

# User Guide

## BT900 *smart*BASIC Extensions

*Release 9.1.12.0*

---

*This guide pertains to BT900-specific smartBASIC routines and functions. For information on functions and routines that apply to all smartBASIC modules, see the [smartBASIC Core Manual](#).*

## REVISION HISTORY

Version	Date	Notes	Approver
9.1.2.0	22 Oct 2014	Initial Release	Jonathan Kaye
9.1.4.0	17 Feb 2015	Addition of RTC Alarm and Low Power Modes sections	Jonathan Kaye
9.1.4.6	5 March 2015	Added SPP parameter function, deprecate cfg keys.	Ben Whitten
9.1.7.1	28 April 2015	Updates to the following sections: BLE Security Manager Functions BTC Security Manager Functions BTC Pairing/Bonding Functions Added Class of device functions. Added BleAdvertConfig.	Ben Whitten
9.1.7.3	5 June 2015	Added HID functions Cleaned up GPIO section	Ben Whitten
9.1.8.0	24 Aug 2015	Tweaked HID API Added HID Report section Added Sniff section Transferred to new template	Ben Whitten
9.1.9.1	17 Sep 2015	Re arranged GATT Client events and messages Added StreamBridgeConfig section and event	Ben Whitten
9.1.10.0	29 Sept 2015	Rev'd up to 9.1.10.0 version; Updated StreamBridge script.	Ben Whitten
9.1.10.5	2 Feb 2016	Typo error OOB Available msg id is 25 not 23	Mahendra Tailor
9.1.10.18	20 June 2016	Add Whitelist section Added 2Mbaud Manual connection parameter negotiation Added module specific ATI section	Ben Whitten
9.1.12.0	2 September 2016	Added BleAttrMetadataEx description Corrected definition of BtcHIDRead and Write	Ben Whitten
		Added CFG Key 51 and Event EVCHARVALEX	Mahendra Tailor
	19 Jan 2017	Added SPP status and break commands and event section	Ben Whitten

© Copyright 2017 Laird. All Rights Reserved. Any information furnished by Laird and its agents is believed to be accurate and reliable. All specifications are subject to change without notice. Responsibility for the use and application of Laird materials or products rests with the end user since Laird and its agents cannot be aware of all potential uses. Laird makes no warranties as to non-infringement nor as to the fitness, merchantability, or sustainability of any Laird materials or products for any specific or general uses. Laird, Laird Technologies, Inc., or any of its affiliates or agents shall not be liable for incidental or consequential damages of any kind. All Laird products are sold pursuant to the Laird Terms and Conditions of Sale in effect from time to time, a copy of which will be furnished upon request. When used as a tradename herein, *Laird* means Laird PLC or one or more subsidiaries of Laird PLC (Laird Technologies, Inc; Laird Technologies; Laird – Lenexa; Laird – Akron; Laird – Taiwan; Laird – Wooburn; Laird – Taiwan (or Zhubei City); Summit Data Communications, Inc.; Ezurio, Ltd.; Aerocomm, Inc.). Laird™, Laird Technologies™, corresponding logos, and other marks are trademarks or registered trademarks of Laird. Other marks may be the property of third parties. Nothing herein provides a license under any Laird or any third party intellectual property right.

## CONTENTS

Introduction.....	13
What Does a BTC/BLE Module Contain? .....	13
Module Configuration .....	14
Interactive Mode Commands.....	14
AT I or ATl .....	14
AT I num.....	14
AT+CFG .....	16
AT+BTd * .....	20
AT+BLX.....	20
AT&F .....	21
Core Language Built-in Routines .....	21
Information Routines .....	21
SYSINFO .....	21
SYSINFO\$ .....	24
UART Interface .....	25
UartOpen .....	25
I2C – Two Wire Interface (TWI) .....	25
SPI Interface.....	26
SpiOpen .....	26
Input/Output Interface Routines.....	27
Events and Messages.....	28
GpioSetFunc.....	28
GpioConfigPwm .....	29
GpioRead .....	32
GpioWrite .....	33
GpioBindEvent/GpioAssignEvent .....	36
GpioUnbindEvent/GpioUnAssignEvent .....	38
Miscellaneous Routines.....	38
ERASEFILESYSTEM .....	38
BTC Extensions Built-in Routines.....	40
Generic Access Profile Functions.....	40
Events and Messages.....	40

EVINQRESP .....	40
EVBTC_INQUIRY_TIMEOUT .....	42
BtcInquiryConfig .....	42
BtcInquiryStart.....	43
BtcInquiryCancel.....	43
BtcDiscoveryConfig.....	44
BtcSetDiscoverable .....	46
BtcSetConnectable .....	47
BtcSetPairable .....	48
BtcInquiryGetReport.....	48
BtcInquiryGetReportFull.....	49
BtcGetClassOfDevice .....	50
BtcSetClassOfDevice .....	50
BtcGetEIRbyIndex .....	51
BtcGetEIRbyTag .....	53
BtcGetFriendlyName .....	55
BtcGetRemoteFriendlyName.....	56
BtcQueryRemoteFriendlyName .....	56
BtcSetFriendlyName .....	56
BtcSniffEnable.....	57
BtcSniffDisable.....	59
BtcQuerySniffSubrating .....	61
BtcQueryModeChange .....	64
BtcSniffSubratingEnable .....	66
Human Interface Device .....	69
Events and Messages.....	69
EVHIDCONN .....	69
EVHIDDISCON.....	70
EVHIDCONTROL.....	71
EVHIDTXEMPTY.....	71
EVBTC_HID_DATA_RECEIVED .....	72
BtcHIDDeviceOpen .....	73

BtcHIDHostOpen.....	74
BtcHIDClose .....	75
BtcHIDConnect .....	75
BtcHIDDisconnect .....	76
BtcHIDRead.....	78
BtcHIDWrite.....	80
BtcHIDControl.....	81
BtcHIDConfig.....	83
Serial Port Profile.....	83
Events and Messages.....	83
EVSPPCONN .....	83
EVBTC_SPP_CONN_TIMEOUT.....	84
EVBTC_SPP_DATA_RECEIVED .....	84
EVSPPTXEMPTY.....	84
EVSPPDISCON.....	84
BtcSPPSetParams.....	87
BtcSPPOpen .....	87
BtcSPPClose .....	88
BtcSPPWrite.....	89
BtcSPPRead.....	90
BtcSPPConnect .....	92
BtcSPPDisconnect .....	93
Stream Functions.....	95
Events and Messages.....	95
EVSTREAMIDLE .....	95
StreamGetUartHandle .....	95
StreamGetSPPHandle .....	96
StreamBridge .....	96
StreamUnBridge .....	98
StreamBridgeConfig.....	99
Pairing, Bonding, and Security Manager Functions.....	99
Events and Messages.....	99

EVBTC_PAIR_REQUEST.....	99
EVBTC_OOB_AVAILABLE_REQUEST.....	100
EVBTC_PIN_REQUEST.....	101
EVBTC_PAIR_RESULT .....	101
EVBTC_AUTHREQ.....	103
EVBTC_PASSKEY .....	105
BtcGetPAIRRequestBDAddr.....	105
BtcGetPINRequestBDAddr.....	105
BtcSendPAIRResp .....	105
BtcSendPINResp .....	106
BtcSavePairings.....	106
BtcPair.....	107
BtcBondingStats .....	109
BtcBondingEraseKey.....	110
BtcBondingEraseAll.....	111
BtcBondingPersistKey.....	112
BtcBondingGetFirst.....	113
BtcBondingGetNext .....	115
BtcSecMngrPasskey.....	116
BtcSecMngrJustWorksConf .....	118
BtcSecMngrOOBAvailable .....	119
BtcSecMngrOOBPref .....	121
BtcSecMngrRetrieveLocalOOBKey .....	122
BtcSecMngrOOBKey .....	123
BtcSecMngrIoCap .....	125
Miscellaneous Functions .....	127
Events and Messages.....	127
EVBTC_DISCOV_TIMEOUT .....	127
EVBTC_REMOTENAME_RECEIVED .....	127
EVBTC_MODE_CHANGE .....	127
EVBTC_SNIFF_SUBRATING.....	127
BtcTxPowerSet.....	127

BtcSetPNPInformation.....	128
BtcGetBDAddrFromHandle .....	129
BtcGetHandleFromBDAddr .....	129
BLE Extensions Built-in Routines .....	131
Bluetooth Address .....	131
BleSetAddressType .....	132
Events and Messages.....	132
EVBLE_ADV_TIMEOUT.....	132
EVBLE_CONN_TIMEOUT.....	133
EVBLE_ADV_REPORT .....	133
EVBLE_FAST_PAGED.....	133
EVBLE_SCAN_TIMEOUT.....	133
EVBLEMSG .....	134
EVDISCON .....	136
EVCONNPARAMREQ.....	137
EVCHARVALEX .....	138
EVCHARVAL .....	140
EVCHARHVC.....	143
EVCHARCCCD .....	143
EVCHARSCCD .....	146
EVCHARDESC .....	151
EVNOTIFYBUF .....	154
Miscellaneous Functions .....	157
BleTxPowerSet.....	157
BleTxPwrWhilePairing .....	158
BleGetConnHandleFromAddr .....	160
BleGetAddrFromConnHandle.....	162
Advertising Functions .....	164
BleAdvertStart .....	164
BleAdvertStop.....	166
BleAdvertConfig.....	167
BleAdvRptInit.....	168
BleScanRptInit .....	169

