: Invalid Attribute Handle</td>
</tr>
<tr>
<td>0x0102</td>
<td>ATT Error: Read not permitted</td>
</tr>
<tr>
<td>0x0103</td>
<td>ATT Error: Write not permitted</td>
</tr>
<tr>
<td>0x0104</td>
<td>ATT Error: Used in ATT as Invalid PDU</td>
</tr>
<tr>
<td>0x0105</td>
<td>ATT Error: Authenticated link required</td>
</tr>
<tr>
<td>0x0106</td>
<td>ATT Error: Used in ATT as Request Not Supported</td>
</tr>
<tr>
<td>0x0107</td>
<td>ATT Error: Offset specified was past the end of the attribute</td>
</tr>
<tr>
<td>0x0108</td>
<td>ATT Error: Used in ATT as Insufficient Authorisation</td>
</tr>
<tr>
<td>0x0109</td>
<td>ATT Error: Used in ATT as Prepare Queue Full</td>
</tr>
<tr>
<td>0x010A</td>
<td>ATT Error: Used in ATT as Attribute not found</td>
</tr>
<tr>
<td>0x010B</td>
<td>ATT Error: Attribute cannot be read or written using read/write blob requests</td>
</tr>
<tr>
<td>0x010C</td>
<td>ATT Error: Encryption key size used is insufficient</td>
</tr>
<tr>
<td>0x010D</td>
<td>ATT Error: Invalid value size</td>
</tr>
<tr>
<td>0x010E</td>
<td>ATT Error: Very unlikely error</td>
</tr>
<tr>
<td>0x010F</td>
<td>ATT Error: Encrypted link required</td>
</tr>
<tr>
<td>0x0110</td>
<td>ATT Error: Attribute type is not a supported grouping attribute</td>
</tr>
<tr>
<td>0x0111</td>
<td>ATT Error: Encrypted link required</td>
</tr>
<tr>
<td>0x0112</td>
<td>ATT Error: Reserved for Future Use range #1 begin</td>
</tr>
<tr>
<td>0x017F</td>
<td>ATT Error: Reserved for Future Use range #1 end</td>
</tr>
<tr>
<td>0x0180</td>
<td>ATT Error: Application range begin</td>
</tr>
<tr>
<td>0x019F</td>
<td>ATT Error: Application range end</td>
</tr>
<tr>
<td>0x01A0</td>
<td>ATT Error: Reserved for Future Use range #2 begin</td>
</tr>
<tr>
<td>0x01DF</td>
<td>ATT Error: Reserved for Future Use range #2 end</td>
</tr>
<tr>
<td>0x01E0</td>
<td>ATT Error: Reserved for Future Use range #3 begin</td>
</tr>
<tr>
<td>0x01FC</td>
<td>ATT Error: Reserved for Future Use range #3 end</td>
</tr>
<tr>
<td>0x01FD</td>
<td>ATT Error: Common Profile and Service Error:</td>
</tr>
<tr>
<td></td>
<td>Client Characteristic Configuration Descriptor (CCCD)</td>
</tr>
</tbody>
</table>
BLEGATTWRITE (connHndl, attrHndl, attrData$)

A typical pseudo code for writing to an attribute which results in the EVATTRWRITE event message and typically is as follows:

Register a handler for the EVATTRWRITE event message

On EVATTRWRITE event message
   If GATT_Status == 0 then
       Attribute was written successfully
   Else
       Attribute could not be written

Call BleGattcWrite()
If BleGattcWrite() ok then Wait for EVATTRWRITE

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

Arguments:

<table>
<thead>
<tr>
<th>connHndl</th>
<th>byVal connHndl AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>attrHndl</th>
<th>byVal attrHndl AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The handle for the attribute that is to be written to.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>attrData$</th>
<th>byRef attrData$ AS STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The attribute data to write.</td>
</tr>
</tbody>
</table>

Interactive Command No

//Example :: BleGATTcWrite.sb
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGATTcTblWrite.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000
DIM rc,at$,conHndl,uHndl,atHndl

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

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

//==============================================================================
// Blem event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n\- Connected, so write to attribute handle 3"
        atHndl = 3
        at$="\01\02\03\04"
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\nwrite to attribute handle 300 which does not exist"
        atHndl = 300
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

//==============================================================================
// Main() equivalent
//==============================================================================
ONEVENT EVBLEMSG CALL HndlrBleMsg
OnEvent EVATTRWRITE call HandlerAttrWrite

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

function HandlerAttrWrite(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRWRITE "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute write OK"
    else
        print "\nFailed to write attribute"
    endif
endfunc 0
PRINT "\nFailure OnStartup"
ENDIF

WAIT EVENT
PRINT "\nExiting..."

Expected Output:

Advertising, and GATT Client is open
- Connected, so read attribute handle 3
  EVATTRWRITE  chndl=2687 attrHndl=3 status=00000000
  Attribute write OK
  Write to attribute handle 300 which does not exist
  EVATTRWRITE  chndl=2687 attrHndl=300 status=00000101
  Failed to write attribute
- Disconnected
Exiting...

BLEGATTWRITE is an extension function.

BleGattcWriteCmd

FUNCTION
If the handle for an attribute is known, then this function is used to write into an attribute at offset 0 when no acknowledgment response is expected. The signal that the command has actually been transmitted and that the remote link layer has acknowledged is by the EVNOTIFYBUF event.

Note: The acknowledgement received for the BleGattcWrite() command is from the higher level GATT layer. Do not confuse this with the link layer ACK.

All packets are acknowledged at link layer level. If a packet fails to get through, then that condition manifests as a connection drop due to the link supervision timeout.

Given that the transmission and link layer ACK of this write operation is indicated in an event message, a handler must be registered for the EVNOTIFYBUF event.

Depending on the connection interval, the write to the attribute may take many 100s of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

EVNOTIFYBUF event

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

BLEGATTWRITECMD (connHndl, attrHndl, attrData$)

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

Register a handler for the EVNOTIFYBUF event message

On EVNOTIFYBUF event message
  Can now send another write command
Call `BleGattcWriteCmd()`
If `BleGattcWrite() ok` then Wait for EVNOTIFYBUF

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The typical value is 0x0000, indicating a successful read.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td><code>connHndl</code></td>
<td><code>byVal connHndl AS INTEGER</code>&lt;br&gt;This is the connection handle as returned in the on-connect event for the connection on which the remote GATT Server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.</td>
</tr>
<tr>
<td><code>attrHndl</code></td>
<td><code>byVal attrHndl AS INTEGER</code>&lt;br&gt;The handle for the attribute that is to be written to.</td>
</tr>
<tr>
<td><code>attrData$</code></td>
<td><code>byRef attrData$ AS STRING</code>&lt;br&gt;The attribute data to write.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```plaintext
//Example :: BleGATTcWriteCmd.sb
//Remote server has 3 prim services with 16 bit uuid. First service has one characteristic whose value attribute is at handle 3 and has read/write props
//Server created using BleGATTcTblWriteCmd.sub invoked in _OpenMcp.scr
//using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,atHndl

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

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

//==============================================
// Ble event handler
//==============================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
    ENDIF
ENDFUNC
```
EXITFUNC 0
ELSEIF nMsgID==0 THEN
  PRINT "\n- Connected, so write to attribute handle 3"
  atHndl = 3
  at$="\01\02\03\04"
  rc=BleGattcWriteCmd(conHndl,atHndl,at$)
  IF rc==0 THEN
    WAITEVENT
  ENDIF
  PRINT "\n- write again to attribute handle 3"
  atHndl = 3
  at$="\05\06\07\08"
  rc=BleGattcWriteCmd(conHndl,atHndl,at$)
  IF rc==0 THEN
    WAITEVENT
  ENDIF
  PRINT "\n- write again to attribute handle 3"
  atHndl = 3
  at$="\09\0A\0B\0C"
  rc=BleGattcWriteCmd(conHndl,atHndl,at$)
  IF rc==0 THEN
    WAITEVENT
  ENDIF
  PRINT "\n- write to attribute handle 300 which does not exist"
  atHndl = 300
  rc=BleGattcWriteCmd(conHndl,atHndl,at$)
  IF rc==0 THEN
    PRINT "\n- Even when the attribute does not exist an event will occur"
    WAITEVENT
  ENDIF
  CloseConnections()
ENDIF
ENDFUNC

'//==============================================================================
'//==============================================================================
function HandlerNotifyBuf() as integer
  print "\nEVNOTIFYBUF Event"
eendfunc 0 '/need to progress the WAITEVENT

'//==============================================================================
// Main() equivalent
'//==============================================================================
ONEVENT EVBLEMSG CALL HndlrBleMsg
OnEvent EVNOTIFYBUF call HandlerNotifyBuf

IF OnStartup()==0 THEN
  PRINT "\nAdvertising, and GATT Client is open"
ELSE
  PRINT "\nFailure OnStartup"
ENDIF
WAITEVENT
PRINT "\nExiting..."

Expected Output:

Advertising, and GATT Client is open
- Connected, so write to attribute handle 3
  EVNOTIFYBUF Event
- write again to attribute handle 3
  EVNOTIFYBUF Event
- write again to attribute handle 3
  EVNOTIFYBUF Event
write to attribute handle 300 which does not exist
Even when the attribute does not exist an event will occur
EVNOTIFYBUF Event
- Disconnected
Exiting...

BLEGATTCWRITECMD is an extension function.

**BleGattNotifyRead**

**FUNCTION**

A GATT server has the ability to notify or indicate the value attribute of a characteristic when enabled via the Client Characteristic Configuration Descriptor (CCCD). This means data arrives from a GATT server at any time and must be managed so that it can synchronised with the **smartBASIC** runtime engine.

Data arriving via a notification does not require GATT acknowledgements, however indications require them. This GATT client manager saves data arriving via a notification in the same ring buffer for later extraction using the command BleGattcNotifyRead(); for indications, an automatic GATT acknowledgement is sent when the data is saved in the ring buffer. This acknowledgement happens even if the data is discarded because the ring buffer is full. If the data must not be acknowledged when it is discarded on a full buffer, then set the flags parameter in the BleGattcOpen() function where the GATT client manager is opened.

In the case when an ACK is NOT sent on data discard, the GATT server is throttled and no further data is notified or indicated by it until BleGattNotifyRead() is called to extract data from the ring buffer to create space and it triggers a delayed acknowledgement.

When the GATT client manager is opened using BleGattcOpen(), it is possible to specify the size of the ring buffer. If a value of 0 is supplied, then a default size is created. SYSINFO(2019) in a **smartBASIC** application or the interactive mode command AT I 2019 returns the default size. Likewise SYSINFO(2020) or the command AT I 2020 returns the maximum size.

Data that arrives via notifications or indications get stored in the ring buffer. At the same time, an EVATTRNOTIFY event is thrown to the **smartBASIC** runtime engine. This is an event, in the same way an incoming UART receive character generates an event; that is, no data payload is attached to the event.

**EVATTRNOTIFY event message**

This event is thrown when an notification or an indication arrives from a GATT server. The event contains no parameters. Please note that if one notification/indication arrives or many, like in the case of UART events, the same event mask bit is asserted. The **smartBASIC** application is informed that it must go and service the ring buffer using the function BleGattNotifyRead.

**BLEGATTNOTIFYREAD (connHndl, attrHndl, attrData$, discardCount)**

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

```c
Register a handler for the EVATTRNOTIFY event message

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

 предложения выделены жирным шрифтом
Returns

INTEGER, a result code. The typical value is 0x0000, indicating data was successful read.