BleAdvRptGetSpace.....	169
BleAdvRptAddUuid16.....	170
BleAdvRptAddUuid128.....	171
BleAdvRptAppendAD.....	172
BleAdvRptsCommit.....	173
Scanning Functions.....	174
BleScanStart.....	174
BleScanAbort .....	176
BleScanStop .....	177
BleScanFlush .....	178
BleScanConfig .....	179
BleScanGetAdvReport .....	181
BleScanGetAdvReportEx.....	184
BleGetADbyIndex.....	184
BleGetADbyTag.....	186
BleScanGetPagerAddr .....	188
Connection Functions.....	189
Events and Messages.....	189
BleConnect.....	190
BleConnectCancel.....	192
BleConnectConfig .....	194
BleDisconnect .....	196
BleSetCurConnParms.....	197
BleGetCurConnParms .....	200
BleConnMngrUpdCfg.....	201
Whitelist Management Functions .....	201
BleWhitelistCreate.....	202
BleWhitelistDestroy.....	205
BleWhitelistClear .....	205
BleWhitelistSetFilter .....	206
BleWhitelistAddAddr .....	206
BleWhitelistAddIndex .....	207
BleWhitelistInfo .....	207



GATT Server Functions .....	208
Events and Messages.....	215
BleGapSvcInit.....	215
BleGetDeviceName\$.....	217
BleSvcRegDevInfo .....	218
BleHandleUuid16.....	219
BleHandleUuid128.....	220
BleHandleUuidSibling .....	221
BleServiceNew .....	222
BleServiceCommit.....	224
BleSvcAddIncludeSvc .....	224
BleAttrMetadata.....	226
BleAttrMetadataEx .....	228
BleCharNew .....	230
BleCharDescUserDesc.....	232
BleCharDescPrstnFrmt.....	233
BleCharDescAdd .....	235
BleCharCommit.....	237
BleCharValueRead .....	239
BleCharValueWrite .....	241
BleCharValueNotify .....	243
BleCharValueIndicate .....	246
BleCharDescRead.....	249
GATT Client Functions .....	252
Events and Messages.....	254
EVGATTCTOUT .....	254
EVDISCPRIMSVCS .....	256
EVDISCCHAR .....	256
EVDISCDESC .....	257
EVFINDCHAR.....	257
EVFINDDESC.....	257
EVATTRREAD .....	258

<i>EVATTRWRITE</i> .....	258
<i>EVNOTIFYBUF</i> .....	259
<i>EVATTRNOTIFY</i> .....	259
BleGattcOpen .....	259
BleGattcClose .....	261
BleDiscServiceFirst / BleDiscServiceNext .....	262
BleDiscCharFirst / BleDiscCharNext .....	267
BleDiscDescFirst / BleDiscDescNext .....	274
BleGattcFindChar .....	280
BleGattcFindDesc .....	285
BleGattcRead/BleGattcReadData .....	290
BleGattcWrite .....	294
BleGattcWriteCmd .....	298
BleGattcNotifyRead .....	301
Attribute Encoding Functions .....	306
BleEncode8 .....	306
BleEncode16 .....	307
BleEncode24 .....	308
BleEncode32 .....	309
BleEncodeFLOAT .....	310
BleEncodeSFLOAT .....	311
BleEncodeSFLOAT .....	312
BleEncodeTIMESTAMP .....	314
BleEncodeSTRING .....	315
BleEncodeBITS .....	315
Attribute Decoding Functions .....	316
BleDecodeS8 .....	317
BleDecodeU8 .....	318
BleDecodeS16 .....	319
BleDecodeU16 .....	321
BleDecodeS24 .....	322
BleDecodeU24 .....	323
BleDecode32 .....	325

BleDecodeFLOAT .....	326
BleDecodeSFLOAT .....	327
BleDecodeTIMESTAMP .....	329
BleDecodeSTRING .....	330
BleDecodeBITS .....	331
Pairing, Bonding, and Security Manager Functions .....	333
Pairing and Bonding Functions .....	333
BleBondingStats .....	333
BleBondingPersistKey .....	334
BleBondingIsTrusted .....	335
BleBondingEraseKey .....	335
BleBondingEraseAll .....	336
BleBondMngrGetInfo .....	337
Security Manager Functions .....	338
Events and Messages .....	338
BleSecMngrJustWorksConf .....	338
BleSecMngrOobPref .....	338
BleSecMngrOobAvailable .....	339
BleAcceptPairing .....	339
BleSecMngrPasskey .....	340
BleSecMngrOOBkey .....	342
BleSecMngrKeySizes .....	344
BleSecMngrIoCap .....	344
BleSecMngrBondReq .....	345
BlePair .....	345
BleEncryptConnection .....	349
HID Report parsing .....	352
HIDReportInit .....	352
HIDReportAppendInt .....	353
HIDReportAppendStr .....	354
HIDReportImport .....	355

HIDReportExport .....	356
HIDReportExtractInt .....	357
HIDReportExtractStr .....	358
HIDReportDestroy .....	359
RTC Alarm .....	360
RTCSetTime.....	361
RTCGetTime\$ .....	362
RTCGetTime .....	362
RTCSetAlarm .....	363
RTCSetAlarmDuration.....	364
RTCGetAlarm\$ .....	366
RTCGetAlarm .....	366
RTCSetFormat.....	367
RTCSetMinuteAlarm .....	369
RTCSetHourAlarm.....	371
RTCSetDayAlarm.....	372
RTCReset.....	373
Low Power Modes .....	374
Events and Messages.....	375
Miscellaneous.....	376
Bluetooth Result Codes .....	376
Acknowledgements .....	378
License Terms.....	378
Disclaimer .....	378

## INTRODUCTION

This user guide provides detailed information on BT900-specific *smartBASIC* extensions which provides a high-level managed interface to the underlying Bluetooth stack in order to manage the following:

- Bluetooth Classic (BTC) Inquiries, discovery, connections.
- Serial Port Profile (SPP) and Human Interface Device (HID).
- Bluetooth Low Energy (BLE) security and bonding
- BLE advertisements and connections
- GATT Table: Services, characteristics, descriptors.
- GATT server/client operation
- Attribute encoding and decoding
- Laird custom VSP service
- Power management
- Wireless status
- Events related to the above

## What Does a BTC/BLE Module Contain?

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

- A highly integrated radio with an integrated antenna (external antenna options are also available)
- BTC/BLE Physical and Link layer
- Higher level stack
- Multiple SIO and ADC
- Wired communication interfaces such as UART, I2C, and SPI
- A *smartBASIC* run-time engine
- Program accessible flash memory, which contains a robust flash file system exposing a conventional file system and a database for storing user configuration data

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

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

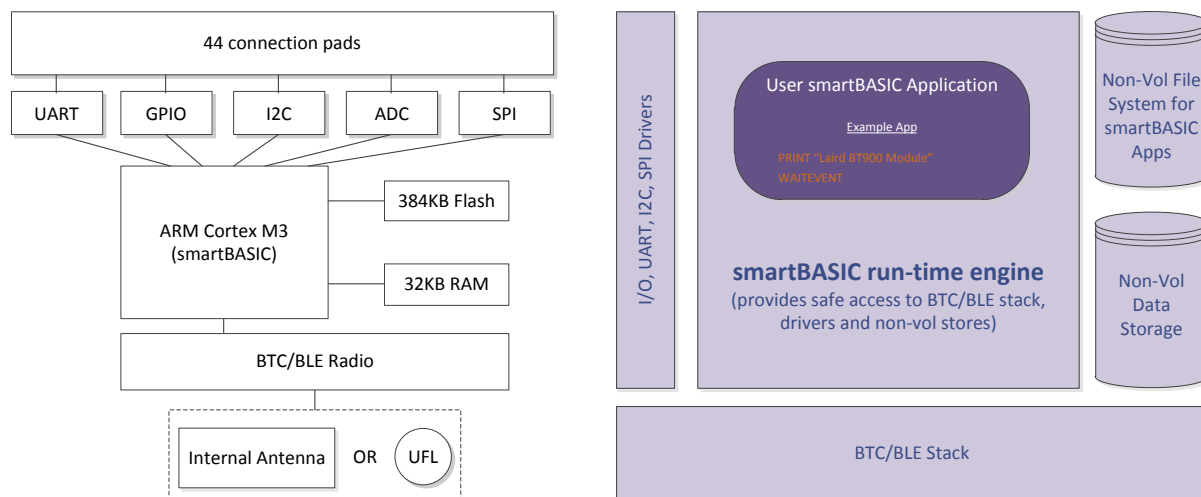


Figure 1: Bluetooth *smartBASIC* module block diagram

## MODULE CONFIGURATION

There are many features of the module that cannot be modified programmatically which relate to interactive mode operation or alter the behaviour of the smartBASIC runtime engine. These configuration objects are stored in non-volatile flash and are retained until the flash file system is erased via AT&F\* or AT&F 1.

To write to these objects, which are identified by a positive integer number, the module must be in interactive mode and the command AT+CFG must be used. To read current values of these objects use the command AT+CFG, described [here](#).

Predefined configuration objects are as listed under details of the AT+CFG command.

## INTERACTIVE MODE COMMANDS

Below are some BT900-specific AT commands. In the case of ATI we list the extensions to core.

### ATI or ATI

#### COMMAND

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

#### ATI *num*

<b>Returns</b>	\n10\tMM\tInformation\r \n00\r Where \n = linefeed character 0x0A \t = horizontal tab character 0x09 MM = a <i>number</i> (see below) Information = string consisting of information requested associated with MM \r = carriage return character 0x0D	
<b>Arguments</b>	<b><i>num</i></b> Integer Constant A number in the range of 0 to 65,535. Currently defined numbers are:	
	1	Stack version
	3	Version number of module firmware
	4	IEEE BT Address
	14	Random static BLE address
	26	BTE Bonding database segment
	36	BTC Bonding database segment
	44	Current random BLE address
	2000	Reset persistence value
	2001	Reset reason
	2002	Timer resolution
	2003	Number of timers
	2004	Tick resolution

	2005	LMP version
	2006	LMP Sub version
	2007	Company ID
	2008	BLE transmit power
	2009	Total number of BLE bonds
	2041	As above
	2042	Number of rolling bonds
	2010	Total BLE bonds with IRK
	2011	Total BLE bonds with CSRK
	2012	Maximum BLE bonds
	2018	Current BLE TX power
	2040	As above
	2043	Minimum rolling BLE bonds
	2013	Max attribute length
	2014	Number of BLE transmission buffers (Unused)
	2015	Unused BLE transmission buffers (Unused)
	2016	Radio activity state: Bitmask : 0 advertising, 1 connected, 2 Scanning, 3 Initiating
	2018	BLE transmit power whilst pairing
	2019	Default ring buffer size for notify/indicates
	2020	Maximum ring buffer size for notify/indicates
	2021	Stack side mark
	2022	Stack size in bytes
	2032	Initial heap size
	2025	MCU frequency trimming value
	2026	MCU temperature trimming value
	2050	Maximum BTC bonds
	2051	Total BTC bonds
	2052	Rolling BTC bonds
	2100	Dropped stack events
	2300	MCU operating frequency
	2301	ADC config
	2302	RTC format
	4020	Stack memory usage
	4021	Stack free memory
	4022	Largest fragment available to stack
	4023	Stack failed allocations
<b>Interactive Command</b>	Yes	

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

**Example :**

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

AT i is a core command with module specific entries

## 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 ?

<b>Returns</b>	If the config key is successfully updated or read, the response is \n00\r.
<b>Arguments:</b>	
<b>num</b>	Integer Constant The ID of the required configuration key. All of the configuration keys are stored as an array of 16-bit words.
<b>value</b>	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.

ID	Definition
40	Maximum size of local simple variables
41	Maximum size of local complex variables
42	Maximum depth of nested user-defined functions and subroutines
43	The size of stack for storing user functions' simple variables
44	The size of stack for storing user functions' complex variables
45	The size of the message argument queue length
51	0 to disable strings in event messages and 1 to enable



ID	Definition
100	Enable/Disable Virtual Serial Port Service when in interactive mode. Valid values are:
	0x0000    Disable
	0x0001    Enable
	0x80nn    Enable VSP command mode if Signal Pin <b>nn</b> on module is HIGH
	0xC0nn    Enable VSP command mode if Signal Pin <b>nn</b> on module is LOW
	0x81nn    Enable VSP command/bridge mode if Signal Pin <b>nn</b> on module is HIGH, command/bridge mode is selected with nAutorun at startup. nAutorun LOW for command mode, HIGH for bridge mode.
	0xC1nn    Enable VSP command/bridge mode if Signal Pin <b>nn</b> on module is LOW, command/bridge mode is selected with nAutorun at startup. nAutorun LOW for command mode, HIGH for bridge mode.
	ELSE        Disable
101	In Virtual Serial Port Service, select either to use INDICATE or NOTIFY to send data to client.
	0            Prefer Notify
	ELSE        Prefer Indicate
	This is a preference and the actual value is forced by the property of the TX characteristic of the service.
102	Advert interval in milliseconds when advertising for connections in interactive mode and AT Parse mode. Valid values: 20 to 10240 milliseconds
103	Advert timeout in milliseconds when advertising for connections in interactive mode and AT Parse mode. Valid values: 1 to 16383 seconds
104	Data transfer is managed in the Virtual Serial Port service manager. When sending data using NOTIFIES, the underlying stack uses transmission buffers of which there is a finite number. This specifies the number of transmissions to leave unused when sending a lot of data and allows other services to send notifies without having to wait for them. The total number of transmission buffers can be determined by calling SYSINFO(2014) or in interactive mode submitting the command ATi 2014
105	When in interactive mode and connected for virtual serial port services, this is the minimum connection interval in milliseconds to be negotiated with the master. Valid values: 0 to 4000 ms. If a value of less than 8 is specified, then the minimum value of 7.5 is selected.
106	When in interactive mode and connected for virtual serial port services, this is the maximum connection interval in milliseconds to be negotiated with the master. Valid values: 0 to 4000 ms. <b>Note:</b> If a value of less the minimum specified in 105, then it is forced to the value in 105 plus 2 milliseconds.
107	When in interactive mode and connected for virtual serial port services, this is the connection supervision timeout in milliseconds to be negotiated with the master. Valid range: 0 to 32000. <b>Note:</b> If the value is less than the value in 106, then a value double the one in 106 is used.

ID	Definition																																
108	When in interactive mode and connected for virtual serial port services, this is the slave latency to be negotiated with the master. An adjusted value is used if this value times the value in 106 is greater than the supervision timeout in 107																																
109	When in interactive mode and connected for virtual serial port services, this is the Tx power used for adverts and connections. The main reason for setting a low value is to ensure that in production, if <i>smartBASIC</i> applications are downloaded over the air, limited range allows many stations to be used to program devices.																																
110	If Virtual Serial Port Service is enabled in interactive mode (see 100), this specifies the size of the transmit ring buffer in the managed layer sitting above the service characteristic FIFO register. Valid range: 32 to 256																																
111	If Virtual Serial Port Service is enabled in interactive mode (see 100), this specifies the size of the receive ring buffer in the managed layer sitting above the service characteristic fifo register. Valid range: 32 to 256																																
250	Deprecated, please refer to BtcSPPSetParams for alternative method.																																
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.																																
400	BT/Wi-Fi coexistence setting. Valid values are: <table> <tr> <td>0</td><td>Disabled</td></tr> <tr> <td>1</td><td>Unity3 (no BLE support)</td></tr> <tr> <td>2</td><td>Unity3e (BLE support)</td></tr> </table>	0	Disabled	1	Unity3 (no BLE support)	2	Unity3e (BLE support)																										
0	Disabled																																
1	Unity3 (no BLE support)																																
2	Unity3e (BLE support)																																
401	Controls transaction priorities, bit encoded (0 = low priority, 1 = high priority): <table> <tr> <th>Bit</th><th>Type</th></tr> <tr> <td>0</td><td>Page</td></tr> <tr> <td>1</td><td>Page Scan</td></tr> <tr> <td>2</td><td>Inquiry</td></tr> <tr> <td>3</td><td>Inquiry Scan</td></tr> <tr> <td>4</td><td>Role switch</td></tr> <tr> <td>5</td><td>LMP transmission to master</td></tr> <tr> <td>6</td><td>LMP from master</td></tr> <tr> <td>7</td><td>Polling (Tpoll)</td></tr> <tr> <td>8</td><td>Start of sniff</td></tr> <tr> <td>9</td><td>Bulk ACL</td></tr> <tr> <td>10</td><td>Broadcast transmissions</td></tr> <tr> <td>11</td><td>Park</td></tr> <tr> <td>12</td><td>Band Scan</td></tr> <tr> <td>13</td><td>Conditional Scan</td></tr> <tr> <td>14</td><td>Radio Trim</td></tr> </table>	Bit	Type	0	Page	1	Page Scan	2	Inquiry	3	Inquiry Scan	4	Role switch	5	LMP transmission to master	6	LMP from master	7	Polling (Tpoll)	8	Start of sniff	9	Bulk ACL	10	Broadcast transmissions	11	Park	12	Band Scan	13	Conditional Scan	14	Radio Trim
Bit	Type																																
0	Page																																
1	Page Scan																																
2	Inquiry																																
3	Inquiry Scan																																
4	Role switch																																
5	LMP transmission to master																																
6	LMP from master																																
7	Polling (Tpoll)																																
8	Start of sniff																																
9	Bulk ACL																																
10	Broadcast transmissions																																
11	Park																																
12	Band Scan																																
13	Conditional Scan																																
14	Radio Trim																																
402	Controls transaction priorities, bit encoded (0 = low priority, 1 = high priority):																																