Arguments:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connHndl</td>
<td>byRef connHndl AS INTEGER On exit, this is the connection handle of the GATT server that sent the notification or indication.</td>
</tr>
<tr>
<td>attrHndl</td>
<td>byRef attrHndl AS INTEGER On exit, this is the handle of the characteristic value attribute in the notification or indication.</td>
</tr>
<tr>
<td>attrData$</td>
<td>byRef attrData$ AS STRING On exit, this is the data of the characteristic value attribute in the notification or indication. It is always from offset 0 of the source attribute.</td>
</tr>
<tr>
<td>discardedCount</td>
<td>byRef discardedCount AS INTEGER On exit, this should contain 0. It signifies the total number of notifications or indications that got discarded because the ring buffer in the GATT client manager was full. If non-zero values are encountered, it is recommended that the ring buffer size be increased by using BleGattcClose() when the GATT client was opened using BleGattcOpen().</td>
</tr>
</tbody>
</table>

Interactive Command

No

//Example :: BleGATTcNotifyRead.sb
//
// Server created using BleGattcTblNotifyRead.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000
//
// Characteristics at handle 15 has notify (16==cccd)
// Characteristics at handle 18 has indicate (19==cccd)

DIM rc,at$,conHndl,uHndl,atHndl

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

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

//====================================
// Ble event handler
//====================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
conHndl=nCtx
IF nMsgID==1 THEN
PRINT "\n\nDisconnected"
EXITFUNC 0
ELSEIF nMsgID==0 THEN
PRINT "\n\nConnected, so enable notification for char with cccd at 16"
atHndl = 16
at$="\01\00"
rc=BleGattcWrite(conHndl,atHndl,at$)
IF rc==0 THEN
WAITEVENT
ENDIF
PRINT "\n\nenable indication for char with cccd at 19"
atHndl = 19
at$="\02\00"
rc=BleGattcWrite(conHndl,atHndl,at$)
IF rc==0 THEN
WAITEVENT
ENDIF
ENDIF
ENDFUNC

function HandlerAttrWrite(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\n\nEVATTRWRITE 
print " chndl";cHndl
print " attrHndl";aHndl
print " status";integer.h' nSts
if nSts == 0 then
    print "Attribute write OK"
else
    print "Failed to write attribute"
endif
endfunc

function HandlerAttrNotify() as integer
    dim chndl,aHndl,att$,dscd
    print "\nEVATTRNOTIFY Event"
    rc=BleGattcNotifyRead(cHndl,aHndl,att$,dscd)
    print "\n BleGattcNotifyRead()"
    if rc==0 then
        print " chndl";cHndl
        print " attrHndl";aHndl
        print " data";StrHexize$(att$)
        print " discarded";dscd
    else
        print " failed with ";integer.h' rc
    endif
Expected Output:

```
Advertising, and GATT Client is open

- Connected, so enable notification for char with cccd at 16
  EVATTRWRITE  cHndl=877 attrHndl=16 status=00000000
  Attribute write OK
- enable indication for char with cccd at 19
  EVATTRWRITE  cHndl=877 attrHndl=19 status=00000000
  Attribute write OK
  EVATRNOTIFY Event
    BleGATTcNotifyRead() cHndl=877 attrHndl=15 data=BAADC0DE discarded=0
  EVATRNOTIFY Event
    BleGATTcNotifyRead() cHndl=877 attrHndl=18 data=DEADBEEF discarded=0
  EVATRNOTIFY Event
    BleGATTcNotifyRead() cHndl=877 attrHndl=15 data=BAADC0DE discarded=0
  EVATRNOTIFY Event
    BleGATTcNotifyRead() cHndl=877 attrHndl=18 data=DEADBEEF discarded=0
```

BLEGATTCNOTIFYREAD is an extension function.

### Attribute Encoding Functions

Data for characteristics are stored in value attributes, arrays of bytes. Multibyte Characteristic Descriptors content is stored similarly. Those bytes are manipulated in smartBASIC applications using STRING variables.

The Bluetooth specification stipulates that multibyte data entities are stored in little endian format and so all data manipulation is done similarly. Little endian means that a multibyte data entity is stored so that lowest significant byte is positioned at the lowest memory address and likewise, when transported, the lowest byte is on the wire first.

This section describes all the encoding functions which allow those strings to be written in smaller byte-wise subfields in a more efficient manner compared to the generic STRXXXX functions that are made available in smartBASIC.

**Note:** CCCD and SCCD descriptors are special cases; they have two bytes which are treated as 16-bit integers. This is reflected in smartBASIC applications so that INTEGER variables are used to manipulate those values instead of STRINGS.
**BleEncode8**

**FUNCTION**

This function overwrites a single byte in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**BLEENCODER8 (attr$, nData, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>attr$</td>
<td>byRef attr$ AS STRING This argument is the string that is written to an attribute.</td>
</tr>
<tr>
<td>nData</td>
<td>byVal nData AS INTEGER The least significant byte of this integer is saved. The rest is ignored.</td>
</tr>
<tr>
<td>nIndex</td>
<td>byVal nIndex AS INTEGER This is the zero-based index into the string attr$ where the new fragment of data is written to. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```plaintext
//Example :: BleEncode8.sb
DIM rc
DIM attr$
attr$="Laird"
PRINT "\nattr$=";attr$

//Remember: - 4 bytes are used to store an integer on the WB45
//write 'C' to index 2 -- '111' will be ignored
rc=BleEncode8(attr$,0x11143,2)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)
//write 'B' to index 1
rc=BleEncode8(attr$,0x42,1)
//write 'D' to index 3
rc=BleEncode8(attr$,0x44,3)
//write 'y' to index 7 -- attr$ will be extended
rc=BleEncode8(attr$,0x67, 7)
PRINT "\nattr$ now = ";attr$
```

**Expected Output:**

```
attr$=Laird
attr$ now = ABCD disadvantaged
```
BLEENCODE8 is an extension function.

**BleEncode16**

**FUNCTION**

This function overwrites two bytes in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**BLEENCODE16 (attr$, nData, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>

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

- **nData** byVal nData AS INTEGER
  The two least significant bytes of this integer is saved. The rest is ignored.

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

**Interactive Command**

No

```asm
//Example :: BleEncode16.sb
DIM rc, attr$
attr$="Laird"
PRINT "\nattr$=";attr$

//write 'CD' to index 2
rc=BleEncode16(attr$, 0x4443, 2)

//write 'AB' to index 0 - '2222' will be ignored
rc=BleEncode16(attr$, 0x22224241, 0)

//write 'EF' to index 3
rc=BleEncode16(attr$, 0x4645, 4)

PRINT "\nattr$ now = ";attr$
```
Expected Output:

```
attr$=Laird
attr$ now = ABCDEF
```

BLEENCODE16 is an extension function.

**BleEncode24**

**FUNCTION**

This function overwrites three bytes in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**BLEENCODE24 (attr$, nData, nIndex)**

**Returns**

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

**Arguments:**

<table>
<thead>
<tr>
<th>attr$</th>
<th>byRef attr$ AS STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>This argument is the string that is written to an attribute.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nData</th>
<th>byVal nData AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>The three least significant bytes of this integer is saved. The rest is ignored.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nIndex</th>
<th>byVal nIndex AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.</td>
<td></td>
</tr>
</tbody>
</table>

**Interactive Command**

No

```
//Example :: BleEncode24.sb

DIM rc
DIM attr$ : attr$="Laird"

//write 'BCD' to index 1
rc=BleEncode24(attr$,0x444342,1)

//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)

//write 'EF' to index 4
rc=BleEncode16(attr$,0x4645,4)

PRINT "attr$=";attr$
```

Expected Output:

```
attr$=ABCDEF
```

BLEENCODE24 is an extension function.

**BleEncode32**

**FUNCTION**
This function overwrites four bytes in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**BLEENCODE32** (attr$, nData, nIndex)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| attr$  | byRef attr$ AS STRING  
This argument is the string that is written to an attribute |
| nData  | byVal nData AS INTEGER  
The four bytes of this integer is saved. The rest is ignored. |
| nIndex | byVal nIndex AS INTEGER  
This is the zero based index into the string attr$ where the new fragment of data is written.  
If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails. |
| Interactive Command | No |

//Example :: BleEncode32.sb

```
DIM rc
DIM attr$ : attr$="Laird"

//write 'BCDE' to index 1
rc=BleEncode32(attr$,0x45444342,1)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)
PRINT "attr$=";attr$
```

Expected Output:

```
attr$=ABCDE
```

BLEENCODE32 is an extension function.

**BleEncodeFLOAT**

**FUNCTION**

This function overwrites four bytes in a string at a specified offset. If the string is not long enough, it is extended with the new extended block uninitialized and then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**BLEENCODEFLOAT** (attr$, nMatissa, nExponent, nIndex)

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

### attr$

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

### nMatissa

- **byVal nMantissa AS INTEGER**
  - This value must be in the range -8388600 to +8388600 or the function fails. The data is written in little endian so that the least significant byte is at the lower memory address.
  - **Note:** The range is not +/- 2048 because after encoding the following 2 byte values have special meaning:
    - 0x007FFFFF - NaN (Not a Number)
    - 0x00800000 - NRes (Not at this resolution)
    - 0x007FFFFE - + INFINITY
    - 0x00800002 - - INFINITY
    - 0x00800001 - Reserved for future use

### nExponent

- **byVal nExponent AS INTEGER**
  - This value must be in the range -128 to 127 or the function fails.

### nIndex

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

### Interactive Command

- **No**

### Example