ID	Definition	
	Bit	Type
	0	Non-connectable Advertising
	1	Discoverable Advertising
	2	Connectable Undirected Advertising
	3	Connectable Directed Advertising
	4	Advertising - Scan Response
	5	Passive Scanning
	6	Active Scanning
	7	Active Scanning - Scan Response
	8	Initiator
	9	Connection Establishment (Master)
	10	Connection Establishment (Slave)
	11	Anchor Point (Master)
	12	Anchor Point (Slave)
	13	Data (Master)
	14	Data (Slave)
	15	<Reserved>
403	Unity3 T1 timing Valid values: 150 to 200 microseconds. Default – 150	
404	Unity3 T2 timing Valid values: 15 to 20 microseconds. Default – 17	
405	Unity3 Active Lead timing Valid values: 18 to 22 microseconds. Default – 20	
406	Unity3 Status Lead timing Valid values: 8 to 12 microseconds. Default – 10	
2300	Running clock frequency in KHz. Valid values are 4000 (4 MHz), 20000 (20 MHz), and 40000 (40 MHz). Default – 40 MHz. Type <b>AT I 2300 ?</b> in UwTerminal and hit <b>Enter</b> to get the current clock frequency in Hz. <b>Note:</b> When using the 4 MHz clock, the maximum supported baud rate is 115200.	
2301	Analogue-to-digital converter configuration scaling. Valid values:	
	0	VRef 3.3V, scale to BL600
	1	VRef 1.8V, scale to BL600
	2	VRef 3.3V, no scaling
2302	Configures the format of the RTC time and date output string	
	1	hh:mm:ss
	2	dd/mm/yy
	3	yy/mm/dd
	4	hh:mm:ss dd/mm/yy

ID	Definition
5	hh:mm:ss yy/mm/dd
6	dd/mm/yy hh:mm:ss
7	yy/mm/dd hh:mm:ss

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\*

Returns	\n00\r
Arguments	None

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

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

### Example:

```
AT+BTD*
```

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. It is particularly useful when the virtual serial port is enabled while in interactive mode.

#### AT+BLX

Returns	\n00\r
Arguments:	None

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

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

### Example:

```
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*

<b>Returns</b>	OK if flash is successfully erased
<b>Arguments</b>	
<b><i>Integermask</i></b>	Integer corresponding to a bit mask or the * character

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

<b>1</b>	Erases normal file system and system config keys (see <a href="#">AT+CFG</a> for examples of config keys)
<b>0x40000</b>	Erases the User config keys only
<b>0x10000</b>	Erase the BLE Bonding Manager
<b>0x20000</b>	Erases the Classic Bluetooth Bonding Manager
<b>*</b>	Erases all data segments
<b>Else</b>	Not applicable to current modules

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

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

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

AT&F is a core command.

## CORE LANGUAGE BUILT-IN ROUTINES

Core language built-in routines are present in every implementation of *smart*BASIC. These routines provide the basic programming functionality. They are augmented with target-specific routines for different platforms which are described in the extension manual for each target platform.

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

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

## Information Routines

### SYSINFO

#### FUNCTION

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

#### *SYSINFO(varId)*

<b>Returns</b>	INTEGER. Value of information corresponding to integer ID requested.
----------------	--

Exceptions	<ul style="list-style-type: none"> <li>Local Stack Frame Underflow</li> <li>Local Stack Frame Overflow</li> </ul>
Arguments:	
<i>varId</i>	byVal varId AS INTEGER An integer ID which is used to determine which information is to be returned as described below.
	0 Device ID. Each platform type has a unique identifier.
	3 Module firmware version number Example: X.Y.Z is returned as a 32-bit value made up as follows: <b><math>(X \ll 24) + (Y \ll 8) + (Z)</math></b> where Y is the build number and Z is the sub-build number
	33 BASIC core version number
	601 Flash File System: Data Segment: Total Space
	602 Flash File System: Data Segment: Free Space
	603 Flash File System: Data Segment: Deleted Space
	611 Flash File System: FAT Segment: Total Space
	612 Flash File System: FAT Segment: Free Space
	613 Flash File System: FAT Segment: Deleted Space
	631 NvRecord Memory Store Segment: Total Space
	632 NvRecord Memory Store Segment: Free Space
	633 NvRecord Memory Store Segment: Deleted Space
	1000 BASIC compiler HASH value as a 32 bit decimal value
	1001 How RAND() generates values: 0 for PRNG and 1 for hardware assist
	1002 Minimum baudrate
	1003 Maximum baudrate
	1004 Maximum STRING size
	1005 Is 1 for run-time only implementation, 3 for compiler included
	1010 Module Type
	2000 Reset Reason <ul style="list-style-type: none"> <li>8 : Self-Reset due to Flash Erase</li> <li>9 : ATZ</li> <li>10 : Self-Reset due to smartBASIC app invoking function RESET()</li> </ul>
	2001 Cause of last reset
	2002 Timer resolution in microseconds
	2003 Number of timers available in a smartBASIC Application
	2004 Tick timer resolution in microseconds
	2005 LMP Version number for BT 4.0 spec
	2006 LMP Sub Version number
	2007 Chipset Company ID allocated by BT SIG
	2008 Returns the current TX power setting (see also 2018)
	2009 Number of devices in trusted device database
	2010 Number of devices in trusted device database with IRK
	2011 Number of devices in trusted device database with CSRK
	2012 Max number of devices that can be stored in trusted device database

2013	Maximum length of a GATT Table attribute in this implementation
2014	Total number of transmission buffers for sending attribute NOTIFIES
2015	Number of transmission buffers for sending attribute NOTIFIES – free
2016	Radio activity of the baseband <ul style="list-style-type: none"> <li>0 : no activity</li> <li>1 : advertising</li> <li>2 : connected</li> <li>3 : broadcasting and connected</li> </ul>
2018	Returns the TX power while pairing in progress (see also 2008)
2021	Stack tide mark in percent. Values near 100 are not good.
2022	Stack size
2023	Initial Heap size
2040	Max number of devices that can be stored in trusted device database
2041	Number of devices in trusted device database
2042	Number of devices in the rolling device database
2043	Maximum number of devices that can be stored in the rolling device Database
2100	Connect scan interval (ms)
2101	Connect scan window (ms)
2102	Connect slave latency (ms)
2105	Connect multi-link connection interval periodicity (ms)
2106	Minimum connection length (ms)
2107	Maximum connection length (ms)
2150	Scan interval (ms)
2151	Scan window (ms)
2152	Scan type <ul style="list-style-type: none"> <li>0 – Passive</li> <li>1 – Active</li> </ul>
2153	Minimum number of reports to store in cache
2300	Returns system clock frequency. Value should be one of 4MHz, 20MHz of 40MHz.
2301	Raw Adc Config <ul style="list-style-type: none"> <li>0 – Scaled to match BL600 output</li> <li>2 – No scaling</li> </ul>
2303	RTC time and date output format <ul style="list-style-type: none"> <li>1 – hh:mm:ss</li> <li>2 – dd/mm/yy</li> <li>3 – yy/mm/dd</li> <li>4 – hh:mm:ss dd/mm/yy</li> <li>5 – hh:mm:ss yy/mm/dd</li> <li>6 – dd/mm/yy hh:mm:ss</li> <li>7 – yy/mm/dd hh:mm:ss</li> </ul>

### Example :: SysInfo.sb (See in Firmware Zip file)

```
PRINT "\nSysInfo 601 = ";SYSINFO(601) // Flash File System: Total Space (Data Segment)
PRINT "\nSysInfo 2300 = ";SYSINFO(2300) // Current system clock speed
PRINT "\nSysInfo 1002 = ";SYSINFO(1002) // Minimum UART baud rate
```

### Expected Output:

```
SysInfo 601 = 49152
SysInfo 2300 = 40000000
SysInfo 1002 = 1200
```

SYSINFO is a core language function.

## SYSINFO\$

### FUNCTION

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

#### *SYSINFO\$(varId)*

<b>Returns</b>	STRING. Value of information corresponding to integer ID requested.
<b>Exceptions</b>	<ul style="list-style-type: none"> <li>Local Stack Frame Underflow</li> <li>Local Stack Frame Overflow</li> </ul>

#### Arguments:

<b><i>varId</i></b>	<p><i>byVal</i> <i>varId</i> AS INTEGER</p> <p>An integer ID which is used to determine which information is to be returned as described below.</p> <table border="1"> <tr> <td>4</td><td> <p>The Bluetooth address of the module.</p> <p>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.</p> </td></tr> <tr> <td>14</td><td> <p>A random public address unique to this module. May be the same value as in 4 above unless an IEEE Bluetooth address is set.</p> <p>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.</p> </td></tr> </table>	4	<p>The Bluetooth address of the module.</p> <p>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.</p>	14	<p>A random public address unique to this module. May be the same value as in 4 above unless an IEEE Bluetooth address is set.</p> <p>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.</p>
4	<p>The Bluetooth address of the module.</p> <p>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.</p>				
14	<p>A random public address unique to this module. May be the same value as in 4 above unless an IEEE Bluetooth address is set.</p> <p>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.</p>				

### Example :: SysInfo\$.sb (See in Firmware Zip file)