```plaintext
//Example :: BleEncodeFloat.sb
DIM rc
DIM attr$ : attr$=""

//write 1234567 x 10^-54 as FLOAT to index 2
PRINT BleEncodeFLOAT(attr$,123456,-54,0)

//write 1234567 x 10^1000 as FLOAT to index 2 and it will fail
//because the exponent is too large, it has to be < 127
IF BleEncodeFLOAT(attr$,1234567,1000,2)!=0 THEN
  PRINT "\nFailed to encode to FLOAT"
ENDIF

//write 10000000 x 10^0 as FLOAT to index 2 and it will fail
//because the mantissa is too large, it has to be < 8388600
IF BleEncodeFLOAT(attr$,10000000,0,2)!=0 THEN
  PRINT "\nFailed to encode to FLOAT"
ENDIF
```

### Expected Output:

```
0
Failed to encode to FLOAT
Failed to encode to FLOAT
```

BLEENCODEFLOAT is an extension function.

### BleEncodeSFLOATEX
FUNCTION
This function overwrites two bytes in a string at a specified offset as short 16-bit float value. If the string is not long enough, it is extended with the extended block uninitialized. Then the bytes are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**BLEENCODESFLOATEx**(attr$, nData, nIndex)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| attr$ | byRef attr$ AS STRING
This argument is the string that is written to an attribute |
| nData | byVal nData AS INTEGER
The 32 bit value is converted into a 2-byte IEEE-11073 16-bit SFLOAT consisting of a 12-bit signed mantissa and a 4-bit signed exponent. This means a signed 32-bit value always fits in such a FLOAT entity, but there is a loss in significance to 12 from 32. |
| nIndex | byVal nIndex AS INTEGER
This is the zero-based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails. |
| Interactive Command | No |

```vbnet
//Example :: BleEncodeSFloatEx.sb
DIM rc, mantissa, exp
DIM attr$ : attr$=""

//write 2,147,483,647 as SFLOAT to index 0
rc=BleEncodeSFloatEx(attr$,2147483647,0)
rc=BleDecodeSFloat(attr$,mantissa,exp,0);
PRINT "The number stored is ";mantissa," x 10^";exp
```

Expected Output:

```
The number stored is 214 x 10^7
```

BLEENCODESFLOAT is an extension function.

**BleEncodeSFLOAT**

FUNCTION
This function overwrites two bytes in a string at a specified offset as short 16-bit float value. If the string is not long enough, it is extended with the new block uninitialized. Then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**BLEENCODESFLOAT**(attr$, nMatissa, nExponent, nIndex)

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

Embedded Wireless Solutions Support Center:  195
http://ews-support.lairdtech.com
www.lairdtech.com/wireless
Americas: +1-800-492-2320
Europe: +44-1628-858-940
Hong Kong: +852 2923 0610
Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>attr$</code></td>
<td>byRef <code>attr$</code> AS STRING&lt;br&gt;This argument is the string that is written to an attribute</td>
</tr>
<tr>
<td><code>nMantissa</code></td>
<td>byVal <code>nMantissa</code> AS INTEGER&lt;br&gt;This must be in the range -2046 to +2046 or the function fails. The data is written in little endian so the least significant byte is at the lower memory address. <strong>Note:</strong> The range is not +/- 2048 because after encoding, the following 2-byte values have special meaning:</td>
</tr>
<tr>
<td><code>0x007Ff</code></td>
<td>NaN (Not a Number)</td>
</tr>
<tr>
<td><code>0x00800</code></td>
<td>NRes (Not at this resolution)</td>
</tr>
<tr>
<td><code>0x007FE</code></td>
<td>+ INFINITY</td>
</tr>
<tr>
<td><code>0x00802</code></td>
<td>- INFINITY</td>
</tr>
<tr>
<td><code>0x00801</code></td>
<td>Reserved for future use</td>
</tr>
<tr>
<td><code>nExponent</code></td>
<td>byVal <code>nExponent</code> AS INTEGER&lt;br&gt;This value must be in the range -8 to 7 or the function fails.</td>
</tr>
<tr>
<td><code>nIndex</code></td>
<td>byVal <code>nIndex</code> AS INTEGER&lt;br&gt;This is the zero based index into the string <code>attr$</code> where the new fragment of data is written. If the string <code>attr$</code> is not long enough to accommodate the index plus the length of the fragment, it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

Expected Output:

Success <br>Failed to encode to SFLOAT <br>Failed to encode to SFLOAT

**BLEENCODESFLOAT** is an extension function.

**BleEncodeTIMESTAMP**

FUNCTION
This function overwrites a 7-byte string into the string at a specified offset. If the string is not long enough, it is extended with the new extended block uninitialized and then the byte specified is overwritten.

The 7-byte string consists of a byte each for century, year, month, day, hour, minute and second. If (year * month) is zero, it is taken as “not noted” year and all the other fields are set zero (not noted).

For example, 5 May 2013 10:31:24 is represented as \14\0D\05\05\0A\1F\18.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

Note: When the attr$ string variable is updated, the two byte year field is convert into a 16-bit integer. Hence \14\0D gets converted to \DD\07

**BLEENCODETIMESTAMP (attr$, timestamp$, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td><strong>attr$</strong></td>
<td>byRef attr$ AS STRING</td>
</tr>
<tr>
<td></td>
<td>This argument is the string that is written to an attribute.</td>
</tr>
<tr>
<td><strong>timestamp$</strong></td>
<td>byRef timestamp$ AS STRING</td>
</tr>
<tr>
<td></td>
<td>This is a 7-byte string as described above. For example 5 May 2013 10:31:24 is entered \14\0D\05\05\0A\1F\18.</td>
</tr>
<tr>
<td><strong>nIndex</strong></td>
<td>byVal nIndex AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>This is the zero based index into the string attr$ where the new fragment of data is written.</td>
</tr>
<tr>
<td></td>
<td>If the string attr$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.</td>
</tr>
</tbody>
</table>

Interactive Command | No

```plaintext
//Example :: BleEncodeTimestamp.sb
DIM rc, ts$
DIM attr$ : attr$=""

//write the timestamp <5 May 2013 10:31:24>
ts$="\14\0D\05\05\0A\1F\18"
PRINT BleEncodeTimestamp(attr$,ts$,0)
```

Expected Output:

0

BLEENCODETIMESTAMP is an extension function.

**BleEncodeSTRING**

**FUNCTION**
This function overwrites a substring at a specified offset with data from another substring of a string. If the destination string is not long enough, it is extended with the new block uninitialized. Then the byte is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**BleEncodeSTRING (attr$, nIndex1 str$, nIndex2, nLen)**

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

Arguments:

- **attr$**
  - byRef attr$ AS STRING
  - This argument is the string is written to an attribute

- **nIndex1**
  - byVal nIndex1 AS INTEGER
  - This is the zero based index into the string attr$ where the new fragment of data is written
  - If the string attr$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

- **str$**
  - byRef str$ AS STRING
  - This contains the source data which is qualified by the nIndex2 and nLen arguments that follow.

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

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

**Interactive Command**

No

```vbnet
//Example :: BleEncodeString.sb
DIM rc, attr$, ts$: ts$="Hello World"
//write "Wor" from "Hello World" to the attribute at index 2
rc=BleEncodeString(attr$,2,ts$,6,3)
PRINT attr$
```

Expected Output:

```
\00\00Wor
```

BLEENCODESTRING is an extension function.

**BleEncodeBITS**

**FUNCTION**

This function overwrites some bits of a string at a specified bit offset with data from an integer which is treated as a bit array of length 32. If the destination string is not long enough, it is extended with the new extended block uninitialized. Then the bits specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512; hence the (nDstIdx + nBitLen) cannot be greater than the maximum attribute length times eight.

**BleEncodeBITS (attr$, nDstIdx, srcBitArr, nSrcIdx, nBitLen)**
Returns

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

Arguments:

<table>
<thead>
<tr>
<th>attr$</th>
<th>byRef attr$ AS STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is the string written to an attribute. It is treated as a bit array.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nDstIdx</th>
<th>byVal nDstIdx AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is the zero based bit index into the string attr$, treated as a bit array, where the new fragment of data bits is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>srcBitArr</th>
<th>byVal srcBitArr AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This contains the source data bits which is qualified by the nSrcIdx and nBitLen arguments that follow.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nSrcIdx</th>
<th>byVal nSrcIdx AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is the zero-based bit index into the bit array contained in srcBitArr from where the data bits is copied. No data is copied if this index is negative or greater than 32.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nBitLen</th>
<th>byVal nBitLen AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This species the number of bits from offset nSrcIdx to be copied into the destination bit array represented by the string attr$. It is clipped to the number of bits left to copy after the index nSrcIdx.</td>
</tr>
</tbody>
</table>

Interactive Command

No

//Example :: BleEncodeBits.sb
DIM attr$, rc, bA: bA=b'1110100001111
rc=BleEncodeBits(attr$,20,bA,7,5): PRINT attr$ //copy 5 bits from index 7 to attr$

Expected Output:

\00\00\A0\01

BLEENCODEBITS is an extension function.

**Attribute Decoding Functions**

Data in a characteristic is stored in a value attribute, a byte array. Multibyte characteristic descriptors content is stored similarly. Those bytes are manipulated in smartBASIC applications using STRING variables.

Attribute data is stored in little endian format.

This section describes decoding functions that allow attribute strings to be read from smaller byte wise subfields more efficiently than the generic STRXXXX functions that are made available in smartBASIC.

**Note:** CCCD and SCCD descriptors are special cases as they are defined as having two bytes which are treated as 16-bit integers mapped to INTEGER variables in smartBASIC.

**BleDecodeS8**

**FUNCTION**

This function reads a single byte in a string at a specified offset into a 32-bit integer variable with sign extension. If the offset points beyond the end of the string, then this function fails and returns zero.
BLEDECODES8 (attr$, nData, nIndex)

**Returns**
INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

**Arguments:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr$</td>
<td>byRef attr$ AS STRING</td>
<td>This references the attribute string from which the function reads.</td>
</tr>
<tr>
<td>nData</td>
<td>byRef nData AS INTEGER</td>
<td>This references an integer to be updated with the 8-bit data from attr$, after sign extension.</td>
</tr>
<tr>
<td>nIndex</td>
<td>byVal nIndex AS INTEGER</td>
<td>This is the zero based index into the string attr$ from which the data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.</td>
</tr>
</tbody>
</table>

**Interactive Command**
No

```plaintext
//Example :: BleDecodeS8.sb
DIM chrHandle, v1, svcHandle, rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

//create random service just for this example
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

//create char and commit as part of service commited above
rc=BleCharNew(0x07, BleHandleUuid16(0x2A1C), mdVal, 0, 0)
rc=BleCharCommit(svcHandle, attr$, chrHandle)

rc=BleServiceCommit(svcHandle)
rc=BleCharValueRead(chrHandle, attr$)

//read signed byte from index 2
rc=BleDecodeS8(attr$, v1, 2)
PRINT "\ndata in Hex = 0x"; v1
PRINT "\ndata in Decimal = "; v1;"n"

//read signed byte from index 6 - two's complement of -122
rc=BleDecodeS8(attr$, v1, 6)
PRINT "\ndata in Hex = 0x"; v1
PRINT "\ndata in Decimal = "; v1;"n"
```

**Expected Output:**

data in Hex = 0x00000002
data in Decimal = 2

data in Hex = 0xFFFFFFFF86
data in Decimal = -122

BLEDECODES8 is an extension function.
BleDecodeU8

FUNCTION

This function reads a single byte in a string at a specified offset into a 32-bit integer variable without sign extension. If the offset points beyond the end of the string, this function fails.

**BLEDECODEU8 (attr$, nData, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| attr$ | byRef attr$ AS STRING  
This references the attribute string from which the function reads. |
| nData | byRef nData AS INTEGER  
This references an integer to be updated with the 8-bit data from attr$, without sign extension. |
| nIndex | byVal nIndex AS INTEGER  
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails. |

Interactive Command

No

//Example :: BleDecodeU8.sb

```
DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\05\06\07\08\09"
DIM uuid : uuid = 0x1853
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
rc=BleServiceCommit(svcHandle)
rc=BleCharValueRead(chrHandle,attr$)

//read unsigned byte from index 2
rc=BleDecodeU8(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H\v1
PRINT "\ndata in Decimal = "; v1;"n"

//read unsigned byte from index 6
rc=BleDecodeU8(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H\v1
PRINT "\ndata in Decimal = "; v1;"n"
```

Expected Output:

```
data in Hex = 0x00000002
data in Decimal = 2
data in Hex = 0x00000086
```
BLEDECODEU8 is an extension function.

**BleDecodeS16**

**FUNCTION**

This function reads two bytes in a string at a specified offset into a 32-bit integer variable with sign extension. If the offset points beyond the end of the string then this function fails.

**BLEDECODES16**(attr$, nData, nIndex)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td><strong>attr$</strong></td>
<td>byRef attr$ AS STRING</td>
</tr>
<tr>
<td><strong>nData</strong></td>
<td>byRef nData AS INTEGER</td>
</tr>
<tr>
<td><strong>nIndex</strong></td>
<td>byVal nIndex AS INTEGER</td>
</tr>
<tr>
<td><strong>Interactive Command</strong></td>
<td>No</td>
</tr>
</tbody>
</table>

//Example :: BleDecodeS16.sb

```basic
DIM chrHandle, v1, svcHandle, rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\05\06\07\08\09"
DIM uuid : uuid = 0x1853
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
rc=BleServiceCommit(svcHandle)
rc=BleCharValueRead(chrHandle,attr$)
//read 2 signed bytes from index 2
rc=BleDecodeS16(attr$ˌv1ˌ2)
PRINT "\ndata in Hex = 0x"; INTEGER.Hˌv1
PRINT "\ndata in Decimal = "; v1;"n"
//read 2 signed bytes from index 6
rc=BleDecodeS16(attr$ˌv1ˌ6)
PRINT "\ndata in Hex = 0x"; INTEGER.Hˌv1
PRINT "\ndata in Decimal = "; v1;"n"
```

*data in Decimal = 134*
Expected Output:

<table>
<thead>
<tr>
<th>data in Hex</th>
<th>data in Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000302</td>
<td>770</td>
</tr>
<tr>
<td>0xFFFF8786</td>
<td>-30842</td>
</tr>
</tbody>
</table>

**BLEDECODEU16** is an extension function.

**BleDecodeU16**

This function reads two bytes from a string at a specified offset into a 32-bit integer variable without sign extension. If the offset points beyond the end of the string, then this function fails.

**BLEDECODEU16 (attr$, nData, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>attr$</strong></td>
<td>byRef attr$ AS STRING</td>
</tr>
<tr>
<td></td>
<td>This references the attribute string from which the function reads.</td>
</tr>
<tr>
<td><strong>nData</strong></td>
<td>byRef nData AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>This references an integer to be updated with the 2-byte data from attr$, without sign extension.</td>
</tr>
<tr>
<td><strong>nIndex</strong></td>
<td>byVal nIndex AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.</td>
</tr>
</tbody>
</table>

**Interactive Command**

No

//Example :: BleDecodeU16.sb

```basic
DIM chrHandle, v1, svcHandle, rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\05\06\07\08\09"
DIM uuid : uuid = 0x1853
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
rc=BleServiceCommit(svcHandle)
rc=BleCharValueRead(chrHandle,attr$)

//read 2 unsigned bytes from index 2
rc=BleDecodeU16(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;

//read 2 unsigned bytes from index 6
rc=BleDecodeU16(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
```
Expected Output:

```
data in Hex = 0x00000302
data in Decimal = 770
```
```
data in Hex = 0x000008796
data in Decimal = 34694
```

BLEDECODEU16 is an extension function.

**BleDecodeS24**

**FUNCTION**

This function reads three bytes in a string at a specified offset into a 32-bit integer variable with sign extension. If the offset points beyond the end of the string, this function fails.

**BLEDECODES24 (attr$, nData, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| attr$            | byRef attr$ AS STRING
This references the attribute string from which the function reads. |
| nData            | byRef nData AS INTEGER
This references an integer to be updated with the 3-byte data from attr$, with sign extension. |
| nIndex           | byVal nIndex AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails. |
| Interactive Command | No                                                                 |

//Example :: BleDecodeS24.sb

```
DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$: attr$="\00\01\02\03\04\05\06\07\08\09"
DIM uuid : uuid = 0x1853
rc=BleServiceNew(1, BleHandlerUuid16(uuid), svcHandle)
rc=BleCharNew(0x07,BleHandlerUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
rc=BleServiceCommit(svcHandle)
rc=BleCharValueRead(chrHandle,attr$)
//read 3 signed bytes from index 2
rc=BleDecodeS24(attr$,v1,2)
```
PRINT "\ndata in Hex = 0x"; INTEGER.H\'v1
PRINT "\ndata in Decimal = "; v1;\n"

//read 3 signed bytes from index 6
rc=BleDecodeS24(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H\'v1
PRINT "\ndata in Decimal = "; v1;\n"

Expected Output:

- data in Hex = 0x00040302
- data in Decimal = 262914
- data in Hex = 0xFF888786
- data in Decimal = -7829626

BLEDECODES24 is an extension function.

**BleDecodeU24**

**FUNCTION**

This function reads three bytes from a string at a specified offset into a 32-bit integer variable **without** sign extension. If the offset points beyond the end of the string, then this function fails.

**BLEDECODEU24 (attr$, nData, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments:</strong></td>
<td></td>
</tr>
</tbody>
</table>
| attr$  | byRef attr$ AS STRING  
This references the attribute string from which the function reads. |
| nData  | byRef nData AS INTEGER  
This references an integer to be updated with the 3-byte data from attr$, without sign extension. |
| nIndex | byVal nIndex AS INTEGER  
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails. |

**Interactive Command**

No

//Example :: BleDecodeU24.sb

```
DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="00\01\02\03\04\05\06\07\08\09"
DIM uuid : uuid = 0x1853
rc=BleServiceNew(1, BleHandleUuid16 (uuid), svcHandle)
rc=BleCharNew(0x07, BleHandleUuid16 (0x2A1C), mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
```
WB45 smartBASIC Extensions
User Guide

rc=BleServiceCommit(svcHandle)
rc=BleCharValueRead(chrHandle,attr$)

//read 3 unsigned bytes from index 2
rc=BleDecodeU24(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H\v1
PRINT "\ndata in Decimal = "; v1;\n"

//read 3 unsigned bytes from index 6
rc=BleDecodeU24(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H\v1
PRINT "\ndata in Decimal = "; v1;\n"

Expected Output:

\ndata in Hex = 0x00040302
\ndata in Decimal = 262914
\ndata in Hex = 0x00888786
\ndata in Decimal = 8947590

BLEDECODEU24 is an extension function.

**BleDecode32**

**FUNCTION**

This function reads four bytes in a string at a specified offset into a 32-bit integer variable. If the offset points beyond the end of the string, this function fails.

** BLEDECODE32 (attr$, nData, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments:</strong></td>
<td></td>
</tr>
<tr>
<td>attr$</td>
<td>byRef attr$ AS STRING This references the attribute string from which the function reads.</td>
</tr>
<tr>
<td>nData</td>
<td>byRef nData AS INTEGER This references an integer to be updated with the 3-byte data from attr$, after sign extension.</td>
</tr>
<tr>
<td>nIndex</td>
<td>byVal nIndex AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

//Example :: BleDecode32.sb
DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\001\01\02\03\04\05\06\07\08\09"
DIM uuid : uuid = 0x1853
BLEDECODE32 is a extension function.

**BleDecodeFLOAT**

**FUNCTION**

This function reads four bytes in a string at a specified offset into a couple of 32-bit integer variables. The decoding results in two variables, the 24-bit signed mantissa and the 8-bit signed exponent. If the offset points beyond the end of the string, this function fails.

**BLEDECODEFLOAT**(attr$, nMantissa, nExponent, nIndex)

**Returns**: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

**Arguments:**

<table>
<thead>
<tr>
<th>attr$</th>
<th>byRef attr$ AS STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This references the attribute string from which the function reads.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nMantissa</th>
<th>byRef nMantissa AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is updated with the 24 bit mantissa from the 4-byte object.</td>
</tr>
<tr>
<td></td>
<td>If nExponent is 0, you must check for the following special values:</td>
</tr>
<tr>
<td>0x007FFFFF</td>
<td>NaN (Not a Number)</td>
</tr>
<tr>
<td>0x00800000</td>
<td>NRes (Not at this resolution)</td>
</tr>
<tr>
<td>0x007FFFFE</td>
<td>+ INFINITY</td>
</tr>
<tr>
<td>0x00800002</td>
<td>- INFINITY</td>
</tr>
<tr>
<td>0x00800001</td>
<td>Reserved for future use</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nExponent</th>
<th>byRef nExponent AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is updated with the 8-bit mantissa. If it is zero, check nMantissa for special cases as</td>
</tr>
</tbody>
</table>
WB45 smartBASIC Extensions
User Guide

nIndex

byVal nIndex AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command

No

//Example :: BledDecodeFloat.sb
DIM chrHandle, v1, svcHandle, rc, mantissa, exp
DIM mdVal : mdVal = BleAttrMetadata (1, 1, 50, 0, rc)
DIM attr$ : attr$ = "\00\01\02\03\04\05\06\07\08\09"
DIM uuid : uuid = 0x1853
rc = BleServiceNew (1, BleHandleUuid16 (uuid), svcHandle)
rc = BleCharNew (0x07, BleHandleUuid16 (0x2A1C), mdVal, 0, 0)
rc = BleCharCommit (svcHandle, attr$, chrHandle)
rc = BleServiceCommit (svcHandle)
rc = BleCharValueRead (chrHandle, attr$)
//read 4 bytes FLOAT from index 2 in the string
rc = BledDecodeFloat (attr$, mantissa, exp, 2)
PRINT "\nThe number read is ";mantissa;" x 10^";exp

//read 4 bytes FLOAT from index 6 in the string
rc = BledDecodeFloat (attr$, mantissa, exp, 6)
PRINT "\nThe number read is ";mantissa;" x 10^";exp

Expected Output:
The number read is 262914*10^-123
The number read is -7829626*10^-119

BLEDECODEFLOAT is an extension function.

BleDecodeSFLOAT

FUNCTION
This function reads two bytes in a string at a specified offset into a couple of 32-bit integer variables. The decoding results in two variables, the 12-bit signed mantissa and the 4-bit signed exponent. If the offset points beyond the end of the string then this function fails.

BLEDECODEFLOAT (attr$, nMatissa, nExponent, nIndex)

Returns
INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$ byRef attr$ AS STRING
This references the attribute string from which the function reads.

nMantissa byRef nMantissa AS INTEGER
This is updated with the 12-bit mantissa from the two byte object.
If the nExponent is 0, you must check for the following special values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x007FFFFF</td>
<td>NaN (Not a Number)</td>
</tr>
<tr>
<td>0x00800000</td>
<td>NRRes (Not at this resolution)</td>
</tr>
<tr>
<td>0x007FFFFE</td>
<td>+ INFINITY</td>
</tr>
<tr>
<td>0x00800002</td>
<td>- INFINITY</td>
</tr>
<tr>
<td>0x00800001</td>
<td>Reserved for future use</td>
</tr>
</tbody>
</table>

**nExponent** byRef nExponent AS INTEGER
This is updated with the 4-bit mantissa. If it is zero, check the nMantissa for special cases as stated above.

**nIndex** byVal nIndex AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

---