```
PRINT "\nSysInfo$(4) = ";SYSINFO$(4) // address of module
PRINT "\nSysInfo$(14) = ";SYSINFO$(14) // public random address
PRINT "\nSysInfo$(0) = ";SYSINFO$(0)
```

### Expected Output:

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



SYSINFO\$ is a core language function.

## 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.

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

---

##### *byVal stOptions AS STRING*

This string (can be a constant) MUST be exactly 5 characters long where each character is used to specify further comms parameters as follows.

Character Offset:

<i>stOptions</i>	0	DTE/DCE role request: <ul style="list-style-type: none"><li>▪ <b>T</b> – DTE</li><li>▪ <b>C</b> – DCE</li></ul>
	1	Parity: <ul style="list-style-type: none"><li>▪ <b>N</b> – None</li><li>▪ <b>O</b> – Odd</li><li>▪ <b>E</b> – Even</li></ul>
	2	Databits: 8
	3	Stopbits: 1
	4	Flow Control: <ul style="list-style-type: none"><li>▪ <b>N</b> – None</li><li>▪ <b>H</b> – CTS/RTS hardware</li><li>▪ <b>X</b> – Xon/Xof (Not Available)</li></ul>

---

The following baud rates are supported: 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400, 250000, 460800, 921600, and 2000000 baud.

## I2C – Two Wire Interface (TWI)

The BT900 can be only be configured as an I2C master if it is the only master on the bus and only 7-bit slave addressing is supported.

BT900 supports three clock frequencies: 100000, 250000 and 400000. The values must be entered in this format.

---

**Note:** I2C can only supported on modules that have been configured with a System Clock of 20 or 40MHz. It is not supported in modules with a 4MHz System Clock.

---

## SPI Interface

The BT900 module can only be configured as a SPI master.

### SpiOpen

#### FUNCTION

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

#### *SPIOPEN (nMode, nClockHz, nCfgFlags, nHandle)*

Returns	INTEGER Indicates success of command:		
	0	Opened successfully	
	0x5200	Driver not found	
	0x5207	Driver already open	
	0x5225	Invalid clock frequency requested	
	0x521D	Driver resource unavailable	
	0x522B	Invalid	
Exceptions	<ul style="list-style-type: none"><li>Local Stack Frame Underflow</li><li>Local Stack Frame Overflow</li></ul>		
Arguments			
nMode	byVal nMode AS INTEGER		
	This is the mode, as in phase and polarity of the clock line, that the interface shall operate at. Valid values are 0 to 3 inclusive:		
	Mode	CPOL	CPHA
	0	0	0
	1	0	1
nClockHz	byVal nClockHz AS INTEGER		
	This is the clock frequency to use, and can be one of 125000, 250000, 500000, 1000000, 2000000, 4000000 or 8000000.		
	byVal nCfgFlags AS INTEGER		
nCfgFlags	This is a bit mask used to configure the SPI interface. All unused bits are allocated as <i>for future use</i> and MUST be set to 0. Used bits are as follows:		
	Bit	Description	
	0	If set, then the least significant bit is clocked in/out first.	
nHandle	1-31 Unused and <i>must</i> be set to 0.		
	byRef nHandle AS INTEGER		
	The handle for this interface is returned in this variable if it is successfully opened. This handle is subsequently used to read/write and close the interface.		
Related Commands	SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD		

SPIOPEN is a core function.

**Note:** Firmware versions prior to 9.1.10.3 contained a bug relating to the nMode parameter. This parameter had been implemented incorrectly. Entering a value of 1 would actually result in SPI Mode 0 and a value of 0 would result in SPI Mode 2. This has now been corrected and from firmware version 9.1.10.3 onwards the nMode parameter has been implemented correctly.

Any *smartBasic* applications using this command on pre-9.1.10.3 firmware should be corrected accordingly.

## Input/Output Interface Routines

I/O and interface commands allow access to the physical interface pins and ports of the *smartBASIC* modules. Most of these commands are applicable to the entire range of modules. However, some are dependent on the actual I/O availability of each module.

There are 23 SIO (Special I/O) pins available on the BT900. All of these pins can be configured to provide additional types of functionality. However, some of the pins have set functionality that should never be changed. For example, sio0-sio3 are used to provide the Serial UART capability to a remote host.

The functionality that is supported for each pin is listed below. With the exception of the sio0 to sio3 and sio 14 to sio16, all of the pins boot up as digital inputs.

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

**Table 1: SIO pin functionality**

SIO	Functionality
0	UART Rx, Wakeup pin 2, External interrupt
1	UART Tx
2	UartRts
3	UartCts, Wakeup pin 4, External interrupt
4	No extra functionality
5	External interrupt
6	SpiMiso
7	SpiMosi
8	SpiCs, External interrupt
9	SpiClk
10	I2cData
11	I2cClock
12	Pwm/Freq
13	Pwm/Freq, External interrupt
14	No extra functionality
15	No extra functionality
16	No extra functionality
17	Pwm/Freq

SIO	Functionality
18	No extra functionality
19	Vsp
20	Adc01, Wakeup pin 1, external interrupt
21	Adc00
22	Autorun, external interrupt

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

Pins that provide the External interrupt functionality are used for the GpioBindEvents and GpioAssignEvents. With this firmware, Bind and Assign are handled identically. It is only the type of returned event that is different.

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

## Events and Messages

<b>EVGPIOCHANn</b>	N is from 0 to 3 where N is platform-dependent and an event is generated when a preconfigured digital input transition occurs. The number of digital inputs that can auto-generate is hardware-dependent. Use GpioBindEvent() to generate these events. See example for <a href="#">GpioBindEvent()</a>
<b>EVDETECTCHANn</b>	N is from 0 to 3 where N is platform dependent and an event is generated when a preconfigured digital input transition occurs. The number of digital inputs that can auto-generate is hardware dependent. Use GpioAssignEvent() to generate these events.

## GpioSetFunc

### FUNCTION

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

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

The nFunction argument denotes the required functionality. Use only supported values from [Table 1](#).

The nSubFunc argument defines the configuration of the requested function, for DIGITAL\_IN this value is a bit field, for DIGITAL\_OUT this is a value.

### *GPIOSSETFUNC (nSigNum, nFunction, nSubFunc)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nSigNum</b>	byVal nSigNum AS INTEGER. The signal number as stated in the pinout table of the module.

<b><i>nFunction</i></b>	<b>byVal nFunction AS INTEGER.</b> Specifies the configuration of the SIO pin as follows: 1 = DIGITAL_IN 2 = DIGITAL_OUT 3 = ANALOG_IN (SIO 21 – Temperature Sensor and SIO20 – Trim Pot only)
<b><i>nSubFunc</i></b>	<b>byVal nSubFunc INTEGER.</b> Configures the pin with the following bit field values: If nFunction == DIGITAL_IN <ul style="list-style-type: none"> <li>2 or 4 – Use pull-up resistor</li> <li>16 – Wake up when signal is low</li> <li>32 – Wake up when signal is high</li> <li>48 – Wake up when signal changes</li> </ul> If nFunction == DIGITAL_OUT <b>Values:</b> <ul style="list-style-type: none"> <li>0 = Initial output to LOW</li> <li>1 = Initial output to HIGH</li> <li>2 = Pwm output</li> <li>3 = Freq output</li> </ul>

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

#### Example :: GpioSetFunc.sb

```
PRINT GpioSetFunc(15,1,2) //Digital In SIO 15, strong pull up resistor
PRINT GpioSetFunc(20,3,0) //Analog In SIO 20 (Trim Pot), default settings
PRINT GpioSetFunc(17,2,1) //SIO17 (LED0) digital out, initial output high
```

#### Expected Output:

```
000
```

GPIOSETFUNC is a Module function.

### GpioConfigPwm

#### FUNCTION

This routine configures the PWM (Pulse Width Modulation) of all output pins when they are set as a PWM output using GpioSetFunc() function described above.

**Note:** This is a 'sticky' configuration; calling it affects all PWM outputs already configured. We advise that you call this once at the beginning of your application and do not change it again within the application unless all PWM outputs are deconfigured and then re-enabled after this function is called.

The PWM output is generated using 16-bit hardware timers.

A PWM signal has a frequency and a duty cycle property, the frequency is set using this function and is defined by the `nMinFreqHz` parameter. The maximum PWM frequency value is 5000000Hz (5 MHz) for modules where the system clock has been set to 20 or 40 MHz and 1000000Hz (1 MHz) for modules configured with a System Clock of 4MHz. The `nMaxPeriodus` parameter is purely historical and serves no real function in the BT900. It should be set to the same value as the `nMinFreqHz` parameter.

The mark space ratio of the output waveform is controlled by the `GpioWrite` function.

The PWM output frequency is produced by dividing the System Clock Frequency, which can be obtained from AT I 2300, by the `nMinFreqHz` value. The result of this calculation is loaded into a 16 bit hardware register and represents the number of clock cycles that is required to obtain the the required output frequency. Take note of this register value as it's needed to verify if the required mark:space ratio is valid.

**Note:** If the calculation does not provide an integer result the register value obtained is rounded down, so does not provide an accurate frequency for the PWM output. This is more noticeable at higher frequencies.

The lowest PWM output frequency at the maximum system frequency of 40 MHz is approximately 611 Hz, i.e. 40 MHz / 65536.

On exit, the function returns with the actual frequency in the `nMinFreqHz` parameter.

**Note:** On the BT900 only **SIO 12, 13, and 17** may be used for PWM.

#### ***GPIOCONFIGPWM (nMinFreqHz, nMaxPeriodus)***

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nMinFreqHz</i></b>	<b>byRef nMinFreqHz AS INTEGER.</b> The nominal frequency of the waveform.
<b><i>nMaxPeriodus</i></b>	<b>byVal nMaxPeriodus INTEGER.</b> Set to same value as nMinFreqHz.

#### **Example :: pwm.sb**

```
dim retval
dim i
dim nFreq
dim nResolution
dim res[5] as integer

FUNCTION HandlerTimer1()
    dim TmpVal
    i=i+1
    if i==5 then
        i=0
    endif
    TmpVal = (res[i]*100/nFreq)
    PRINT "\nTimer event! PWM changed to "; TmpVal; "% duty cycle."
```

```
GpioWrite(13,res[i])
ENDFUNC 1

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

ONEVENT EVTMR1 CALL HandlerTimer1

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

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

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

WAITEVENT
```

#### Expected Output:

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

GPIOCONFIGPWM is a Module function.

## GpioRead

### FUNCTION

This routine reads the value from a SIO pin.

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

#### *GPIOREAD (nSigNum)*

<b>Returns</b>	INTEGER, the value from the signal. If the signal number is invalid, then it returns a value of 0. For digital pins, the value is 0 or 1. For ADC pins it is a value in the range 0 to M where M is the maximum value based on the bit resolution of the analogue to digital converter.
<b>Arguments:</b>	
<b><i>nSigNum</i></b>	<b>byVal nSigNum INTEGER.</b> The signal number as stated in the pinout table of the module.

Refer to the example for [GpioBindEvent](#).

GPIOREAD is a Module function.

**Note:** The raw output, when in ADC mode, is controlled by config register 2301. The ADC pins in the BT900 are fed into a 12-bit converter. The reference voltage used for the conversion is 3.3V. The BL600 has a 10-bit converter with a reference voltage of 1.2V. To ensure that both modules return the same raw value for the same input voltage the BT900 (by default) scales the raw output to match that of the BL600. For example, if 1.2 volts is applied to the pin, 1023 is the raw output. In this mode config register 2301 has a value of 0.

If config register 2301 is changed to 2, then the raw output is not scaled. Therefore, if 1.2V is applied to the pin, then the raw output is 1489.

The equation to convert from raw value to voltage differs depending on the value stored in config register 2301.

For the default case of 0 and to match the BL600 then use:  **$V_{in} = ((\text{Raw output} * 1.2) / 1023)V$**

If the configuration register is set to 2 then use:  **$V_{in} = ((\text{Raw output} * 3.3) / 4095)V$**

#### ADC Example:

```
//Example: adc.sb
//This example reads from the trim pot, for it to work, TrimPot on CON14 needs to be in the
ON position
#define GPIO_TRIM_POT      20

dim rc, adc

//Start timer to read trim pot
```



```
TimerStart(0,1000,1)

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

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

FUNCTION HandlerTimer0()
    //Read the ADC
    adc = GpioRead(GPIO_TRIM_POT)
    PRINT "\nTrim Pot Reading: ";adc
ENDFUNC 1

OnEvent EVTMR0 call HandlerTimer0

WAITEVENT
```

Expected output:

```
Trim Pot Reading: 1943
Trim Pot Reading: 1943
```

## GpioWrite

### FUNCTION

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

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

For modules that have been configured with a System Clock of 20 or 40 MHz, if the SIO pin has been configured as a FREQUENCY output then the nNewValue specifies the desired frequency in Hertz in the range 0 to 5000000. Setting a value of 0 makes the output a constant low value. Setting a value greater than 5000000 causes an error message to be sent.

For modules that have been configured with a System Clock of 4 MHz the nNewValue range is 0 to 1000000.

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

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

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

### *GPIOWRITE (nSigNum, nNewValue)*

Returns	
Arguments:	
<i>nSigNum</i>	<b>byVal nSigNum INTEGER.</b> The signal number as stated in the pinout table of the module.
<i>nNewValue</i>	<b>byVal nNewValue INTEGER.</b> The value to be written to the port. If the pin is configured as digital, then 0 clears the pin and a non-zero value sets it. If the pin is configured as a PWM then this value sets the duty cycle. If the pin is configured as a FREQUENCY then this value sets the frequency.

#### Example:

```

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

'//-----
'// For debugging
'// --- rc = result code
'// --- ln = line number
'//-----

Sub AssertRC(rc,ln)
    if rc!=0 then
        print "\nFail :";integer.h' rc;" at tag ";ln
    //else
        //print "\nOk: line ";ln
    endif
EndSub

rc=GpioSetFunc(17,2,1)
AssertRC(rc,16)

rc=GpioSetFunc(18,2,1)
AssertRC(rc,19)

```

```
function HandlerTmr0 ()
    i1=!i1
    GpioWrite(18,i1)
    AssertRC(rc,30)
endfunc 1

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

function HandlerUartRx()
endfunc 0

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

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

waitevent

print "\nExiting..."
```

## Expected Output:

```
Ensure the DIP switches on CON14 are set as follows:

      OFF    ON
LED0  [1  -->]
LED1  [2  -->]
TempS [3  x  ]
TrimPot [4  x  ]
x = doesn't matter

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

GPIOWRITE is a Module function.

## GpioBindEvent/GpioAssignEvent

### FUNCTION

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

When this function is called on the BT900, the SIO pin specified by *nSigNum* is set up as a digital input in the underlying firmware so *GpioSetFunc()* does **not** need to be called beforehand.

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

---

**Note:** In the BT900 module an SIO pin can only be bound to one event at a time.

---

### *GPIOBINDEVENT (nEventNum, nSigNum, nPolarity)*

### *GPIOASSIGNEVENT (nEventNum, nSigNum, nPolarity)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nEventNum</i></b>	<b>byVal <i>nEventNum</i> INTEGER.</b> The SIO event number (in the range of 0 - N) which results in the event EVGPIOCHANn being thrown to the <i>smart</i> BASIC runtime engine.
<b><i>nSigNum</i></b>	<b>byVal <i>nSigNum</i> INTEGER.</b> The signal number as stated in the pinout table of the module. On the BT900 this can only be SIO 0, 3, 5, 8, 13, 20 or 22. <b>Note:</b> SIO0 and SIO3 can only be used as bind or assign events if they are not being used as UART pins, i.e. that the <i>UartClose()</i> function has been called and the UART has not been reopened.

<b><i>nPolarity</i></b>	<b>byVal <i>nPolarity</i> INTEGER.</b> States the transition as follows: <ul style="list-style-type: none"> <li>▪ 0 - Low to high transition</li> <li>▪ 1 - High to low transition</li> <li>▪ 2 - Either a low to high or high to low transition</li> </ul>
-------------------------	--

#### Example :: GpioBindEvent.sb

```

dim rc

function HandlerBtn0()
    dim i : i = GpioRead(13)

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

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

function HandlerUartRx()
endfunc 0

rc= GpioBindEvent(0,13,2)    '//Bind event 0 to high or low transition on SI013 (button 1)
if rc==0 then
    onevent evgpiochan0 call HandlerBtn0 '//When event 0 happens, call Btn0Press
    print "\nSI013 - Button 0 is bound to event 0. Press button 0"
else
    print "\nGpioBindEvent Err: ";integer.h'rc
endif

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

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

```

```
endif
```

#### Expected Output:

```
SIO13 - Button 0 is bound to event 0. Press button 0

Press any key to exit
Button 0 Pressed
Button 0 Released
Button 0 Pressed
Button 0 Released

Event 0 unbound
Exiting...
00
```

GPIOBINDEVENT is a Module function.

### GpioUnbindEvent/GpioUnAssignEvent

#### FUNCTION

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

**GPIOUNBINDEVENT** (*nEventNum*)

**GPIOUNASSIGNEVENT** (*nEventNum*)

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

See example for [GpioBindEvent](#).

### 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, **if and only if**, the SIO19 input pin is held low.

Given that SIO19 is low, after erasing the file system, the module resets and reboots into command mode with the virtual serial port service enabled; the module advertises for a few seconds. See the [virtual serial port service section](#) for more details.

This facility allows the current \$autorun\$ application to be replaced with a new one.