```vbnet
//Example :: BleDecodeSFloat.sb

DIM chrHandle,v1,svcHandle,rc, mantissa, exp
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="00\01\02\03\04\05\06\07\08\09"
DIM uuid : uuid = 0x1853
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
rc=BleServiceCommit(svcHandle)
rc=BleCharValueRead(chrHandle,attr$)

//read 2 bytes FLOAT from index 2 in the string
rc=BleDecodeSFloat(attr$,mantissa,exp,2)
PRINT "\nThe number read is ";mantissa;" x 10^";exp

//read 2 bytes FLOAT from index 6 in the string
rc=BleDecodeSFloat(attr$,mantissa,exp,6)
PRINT "\nThe number read is ";mantissa;" x 10^";exp
```

**Expected Output:**

```
The number read is 770 x 10^0
The number read is 1926 x 10^-8
```

BLEDECODESFLOAT is an extension function.

**BleDecodeTIMESTAMP**

**FUNCTION**
This function reads seven bytes from string an offset into an attribute string. If the offset plus seven bytes points beyond the end of the string then this function fails.
The seven byte string consists of a byte each for century, year, month, day, hour, minute and second. If (year * month) is zero, it is taken as “not noted” year and all the other fields are set zero (not noted).

For example: 5 May 2013 10:31:24 is represented in the source as \DD\07\05\05\0A\1F\18 and the year is be translated into a century and year so that the destination string is \14\0D\05\05\0A\1F\18.

**BLEDECODETIMESTAMP (attr$, timestamp$, nIndex)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.</th>
</tr>
</thead>
</table>

**Arguments:**

- **attr$** byRef attr$ AS STRING
  - This references the attribute string from which the function reads.

- **timestamp$** byRef timestamp$ AS STRING
  - On exit this is an exact 7-byte string as described above.
  - For example: 5 May 2013 10:31:24 is stored as \14\0D\05\05\0A\1F\18

- **nIndex** byVal nIndex AS INTEGER
  - This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

**Interactive Command** No

```
// Example :: BleDecodeTimestamp.sb
DIM chrHandle, v1, svcHandle, rc, ts$
DIM mdVal : mdVal = BleAttrMetadata(1, 1, 50, 0, rc)
// 5th May 2013, 10:31:24
DIM attr$ : attr$ = "\00\01\02\DD\07\05\05\0A\1F\18"
DIM uuid : uuid = 0x1853
rc = BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)
rc = BleCharNew(0x07, BleHandleUuid16(0x2A1C), mdVal, 0, 0)
rc = BleCharCommit(svcHandle, attr$, chrHandle)
rc = BleServiceCommit(svcHandle)
rc = BleCharValueRead(chrHandle, attr$)
// read 7 byte timestamp from the index 3 in the string
rc = BleDecodeTimestamp(attr$, ts$, 3)
PRINT "\nTimestamp = "; StrHexize$(ts$)
```

**Expected Output:**

```
Timestamp = 140D05050A1F18
```

**BLEENCODETIMESTAMP** is an extension function.

**BleDecodeSTRING**

**FUNCTION**
This function reads a maximum number of bytes from an attribute string at a specified offset into a destination string. Because the output string can handle truncated bit blocks, this function does not fail.

**BLEDECODESTRING (attr$, nIndex, dst$, nMaxBytes)**

| **Returns** | INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string. |
| **Arguments:** | |
| attr$ | byRef attr$ AS STRING |
| nIndex | byVal nIndex AS INTEGER |
| dst$ | byRef dst$ AS STRING |
| nMaxBytes | byVal nMaxBytes AS INTEGER |

**Interactive Command** No

//Example :: BleDecodeString.sb

```vbnet
DIM chrHandle,v1,svcHandle,rc, ts$,decStr$
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//"ABCDEFGHIJ"
DIM attr$ : attr$="41\42\43\44\45\46\47\48\49\4A"
DIM uuid : uuid = 0x1853
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rcl=BleCharCommit(svcHandle,attr$,chrHandle)
rc=BleServiceCommit(svcHandle)
rc=BleCharValueRead(chrHandle,attr$
//read max 4 bytes from index 3 in the string
rc=BleDecodeSTRING(attr$,3,decStr$,4)
PRINT "\nd$=\";decStr$
//read max 20 bytes from index 3 in the string - will be truncated
rc=BleDecodeSTRING(attr$,3,decStr$,20)
PRINT "\nd$=\";decStr$
//read max 4 bytes from index 14 in the string - nothing at index 14
rc=BleDecodeSTRING(attr$,14,decStr$,4)
PRINT "\nd$=\"
```

**Expected Output:**

| d$=CDEF |
| d$=CDEFGHIJ |
| d$= |
BLEDECODESTRING is an extension function.

**BleDecodeBITS**

**FUNCTION**

This function reads bits from an attribute string at a specified offset (treated as a bit array) into a destination integer object (treated as a bit array of fixed size of 32). This implies a maximum of 32 bits can be read. Because the output bit array can handle truncated bit blocks, this function does not fail.

**BLEDECODEBITS (attr$, nSrcIdx, dstBitArr, nDstIdx,nMaxBits)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, the number of bits extracted from the attribute string. Can be less than the size expected if the nSrcIdx parameter is positioned towards the end of the source string or if nDstIdx will not allow more to be copied.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| **attr$** | byRef attr$ AS STRING  
This references the attribute string from which to read, treated as a bit array. Hence a string of 10 bytes is an array of 80 bits. |
| **nSrcIdx** | byVal nSrcIdx AS INTEGER  
This is the zero based bit index into the string attr$ from which data is read. For example, the third bit in the second byte is index number 10. |
| **dstBitArr** | byRef dstBitArr AS INTEGER  
This argument references an integer treated as an array of 32 bits into which data is copied. Only the written bits are modified. |
| **nDstIdx** | byVal nDstIdx AS INTEGER  
This is the zero based bit index into the bit array dstBitArr to where the data is written. |
| **nMaxBits** | byVal nMaxBits AS INTEGER  
This argument specifies the maximum number of bits to read from attr$. Due to the destination being an integer variable, it cannot be greater than 32. Negative values are treated as zero. |
| **Interactive Command** | No |

//Example :: BleDecodeBits.sb

```
DIM chrHandle, v1,srvHandle, rc, ts$, decStr$
DIM ba : ba = 0
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
DIM attr$ : attr$="41\42\43\44\45\46\47\48\49\4A"
DIM uuid : uuid = 0x1853
rc=BleServiceNew(1, BleHandleUuid16 (uuid), srvHandle)
rc=BleCharNew(0x07,BleHandleUuid16 (0x2A1C),mdVal,0,0)
rc=BleCharCommit(srvHandle, attr$, chrHandle)
rc=BleServiceCommit(srvHandle)
rc=BleCharValueRead(chrHandle,attr$)
```
Pairing/Bonding Functions

This section describes all functions related to the pairing and bonding manager which manages trusted devices. The database stores information such as the address of the trusted device along with the security keys. At the time of writing this guide, a maximum of four devices can be stored in the database.

The command AT I 2012 or at runtime SYSINFO(2012) returns the maximum number of devices that can be saved in the database.

The type of information that can be stored for a trusted device is:

- The Bluetooth address of the trusted device.
- The eDIV and eRAND for the long term key.
- A 16-byte Long Term Key (LTK).
- The size of the LTK.
- A flag to indicate if the LTK is authenticated – Man-In-The-Middle (MITM) protection.
- A 16-byte Identity Resolving Key (IRK).
- A 16-byte Connection Signature Resolving Key (CSRK)

BleBondingStats

FUNCTION

This function is used to get the BLE bonding manager database statistics.

BEBONDINGSTATS (nRolling, nPersistent)

<table>
<thead>
<tr>
<th>Returns</th>
<th>The total capacity of the database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nRolling</td>
<td>byREF nRolling AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>On return, this integer contains the total number of bonds in the rolling database.</td>
</tr>
<tr>
<td>nPersistent</td>
<td>byREF nPersistent AS INTEGER</td>
</tr>
<tr>
<td></td>
<td>On return, this integer contains the total number of bonds in the persistent database.</td>
</tr>
</tbody>
</table>
Interactive Command | No
---|---

```plaintext
dim rc, nRoll, nPers
print "\n:Bonding Manager Database Statistics:"
print "\nCapacity: ", "", BleBondingStats(nRoll, nPers)
print "\nRolling: ", "", nRoll
print "\nPersistent: ", nPers
```

**Expected Output:**

```
:Bonding Manager Database Statistics:
Capacity: 16
Rolling: 2
Persistent: 0
```

BLEBONDINGSTATS is a built-in function.

**BleBondingIsTrusted**

**FUNCTION**

This function is used to check if a device identified by the address is a trusted device which means it exists in the bonding database.

**BLEBONDINGISTRUSTED (addr$, fAsCentral, keyInfo, rollingAge, rollingCount)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER: Is 0 if not trusted, otherwise it is the length of the long term key (LTK)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td><strong>addr$</strong></td>
<td>byRef addr$ AS STRING This is the address of the device for which the bonding information is to be checked.</td>
</tr>
<tr>
<td><strong>fAsCentral</strong></td>
<td>Set to 0 if the device is to be trusted as a peripheral and non-zero if to be trusted as central.</td>
</tr>
<tr>
<td><strong>keyInfo</strong></td>
<td>This is a bit mask with bit meanings as follows: This specifies the write rights and shall have one of the following values:</td>
</tr>
<tr>
<td></td>
<td>Bit 0 Set if MITM is authenticated</td>
</tr>
<tr>
<td></td>
<td>Bit 1 Set if it is a rolling bond and can be automatically deleted if the database is full and a new bonding occurs</td>
</tr>
<tr>
<td></td>
<td>Bit 2 Set if an IRK (identity resolving key) exists</td>
</tr>
<tr>
<td></td>
<td>Bit 3 Set if a CSRK (connection signing resolving key) exists</td>
</tr>
<tr>
<td></td>
<td>Bit 4 Set if LTK as slave exists</td>
</tr>
<tr>
<td></td>
<td>Bit 5 Set if LTK as master exists</td>
</tr>
<tr>
<td><strong>rollingAge</strong></td>
<td>If the value is &lt;= 0 then this is not a rolling device 1 implies it is the newest bond 2 implies it is the second newest bond etc</td>
</tr>
<tr>
<td><strong>rollingCount</strong></td>
<td>On exit this will contain the total number of rolling bonds. Which give a a sense of how old this device is compared to other bonds in the rolling group.</td>
</tr>
</tbody>
</table>

Interactive Command | No
//Example
DIM rc, addr$
addr$="00\00\16\A4\12\34\56"
rc = BleBondingPersistKey(addr$)

BLEBONDINGISTRUSTED is an extension function.

BleBondingPersistKey

FUNCTION
This function is used to make a bonding link key persistent. Its entry is moved from the rolling database to the persistent database so that it is never automatically overwritten.

BLEBONDINGPERSISTKEY (bdAddr$)

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

Arguments:

bdAddr$ byREF bdAddr$ AS STRING
Bluetooth address in big endian. Must be exactly seven bytes long

Interactive Command
No

'//Loop through the bonding manager. Make all entries persistent
for i=0 to BleBondingStats(j,k)
rc=BleBondMngrGetInfo(i,adr$,inf)
if rc==0 then
   rc=BleBondingPersistKey(adr$)
   print "\n(\"i:\") : \";StrHexize$(adr$);" Now Persistent"
endif
next

Expected Output:

(0) : 01F63627A60BEA Now Persistent
(1) : 01D8CFCF14498D Now Persistent

BLEBONDINGPERSISTKEY is a built-in function.

BleBondingEraseKey

FUNCTION
This function is used to erase a link key from the database for the address specified.

BLEBONDINGERASEKEY (bdAddr$)

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

Arguments:

bdAddr$ byREF bdAddr$ AS STRING
Bluetooth address in big endian. Must be exactly seven bytes long
**Interactive Command**

| Interactive Command | No |

```basic
dim rc, i, adr$, inf

//delete link key at index 0
rc=BleBondMngrGetInfo(0,adr$,inf)    //get the BT address
rc=BleBondErasingEraseKey(adr$
if rc==0 then
   print "\nLink key for device ";StrHexize$(adr$);": erased"
else
   print "\nError erasing link key ";integer.h \rc
endif
```

**Expected Output:**

```
Link key for device 01FA84D748D903 erased
```

**BLEBONDINGERASEKEY** is a built-in function.

**BleBondingEraseAll**

**FUNCTION**

This function is used to erase all bondings in the database

```basic
BLEBONDINGERASEALL ()
```

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```basic
dim rc

//erase all bondings in database
rc=BleBondingEraseAll()
if rc==0 then
   print "\nBonding database cleared"
endif
```

**Expected Output:**

```
Bonding database cleared
```

**BLEBONDINGERASEALL** is a built-in function.

**BleBondMngrGetInfo**

**FUNCTION**

This function retrieves the Bluetooth address and other information from the trusted device database via an index.
Note: Do not rely on a device in the database mapping to a static index. New bondings change the position in the database.

**BLEBONDMMGRGETINFO (nIndex, addr$, nExtraInfo)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nIndex</td>
<td>byVal nIndex AS INTEGER This is an index in the range 0 to 1, less than the value returned by SYSINFO(2012).</td>
</tr>
<tr>
<td>addr$</td>
<td>byRef addr$ AS STRING On exit, if nIndex points to a valid entry in the database, this variable contains a Bluetooth address exactly seven bytes long. The first byte identifies public or private random address. The next six bytes are the address.</td>
</tr>
<tr>
<td>nExtraInfo</td>
<td>byRef nExtraInfo AS INTEGER On exit, if nIndex points to a valid entry in the database, this variable contains a composite integer value where the lower 16 bits are for internal use and should be treated as opaque data. Bit 16 is set if the IRK (Identity Resolving Key) exists for the trusted device and bit 17 is set if the CSRK (Connection Signing Resolving Key) exists for the trusted device.</td>
</tr>
<tr>
<td>Interactive Command</td>
<td>No</td>
</tr>
</tbody>
</table>

```c
//Example :: BleBondMngrGetInfo.sb
#define BLE_INV_INDEX 24619
DIM rc, addr$, exInfo
rc = BleBondMngrGetInfo(0,addr$,exInfo) //Extract info of device at index 1
IF rc==0 THEN
    PRINT "\nBluetooth address: ";addr$
    PRINT "\nInfo: ";exInfo
ELSEIF rc==BLE_INV_INDEX THEN
    PRINT "\nInvalid index"
ENDIF
```

**Expected Output when valid entry present in database:**

```
Bluetooth address: \00\BC\B1\F3x3\AB
Info: 97457
```

**Expected Output with invalid index:**

```
Invalid index
```

BLEBONDMMGRGETINFO is an extension function.
7. SOCKET EXTENSIONS BUILT-IN Routines

Socket Functions

Events and Messages

**EVSOCKETCONN**
This event is thrown to indicate that a new connection has been made to the socket. It is thrown for both the initiating and the listening socket. The handler for this event contains **nHandle** which is the handle of the connection created. This handle should be used when disconnecting.

**EVSOCKETDISCON**
This event is thrown to inform the smartBASIC application that a connection to the socket has been dropped. The handler of the event contains **nHandle**, which is the handle of the connection that has been dropped.

**EVSOCKET_DATA_RECEIVED**
This event is thrown to inform the smartBASIC app that data is now available to be read at the socket. The user must then use the socketread function to read the data and the handle on which the data was received.

SocketOpenSock

FUNCTION
This function is used to create a new socket to listen for incoming UNIX/IPV4 connections.

**SocketOpenSock (path$, nPort, nFamily, nType, nHandle)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| **path$** | byREF **path$** AS STRING  
For UNIX domain sockets, this is the local path to the socket. For IPv4 sockets this is irrelevant and can be passed as an empty string. |
| **nPort** | byVAL **nPort** AS INTEGER  
The port number to use for the socket. |
| **nFamily** | byVAL **nFamily** AS INTEGER  
The family/domain of the opened socket.  
   0 Unix domain socket  
   1 IPv4 domain socket  |
| **nType** | byVAL **nType** AS INTEGER  
The type of the socket  
   0 Stream  
   1 Datagram  |
| **nHandle** | byVAL **nHandle** AS INTEGER  
On return, this integer contains the handle of the created socket. |

Interactive Command
No

**Note:** Datagram sockets are not supported in this release of smartBASIC and will be added in future releases.
dim rc, path$, nHandle

#define SOCKET_FAMILY_UNIX      0
#define SOCKET_FAMILY_IPV4      1
#define SOCKET_TYPE_STREAM      0

path$= "/tmp/MyTempSocket"

// Open a UNIX domain socket
rc = SocketOpenSock(path$, 0, SOCKET_FAMILY_UNIX, SOCKET_TYPE_STREAM, nHandle)
if rc == 0 THEN
    print "UNIX socket successfully opened with handle = ";nHandle;"\n"
else
    print "Failed to open UNIX socket\n"
endif

path$= ""

// Open an IPV4 domain socket
rc = SocketOpenSock(path$, 0, SOCKET_FAMILY_IPV4, SOCKET_TYPE_STREAM, nHandle)
if rc == 0 THEN
    print "IPV4 socket successfully opened with handle = ";nHandle;"\n"
else
    print "Failed to open IPV4 socket\n"
endif

Expected output:

UNIX socket successfully opened with handle = 5
IPV4 socket successfully opened with handle = 6

SOCKETOPENSOCK is an extension function.

**SocketCloseSock**

**FUNCTION**

This function will close the listening socket given by the handle that was generated when the socket was opened.

**SocketCloseSock (nHandle)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
</table>

| Arguments: |
| nHandle | **byVAL nHandle** AS INTEGER |
|         | The handle of the socket to be closed. |

<table>
<thead>
<tr>
<th>Interactive Command</th>
<th>No</th>
</tr>
</thead>
</table>

dim rc, path$, nHandle

#define SOCKET_FAMILY_UNIX      0
#define SOCKET_TYPE_STREAM      1
path$ = "/tmp/MyTempSocket"

// Open the socket
rc = SocketOpenSock(path$, 0, SOCKET_FAMILY_UNIX, SOCKET_TYPE_STREAM, nHandle)

// Now close it immediately
rc = SocketCloseSock(nHandle)
if rc == 0 then
    print "Successfully closed socket\n"
else
    print "Failed to close socket\n"
endif

Expected output:
Successfully closed socket

SOCKETCLOSESOCK is an extension function.