**WARNING:**

**If this function is called from within \$autorun\$ and the SIO19 input is low, then it is erased and a fresh download of the application is required which can be facilitated over the air.**

**ERASEFILESYSTEM (nArg)**

<b>Returns</b>	INTEGER Indicates success of command: 0 Successful erasure. The module reboots. <>0 Failure.
<b>Exceptions</b>	<ul style="list-style-type: none"><li>Local Stack Frame Underflow</li><li>Local Stack Frame Overflow</li></ul>
<b>Arguments:</b>	
<b>nArg</b>	<b>byVal nArg AS INTEGER</b> This is for future use and MUST always be set to 1. Any other value results in a failure.

**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 erase file system because SIO19 is low"
ENDIF
```

**Expected Output:**

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

## BTC EXTENSIONS BUILT-IN ROUTINES

### Generic Access Profile Functions

This section describes routines related to the Generic Access Profile.

#### 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 *smart*BASIC application, contains **respType**, the type of inquiry response received. It is one of the following values:

0	Standard
1	With RSSI
2	Extended (contains EIR data)

#### Example:

```
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(adr$,ad$) != 0 then
        print "\nBluetooth: "; StrHexize$(ad$)

    if respType > 0 then
        print " ";rsi

    if respType == 2 then
        print "\n 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

//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
endifunc 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:

```
Bluetooth: 0C8BFD515094 -57
EIR: 0D094C4F4E444C31395458525931020A0A
- Tag 0x09: LONDL19TXRY1
- Tag 0x0A: 0A

Bluetooth: 94350AA99A3C -45
EIR:
1409446176696420446176697327732050686F6E65170305110A110C111211151116111F112
D112F110012321101050107
- Tag 0x09: David Davis's Phone
- Tag 0x03: 05110A110C111211151116111F112D112F1100123211

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

### *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 currently not supported and will be implemented in future releases of the firmware.

### **BTCINQUIRYCONFIG** (*nConfigID*,*nValue*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
<b>Arguments:</b>		
<b>nConfigID</b>	<b>byVal nConfigID AS INTEGER.</b> This identifies the value to update as follows:	
	0	Inquiry Type (0 for General Inquiry, 1 for Limited Inquiry)
	1	Inquiry Mode (0 for Standard, 1 for with RSSI, 2 for Extended)
	2	Max number of inquiry responses to receive (Range is from 0-255)

	3	Inquiry Tx Power (Range is from -70 to 20 dBm)
<i>nValue</i>	<b>byVal nValue AS INTEGER.</b> The new value to set for the parameter identified by configID.	

See example for [EvInqResp](#).

## BtcInquiryStart

### FUNCTION

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

#### BTCINQUIRYSTART (*nTimeout*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<i>nTimeout</i>	<b>byVal nTimeout AS INTEGER.</b>
This is how long in seconds the inquiry lasts. If the timer times out then the event EVBTC_INQUIRY_TIMEOUT is thrown to the <i>smart</i> BASIC application.	

See example for [EvInqResp](#).

## BtcInquiryCancel

### FUNCTION

Cancel an ongoing inquiry.

#### BTCINQUIRYCANCEL()

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	None

### Example:

```

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
```

## 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 for the inquiry scan.

This function sets 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 currently supported and will be implemented in future releases of the firmware.

---

### BTCDISCOVERYCONFIG (nConfigID,nValue)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
----------------	---

---

**Arguments:**

---

<b>nConfigID</b>	<b>byVal nConfigID AS INTEGER.</b> This identifies the value to update as follows:	
	0	Discoverability type: ▪ 0 = General (default) ▪ 1 = Limited
	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)
<b>Note:</b> For all other configID values, the function returns an error.		
<b>nValue</b>	<b>byVal nValue AS INTEGER.</b> The new value to set for the parameter identified by configID.	

**Examples:**

```

'//-----
'// 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
```

## BtcSetDiscoverable

### FUNCTION

This function sets the module discoverable for the time specified time or not discoverable. It sets the module for the discoverability type specified by [BtcDiscoveryConfig\(\)](#).

#### **BTCSETDISCOVERABLE** (*nEnable*, *nTimeout*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nEnable</i></b>	<b>byVal <i>nEnable</i> AS INTEGER</b> 0 – Not discoverable 1 – Discoverable
<b><i>nTimeout</i></b>	<b>byVal <i>nTimeout</i> AS INTEGER</b> The length of time in seconds that the module is discoverable. Default: 60 seconds. If <i>nEnable</i> is set to zero (0), this parameter is ignored.

### Example:

```
dim rc, n$
n$ = "My BT900"

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...
```

## 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)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nEnable</i></b>	<b>byVal <i>nEnable</i> AS INTEGER</b> 0 – Not connectable 1 – Connectable

#### Example:

```
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
```

## 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 BT900 has to also be set as connectable in order to receive incoming pairing requests.

### *BTCSETPAIRABLE(nEnable)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nEnable</i></b>	<b>byVal <i>nEnable</i> AS INTEGER</b> 0 – Not pairable 1 – Pairable

### Example:

```
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](#).

## 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 *smart*BASIC 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)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>addr\$</i></b>	<b>byREF periphAddr\$ AS STRING</b> The address of the advertiser is returned in this string. It is a 6-byte string.



<b>inqData\$</b>	<b>byREF advData\$ AS STRING</b> The data payload is returned in this string.
<b>nDiscarded</b>	<b>byREF nDiscarded AS INTEGER</b> On return, this parameter is updated with the number of adverts that were discarded because there was no space in the internal queue.
<b>nRssi</b>	<b>byREF nRssi AS INTEGER</b> On return, this parameter is updated with the RSSI as reported by the stack for that advert. <b>Note:</b> This is not a value that is sent by the peripheral but rather a value that is calculated by the receiver in this module.

See example for [EvInqResp](#).

## BtInquiryGetReportFull

### FUNCTION

This function is used by the *smart*BASIC application to extract the full inquiry report from the queue for further processing in the handler for the EVINQRESP event.

**BTCINQUIRYGETREPORTFULL(addr\$, nPScanRepMode, nPScanPeriodMode, nPScanMode, nCOD, nClockOffset, inqData\$, nDiscarded, nRssi)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>addr\$</b>	<b>byREF periphAddr\$ AS STRING</b> The address of the advertiser is returned in this string. It is a 6-byte string.
<b>nPScanRepMode</b>	<b>byREF nPScanRepMode AS INTEGER</b> Part of the supported Page Scan Modes that the remote device supports.
<b>nPScanPeriodMode</b>	<b>byREF nPScanPeriodMode AS INTEGER</b> Current setting of this parameter.
<b>nPScanMode</b>	<b>byREF nPScanMode AS INTEGER</b> The other part of the supported Page Scan Modes that the remote device supports.
<b>nCOD</b>	<b>byREF nCOD AS INTEGER</b> Class of device of the remote device.
<b>nClockOffset</b>	<b>byREF nClockOffset AS INTEGER</b> Bits 16 to 2 of the difference between the master and slave device clocks, mapped to bits 14 to 0 of this parameter (i.e., computed from (<clock_slave – clock_master> ShiftRight 2). Bit 15 (MSB) is the Clock_Offset_Valid flag which is 1 if the offset value is valid.
<b>inqData\$</b>	<b>byREF advData\$ AS STRING</b> The data payload is returned in this string.
<b>nDiscarded</b>	<b>byREF nDiscarded AS INTEGER</b> On return, this parameter is updated with the number of adverts that were discarded because there was no space in the internal queue.
<b>nRssi</b>	<b>byREF nRssi AS INTEGER</b> On return, this parameter is updated with the RSSI as reported by the stack for that advert. <b>Note:</b> This is not a value that is sent by the peripheral but rather a value that is calculated by the receiver in this module.

See example for [EvInqResp](#).

## BtcGetClassOfDevice

### FUNCTION

Get the class of device as seen by other devices.

#### *BTCGETCLASSOFDEVICE(nClassOfDevice)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nClassOfDevice</b>	<b>byREF nClassOfDevice AS INTEGER</b> On return this integer contains the class of device in the baseband.

#### Example:

```
dim rc, class
rc= BtcGetClassOfDevice(class)
print "\n";integer.h' class
```

#### Expected Output:

```
00C0FFEE
```

## BtcSetClassOfDevice

### FUNCTION

Set the class of device for this module. This class is visible to other Bluetooth Classic devices doing an inquiry if they discover the module.

#### *BTCSETCLASSOFDEVICE(nClassOfDevice)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nClassOfDevice</b>	<b>byVAL name\$ AS INTEGER</b> The new class of device to set. The value accepted masks the lower three bytes.

#### Example:

```
dim rc, class
class = 0xC0FFEE
rc= BtcSetClassOfDevice(class)
rc= BtcGetClassOfDevice(class)
print "\n ";integer.h' class
```

#### Expected Output:

```
00C0FFEE
```

## 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\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nIndex</b>	<b>byVAL nIndex AS INTEGER.</b> 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 results in an error
<b>data\$</b>	<b>byREF data\$ AS STRING</b> On exit this contains the report containing concatenated EIR elements
<b>EIRtag</b>	<b>byREF EIRtag AS INTEGER</b> On exit this contains the tag value
<b>EIRval\$</b>	<b>byREF EIRval\$ AS STRING</b> On exit this contains the data from the nth EIR element if it exists.
<b>Note:</b>	Only the data portion of the EIR element is returned. The Tag is seperately provided in the EIRtag argument and the length of the data is strlen(EIRval\$).