**SocketConnect**

**FUNCTION**

This function is used to create a new socket connection to a UNIX or an IPv4 domain socket.

**SocketConnect (path$, nPort, nFamily)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
</table>

**Arguments:**

<table>
<thead>
<tr>
<th>path$</th>
<th>byREF path $AS STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For UNIX domain sockets, this is the local path to the socket. For IPv4 sockets this is the IP address of the device hosting the socket.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nPort</th>
<th>byVAL nPort AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The port number of the listening socket.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nFamily</th>
<th>byVAL nFamily AS INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The family/domain of the opened socket.</td>
</tr>
<tr>
<td>0</td>
<td>Unix domain socket</td>
</tr>
<tr>
<td>1</td>
<td>IPv4 domain socket</td>
</tr>
</tbody>
</table>

**Interactive Command** No

**Note:** Datagram sockets are not supported in this release of smartBASIC and will be added in future releases.

dim rc, unix_path$, ip4_path$, nHandle

#define SOCKET_FAMILY_UNIX 0
#define SOCKET_FAMILY_IPV4 1
#define SOCKET_TYPE_STREAM 0
unix_path$ = "/tmp/MyTempSocket"
ip4_path$ = ""

// Open a UNIX domain socket
rc = SocketOpenSock(unix_path$, 0, SOCKET_FAMILY_UNIX, SOCKET_TYPE_STREAM, nHandle)
if rc == 0 THEN
  print "UNIX socket successfully opened with handle = ";nHandle;
else
  print "Failed to open UNIX socket"
endif

// Open an IPV4 domain socket
rc = SocketOpenSock(ip4_path$, 3000, SOCKET_FAMILY_IPV4, SOCKET_TYPE_STREAM, nHandle)
if rc == 0 THEN
  print "IPV4 socket successfully opened with handle = ";nHandle;
else
  print "Failed to open IPV4 socket"
endif

// Connect to the UNIX socket
rc = SocketConnect(unix_path$, 0, SOCKET_FAMILY_UNIX)
if rc == 0 THEN
  print "UNIX socket connection initiated"
else
  print "Failed to initiate UNIX socket connection"
endif

// Now connect to the local IPv4 socket
ip4_path$ = "0.0.0.0"
rc = SocketConnect(ip4_path$, 3000, SOCKET_FAMILY_IPV4)
if rc == 0 THEN
  print "IPV4 socket connection initiated"
else
  print "Failed to initiate IPV4 socket connection"
endif

'//==============================================================================
'// Called when a socket connection is created
'//==============================================================================
function HandlerSockConn(nHandle)
  print "\n--- Socket Connected : ";nHandle;
endfunc

'//*****************************************************************************/
'// Equivalent to main() in C
'//*****************************************************************************/
OnEvent EvSocketCONN call HandlerSockConn

'="/tmp/MyTempSocket"
"
**Expected output:**

```plaintext
UNIX Socket successfully opened with handle = 5
IPV4 Socket successfully opened with handle = 6
UNIX socket connection initiated
IPV4 socket connection initiated

--- Socket Connected : 8
--- Socket Connected : 9
```

SOCKETCONNECT is an extension function.

**SocketDisconnect**

**FUNCTION**

This function will try and disconnect a socket connection given by the socket connection handle.

**SocketDisconnect (nHandle)**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>nHandle</td>
<td>byVAL nHandle AS INTEGER</td>
</tr>
<tr>
<td>Interactive Command</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
dim rc, unix_path$, ip4_path$, nHandle
#define SOCKET_FAMILY_UNIX      0
#define SOCKET_TYPE_STREAM      0
unix_path$= "/tmp/MyTempSocket"

// Open a UNIX domain socket
rc = SocketOpenSock(unix_path$, 0, SOCKET_FAMILY_UNIX, SOCKET_TYPE_STREAM, nHandle)
if rc == 0 THEN
  print "UNIX socket successfully opened with handle = ";nHandle;
else
  print "Failed to open UNIX socket\n"
endif

// Connect to the UNIX socket
rc = SocketConnect(unix_path$, 0, SOCKET_FAMILY_UNIX)
if rc == 0 THEN
  print "UNIX socket connection initiated\n"
else
  print "Failed to initiate UNIX socket connection\n"
endif
```

```plaintext
'//==============================================================================
'// Called when a socket connection is created
'//==============================================================================
function HandlerSockConn (nHandle)
```
print "\n--- Socket Connected : ";nHandle;"\n"

// Let's disconnect immediately after connecting
rc = SocketDisconnect(nHandle)
if rc == 0 THEN
    print "UNIX socket disconnection initiated\n"
else
    print "Failed to initiate UNIX socket disconnection\n"
endif
endfunc 1

'==============================================================================
'// Called upon a socket disconnection
'==============================================================================
function HandlerSockDiscon(nHandle)
    print "\n--- Socket Disconnected"
endfunc 1

'****************************************************************************
'// Equivalent to main() in C
'*****************************************************************************
OnEvent EvSocketCONN call HandlerSockConn
OnEvent EvSocketDISCON call HandlerSockDiscon

'****************************************************************************
'// Wait for a synchronous event.
'****************************************************************************
Waitevent

Expected output:

UNIX Socket successfully opened with handle = 5
UNIX socket connection initiated
IPV4 socket connection initiated

--- Socket Connected : 6
UNIX socket disconnection initiated

--- Socket Disconnected
--- Socket Disconnected

SOCKETDISCONNECT is an extension function.

**SocketWrite**

**FUNCTION**

This function is used to send data over a given socket connection.

**SocketWrite(nHandle, data$)**

<table>
<thead>
<tr>
<th><strong>Returns</strong></th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments</strong></td>
<td></td>
</tr>
</tbody>
</table>
### SOCKET_WRITE

**Handles**

<table>
<thead>
<tr>
<th><strong>nHandle</strong></th>
<th><strong>byVAL nPort AS INTEGER</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The handle of the socket connection to write to.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>data$</strong></th>
<th><strong>byREF data$ AS STRING</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This contains the data to be written.</td>
</tr>
</tbody>
</table>

**Interactive Command**

No

```vbnet
dim rc, path$, data$, nHandle
#define SOCKET_FAMILY_UNIX    0
#define SOCKET_TYPE_STREAM    0
path$ = "/tmp/MyTempSocket"
data$ = "Some data"

'======================================================================
'// Called after receiving a socket connection event
'======================================================================
function HandlerSockConn(nHandle)
    print "\n--- Socket Connected : ";nHandle;"\n"  // Now close it immediately
    rc = SocketWrite(nHandle, data$)
    if rc == 0 then
        print "Successfully sent some data over the socket\n"
    else
        print "Failed to send data over the socket\n"
    endif
endfunc 1

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

// Open a UNIX domain socket
rc = SocketOpenSock(path$, 0, SOCKET_FAMILY_UNIX, SOCKET_TYPE_STREAM, nHandle)

// Connect to the opened socket
rc = SocketConnect(path$, 0, SOCKET_FAMILY_UNIX)

OnEvent EvSocketCONN call HandlerSockConn
waitevent
```

**Expected output:**

```
--- Socket Connected : 6
Successfully sent some data over the socket
```

SOCKET_WRITE is an extension function.
SocketReadData

Read data from the oldest Socket data event. Since the event EVSOCKET_DATA_RECEIVED is invoked every time data is received, and data can be received from multiple sockets, this function should be called in the EVSOCKET_DATA_RECEIVED handler to process all waiting data.

FUNCTION

SocketReadData (data$, nHandle, nLength)

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
</tbody>
</table>
| data$   | byREF path $AS INTEGER  
The data received from the socket. |
| nHandle | byVAL nPort AS INTEGER  
On return, this will contain the socket handle from which the data has been read. |
| nLength | byVAL nPort AS INTEGER  
On return, this will contain the length of the data read. |
| Interactive Command | No |

'//****************************************************************************
 }// Definitions
'//****************************************************************************
#define SOCKET_FAMILY_UNIX 0
#define SOCKET_TYPE_STREAM 0

'//****************************************************************************
 }// Global Variable Declarations
'//****************************************************************************
dim rc, path$, data$, nHandle
path$= "/tmp/MyTempSocket"
data$= "This is some random data"

'//==============================================================================
 }// Called upon receiving data on the socket interface
'//==============================================================================
function HandlerRxSocket()
    dim data$, nSock, nLen
    rc = SocketReadData(data$, nSock, nLen)
    if rc == 0 then
        print "Socket data received : ";data$;"\n"
    endif
endfunc 1

'//==============================================================================
 }// Called when a connection to our socket has been created
'//==============================================================================
function HandlerSocketConn(nHandle)
    print "\n--- Socket Connected : ";nHandle;"\n"
    rc = SocketWrite(nHandle, data$)
endfunc 1
'==============================================
  OnEvent EVSOCKET_DATA_RECEIVED call HandlerRxSocket
  OnEvent EVSOCKETCONN call HandlerSocketConn

  // Open the socket
  rc = SocketOpenSock(path$, 0, SOCKET_FAMILY_UNIX, SOCKET_TYPE_STREAM, nHandle)

  // Connect to the opened socket
  rc = SocketConnect(path$, 0, SOCKET_FAMILY_UNIX)

  WAITEVENT

Expected Output:

  --- Socket Connected : 6
  Socket data received : This is some random data

SOCKETREaddata is an extension function.
8. EVENTS AND MESSAGES

smartBASIC is designed to be event driven, which makes it suitable for embedded platforms where it is normal to wait for something to happen and then respond.

The event handling is done synchronously, meaning the smartBASIC runtime engine has to process a WAITEVENT statement for any events or messages to be processed. This guarantees that smartBASIC never needs the complexity of locking variables and objects.

The subsystems which generate events and messages relevant to the routines described in this guide are as follows:

- BLE events and messages as described here.
- Generic Characteristics events and messages as described here.

9. MISCELLANEOUS

Result Codes

There are some operations and events that provide a single byte Bluetooth HCI result code (such as the EVDISCON message). The meaning of the result code is as per the list reproduced from the Bluetooth Specifications below. No guarantee is supplied as to its accuracy. Consult the specification for more.

Result codes in grey are not relevant to Bluetooth Low Energy operation.

<table>
<thead>
<tr>
<th>Result Code Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT_HCI_STATUS_CODE_SUCCESS</td>
<td>0x00</td>
</tr>
<tr>
<td>BT_HCI_STATUS_CODE_UNKNOWN_BTLE_COMMAND</td>
<td>0x01</td>
</tr>
<tr>
<td>BT_HCI_STATUS_CODE_UNKNOWN_CONNECTION_IDENTIFIER</td>
<td>0x02</td>
</tr>
<tr>
<td>BT_HCI_HARDWARE_FAILURE</td>
<td>0x03</td>
</tr>
<tr>
<td>BT_HCI_PAGE_TIMEOUT</td>
<td>0x04</td>
</tr>
<tr>
<td>BT_HCI_AUTHENTICATION_FAILURE</td>
<td>0x05</td>
</tr>
<tr>
<td>BT_HCI_STATUS_CODE_PIN_OR_KEY_MISSING</td>
<td>0x06</td>
</tr>
<tr>
<td>BT_HCI_MEMORY_CAPACITY_EXCEEDED</td>
<td>0x07</td>
</tr>
<tr>
<td>BT_HCI_CONNECTION_TIMEOUT</td>
<td>0x08</td>
</tr>
<tr>
<td>BT_HCI_CONNECTION_LIMIT_EXCEEDED</td>
<td>0x09</td>
</tr>
<tr>
<td>BT_HCI_SYNC_CONN_LIM_TO_A_DEVICE_EXCEEDED</td>
<td>0x0A</td>
</tr>
<tr>
<td>BT_HCI_ACL_CONN_LIM_TO_A_DEVICE_ALREADY_EXISTS</td>
<td>0x0B</td>
</tr>
<tr>
<td>BT_HCI_STATUS_CODE_COMMAND_DISALLOWED</td>
<td>0x0C</td>
</tr>
<tr>
<td>BT_HCI_CONN_REJECTED_DUE_TO_LIMITED_RESOURCES</td>
<td>0x0D</td>
</tr>
<tr>
<td>BT_HCI_CONN_REJECTED_DUE_TO_SECURITY_REASONS</td>
<td>0x0E</td>
</tr>
<tr>
<td>BT_HCI_BT_HCI_CONN_REJECTED_DUE_TO_BD_ADDR</td>
<td>0x0F</td>
</tr>
<tr>
<td>BT_HCI_CONN_ACCEPT_TIMEOUT_EXCEEDED</td>
<td>0x10</td>
</tr>
<tr>
<td>BT_HCI_UNSUPPORTED_FEATURE_ONPARM_VALUE</td>
<td>0x11</td>
</tr>
<tr>
<td>BT_HCI_REMOTE_USER_TERMINATED_CONNECTION</td>
<td>0x12</td>
</tr>
<tr>
<td>BT_HCI_REMOTE_DEV_TERMINATION_DUE_TO_LIMITED_RESOURCES</td>
<td>0x13</td>
</tr>
<tr>
<td>BT_HCI_REMOTE_DEV_TERMINATION_DUE_TO_LOW_RESOURCES</td>
<td>0x14</td>
</tr>
<tr>
<td>BT_HCI_LOCAL_HOST_TERMINATED_CONNECTION</td>
<td>0x15</td>
</tr>
<tr>
<td>BT_HCI_LOCAL_HOST_TERMINATED_CONNECTION</td>
<td>0x16</td>
</tr>
<tr>
<td>BT_HCI_REPEATED_ATTEMPTS</td>
<td>0x17</td>
</tr>
<tr>
<td>BT_HCI_PAIRING_NOTALLOWED</td>
<td>0x18</td>
</tr>
<tr>
<td>BT_HCI_LMP_PDU</td>
<td>0x19</td>
</tr>
<tr>
<td>BT_HCI_UNSUPPORTED_REMOTE_FEATURE</td>
<td>0x1A</td>
</tr>
<tr>
<td>BT_HCI SCO OFFSET REJECTED</td>
<td>0x1B</td>
</tr>
<tr>
<td>BT_HCI SCO INTERVAL REJECTED</td>
<td>0x1C</td>
</tr>
</tbody>
</table>
BT_HCI_SCO_AIR_MODE_REJECTED 0x1D
BT_HCI_STATUS_CODE_INVALID_LMP_PARAMETERS 0x1E
BT_HCI_STATUS_CODE_UNSPECIFIED_ERROR 0x1F
BT_HCI_UNSUPPORTED_LMP_PARM_VALUE 0x20
BT_HCI_ROLE_CHANGE_NOT_ALLOWED 0x21
BT_HCI_STATUS_CODE_LMP_RESPONSE_TIMEOUT 0x22
BT_HCI_LMP_ERROR_TRANSACTION_COLLISION 0x23
BT_HCI_STATUS_CODE_LMP_PDU_NOT_ALLOWED 0x24
BT_HCI_ENCRYPTION_MODE_NOT_ALLOWED 0x25
BT_HCI_LINK_KEY_CANT_NOT_BE_CHANGED 0x26
BT_HCI_REQUESTED_QOS_NOT_SUPPORTED 0x27
BT_HCI_INSTANT_PASSED 0x28
BT_HCI_PAIRING_WITH_UNIT_KEY_UNSUPPORTED 0x29
BT_HCI_DIFFERENT_TRANSACTION_COLLISION 0x2A
BT_HCI_QOS_UNACCEPTABLE_PARAMETER 0x2C
BT_HCI_QOS_REJECTED 0x2D
BT_HCI_CHANNEL_CLASSIFICATION_UNSUPPORTED 0x2E
BT_HCI_INSUFFICIENT_SECURITY 0x2F
BT_HCI_PARAMETER_OUT_OF_MANDATORY_RANGE 0x30
BT_HCI_ROLE_SWITCH_PENDING 0x32
BT_HCI_RESERVED_SLOT_VIOLATION 0x34
BT_HCI_ROLE_SWITCH_FAILED 0x35
BT_HCI_EXTENDED_INQUIRY_RESP_TOO_LARGE 0x36
BT_HCI_SSP_NOT_SUPPORTED_BY_HOST 0x37
BT_HCI_HOST_BUSY_PAIRING 0x38
BT_HCI_CONN_REJ_DUE_TO_NO_SUITABLE_CHN_FOUND 0x39
BT_HCI_CONTROLLER_BUSY 0x3A
BT_HCI_CONN_INTERVAL_UNACCEPTABLE 0x3B
BT_HCI_DIRECTED_ADVERTISER_TIMEOUT 0x3C
BT_HCI_CONN_TERMINATED_DUE_TO_MIC_FAILURE 0x3D
BT_HCI_CONN_FAILED_TO_BE_ESTABLISHED 0x3E
10. ACKNOWLEDGEMENTS

The following are required acknowledgements to address our use of open source code on the WB45 to implement AES encryption. Laird’s implementation includes the following files: aes.c and aes.h.

Copyright (c) 1998-2008, Brian Gladman, Worcester, UK. All rights reserved.

LICENSE TERMS

The redistribution and use of this software (with or without changes) is allowed without the payment of fees or royalties providing the following:

- Source code distributions include the above copyright notice, this list of conditions and the following disclaimer;
- Binary distributions include the above copyright notice, this list of conditions and the following disclaimer in their documentation;
- The name of the copyright holder is not used to endorse products built using this software without specific written permission.

DISCLAIMER

This software is provided ‘as is’ with no explicit or implied warranties in respect of its properties, including, but not limited to, correctness and/or fitness for purpose.

Issue 09/09/2006

This is an AES implementation that uses only 8-bit byte operations on the cipher state (there are options to use 32-bit types if available).

The combination of mix columns and byte substitution used here is based on that developed by Karl Malbrain. His contribution is acknowledged.
## INDEX

| AT + BTD * .................................. | 8, 10 |
| AT&F ...................................... | 9 |
| AT+RUN .................................... | 7 |
| BLECONNECT .................................. | 98 |
| BLECONNECTCANCEL ........................... | 99 |
| BLECONNECTCONFIG ........................... | 101 |
| BleDecode32 ................................ | 213 |
| BleDecodeFLOAT .............................. | 219 |
| BleDecodeS16 ................................ | 208 |
| BleDecodeS24 ................................ | 210 |
| BleDecodeSFLOAT ............................. | 215 |
| BleDecodeSTRING ............................. | 217 |
| BleDecodeTIMESTAMP .......................... | 216 |
| BLEDECODEU16 ................................ | 209 |
| BleDecodeU24 ................................ | 212 |
| BleDecodeU8 .................................. | 206, 207 |
| BleEncode16 .................................. | 197 |
| BleEncode24 .................................. | 198 |
| BleEncode32 .................................. | 199 |
| BleEncode8 ................................... | 196 |
| BleEncodeBITS ................................ | 205 |
| BleEncodeFLOAT .............................. | 199 |
| BleEncodeSFLOAT ............................. | 202 |
| BleEncodeSFLOAT ................................ | 201 |
| BleEncodeSTRING ............................. | 204 |
| BleEncodeTIMESTAMP .......................... | 203 |
| BLEGETADBYINDEX ............................. | 93 |
| BLEGETADBYTAG ................................ | 94 |
| BLESCANABORT .................................. | 86 |
| BLESCANCONFIG ................................ | 78, 89 |
| BLESCANGETADVREPORT ........................ | 78, 89 |
| BLESCANGETPAGERADDR ........................ | 91 |
| BLESCANSTART ................................ | 85 |
| BLESCANSTOP .................................. | 87, 88 |
| BLESECMGRKEYSIZES ......................... | 97, 107, 124, 156, 226 |
| BLESSVCCOMMIT ................................ | 130 |
| BLEVCREGDEVINFO ............................ | 126 |
| Bluetooth Result Codes ........................ | 235 |
| Decoding Functions .......................... | 206 |
| Encoding Functions ........................... | 195 |
| EVBLE_ADV_REPORT ............................ | 56 |
| EVBLE_ADV_TIMEOUT ........................... | 55, 56 |
| EVBLE_CONN_TIMEOUT .......................... | 97 |
| EVBLE_FAST_PAGED ............................ | 56 |
| EVBLE_SCAN_TIMEOUT ........................... | 56 |
| EVBLEMSG .................................... | 56 |
| EVBLEMSG .................................... | 56 |
| EVCHARCCCD .................................. | 61 |
| EVCHARDESC .................................. | 66 |
| EVCHARHVC ................................... | 61 |
| EVCHARSCCD .................................. | 63 |
| EVCHARVAL ................................... | 59 |
| EVDISCON .................................... | 58 |
| EVNIFYBUF .................................... | 68 |
| SYSINFO ..................................... | 11 |
| SYSINFO ..................................... | 13 |