### Example:

```

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(adr$,ad$) != 0 then
        print "\nBluetooth Address: "; StrHexize$(ad$)

    if respType > 0 then

```

```
print " ",rsi

if respType == 2 then
    print "\n 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

        //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
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:

```
Bluetooth: 0C8BFD515094 -57
  EIR: 0D094C4F4E444C31395458525931020A0A
    - Tag 0x09: LONDL19TXRY1
    - Tag 0x0A: 0A

Bluetooth: 94350AA99A3C -45
  EIR:
1409446176696420446176697327732050686F6E65170305110A110C111211151116111F112D112F1100123211
01050107
    - Tag 0x09: David Davis's Phone
    - Tag 0x03: 05110A110C111211151116111F112D112F1100123211

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

## 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\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>data\$</b>	<i>byREF data\$ AS STRING</i> On exit this contains the report containing concatenated EIR elements
<b>EIRtag</b>	<i>byREF EIRtag AS INTEGER</i> The tag to look for. Only the first instance can be extracted. If multiple instances are suspected, then use BtcGetEIRbyIndex()
<b>EIRval\$</b>	<i>byREF EIRval\$ AS STRING</i> 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\$).

**Example:**

```
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
            //address is not as before so display the data
            adr$=ad$
            print "\nINQ: ";strhexize$(ad$); " ";rsi

            if cType == 2 then
                // If its extended print the raw EIR data, then the complete local name
                print "\n EIR RAW: ";strhexize$(dta$)
                print "\n EIR:"

                tag = 0x09 //complete local name
                rc=BtcGetEIRbyTag(dta$,tag,ad$)

                print "Complete Local Name: ";ad$
                print "Hex: ";strhexize$(ad$)
            endif
        endif
        //get the next advert in the cache
        rc = BtcInquiryGetReport(ad$,dta$,ndx,rsi)
    endwhile
endfunc 1

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:1409736D6172745A2D303031364134303933413839
EIR:Complete Local Name: smartZ-0016A4093A89Hex:
736D6172745A2D303031364134303933413839
INQ:C4D98776AE3E -65
EIR RAW:0E094C4F4E444C4851535656575A31020A04
EIR:Complete Local Name: LONDLHQSVVWZ1Hex
```

## BtcGetFriendlyName

### FUNCTION

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

### BTCGETFRIENDLYNAME (name\$)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>name\$</b>	<b>byREF name\$ AS STRING</b> On return this string contains the device name

**Example:**

```
dim rc, name$
rc=BtcGetFriendlyName(name$)
print "\n"; name$
```

**Expected Output:**

```
Laird BT900
```

## BtcGetRemoteFriendlyName

### FUNCTION

Function to recall the address and friendly name of the result of a query signified by the event, `EVBTC_REMOTENAME_RECEIVED`.

**BTCGETREMOTEFRIENDLYNAME**(*address\$, name\$, nStatus*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>address\$</b>	<b>byREF address\$ AS STRING</b> Address of the remote device that the friendly name relates to. Returned from the query.
<b>name\$</b>	<b>byREF name\$ AS STRING</b> Friendly name of the remote device, returned from the query.
<b>nStatus</b>	<b>byREF nStatus AS INTEGER</b> Status code of the query, the name string is only valid on a success status of 0.

## BtcQueryRemoteFriendlyName

### FUNCTION

Query the friendly name from a remote device specified in the address.

**BTCQUERYREMOTEFRIENDLYNAME**(*address\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>address\$</b>	<b>byVAL address\$ AS STRING</b> Address of the remote device.

## BtcSetFriendlyName

### 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.

**BTCSETFRIENDLYNAME** (*name\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
----------------	---



**Arguments:**

<b><i>name\$</i></b>	<b><i>byREF name\$ AS STRING</i></b> The new name to set. The maximum allowed length is 31 characters.
----------------------	---

**Example:**

```
dim rc, name$
name$ = "My BT900"
rc=BtcSetFriendlyName(name$)
rc=BtcGetFriendlyName(name$)
print "\n"; name$
```

**Expected Output:**

```
My BT900
```

## BtcSniffEnable

### FUNCTION

This function initiates sniff negotiations with a remote device. There must be an open connection with the device specified. The values taken in this function are in the number of baseband slots (0.625ms).

***BTCNIFFENABLE(strBDAAddr\$, attempt, timeout, minPeriod, maxPeriod)***

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>strBDAAddr\$</i></b>	<b><i>byREF strBDAAddr\$ AS STRING</i></b> This sets the address of the remote device for which sniff should be negotiated.
<b><i>attempt</i></b>	<b><i>byVAL attempt AS INTEGER</i></b> The amount of time for each sniff attempt.
<b><i>timeout</i></b>	<b><i>byVAL timeout AS INTEGER</i></b> The amount of time for a sniff timeout.
<b><i>minPeriod</i></b>	<b><i>byVAL minPeriod AS INTEGER</i></b> Minimum time between each sniff period.
<b><i>maxPeriod</i></b>	<b><i>byVAL maxPeriod AS INTEGER</i></b> Maximum time between each sniff period.

**Example:**

```
dim rc, i

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

```
//Array with handles for spp connections
dim hSpp

#define LINKMODE_ACTIVE (0)
#define LINKMODE_SNIFF (2)

//Handler function called when sniff mode has been enabled
function HandlerSniff(nStatus, nMode) as integer
    dim bdaddr$, link_mode, interval

    if((nStatus==0) && (nMode == LINKMODE_SNIFF))then
        // Successful mode change - Query the link
        rc = BtcQuerySniffChange(bdaddr$, link_mode, interval)
        print "\nSniff mode enabled: "
        print strhexize$(bdaddr$); " Interval "; interval
    endif
endfunc 0

//Handler function called when SPP is connected
function HandlerSppConn(portHndl, result) as integer
    hSpp = portHndl
    print "\n --- Connect : ",hSpp, StrHexize$(mac$)
    print "\nResult: ",integer.h' result
endfunc 0

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

//Make spp connection
rc=BtcSppConnect(mac$)
print "\nConnecting to device ";StrHexize$(mac$)

onevent EvSppConn call HandlerSppConn
waitevent

//Enable sniff parameters with the active Bluetooth link
//Values are defined in timeslots
print "\nEnabling sniff mode with ";StrHexize$(mac$)
rc = BtcSniffEnable(mac$, 80, 128, 544, 1600)
```

```
//Wait for a confirmation from the remote device that parameters were successfully negotiated
onevent EVBTC_SNIFF_CHANGE call HandlerSniff
waitevent
```

### Expected Output:

```
Connecting to device 0016A4093ABF
--- Connect :      130049 0016A4093ABF
Result:      00000000
Enabling sniff mode with 0016A4093ABF
Sniff mode enabled: 0016A4093ABF Interval 1600
```

## BtcSniffDisable

### FUNCTION

This function disables sniff mode with a remote device. There must be an open connection with the device specified.

#### *BTCSNIFFDISABLE(strBDAddr\$)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>strBDAddr\$</i></b>	<b>byREF    <i>strBDAddr\$</i>    AS STRING</b> Bluetooth address of the device with which to disable sniff mode.

### Example:

```
dim rc, i

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

//Array with handles for spp connections
dim hSpp

#define LINKMODE_ACTIVE (0)
#define LINKMODE_SNIFF (2)

//Handler function when the link mode is changed
function HandlerSniff(nStatus, nMode) as integer
    dim bdaddr$, link_mode, interval
```

```
if(nStatus==0) then
    // Successful mode change - Query the link
    rc = BtcQuerySniffChange(bdaddr$, link_mode, interval)

    if(nMode == LINKMODE_SNIFF) then
        print "\nSniff mode enabled: "
        print strhexize$(bdaddr$); " Interval "; interval
    elseif(nMode == LINKMODE_ACTIVE) then
        print "\nSniff mode disabled: "
        print strhexize$(bdaddr$);
    endif
endif
endfunc 0

//Handler function called when SPP is connected
function HandlerSppConn(portHndl, result) as integer
    hSpp = portHndl
    print "\n --- Connect : ",hSpp, StrHexize$(mac$)
    print "\nResult: ",integer.h' result
endfunc 0

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

//Make spp connection
rc=BtcSppConnect(mac$)
print "\nConnecting to device ";StrHexize$(mac$)

onevent EvSppConn call HandlerSppConn
waitevent

//Enable sniff parameters with the active Bluetooth link
//Values are defined in timeslots
print "\nEnabling sniff mode with ";StrHexize$(mac$)
rc = BtcSniffEnable(mac$, 80, 128, 544, 1600)

//Wait for a confirmation from the remote device that parameters were successfully negotiated
onevent EVBTC_SNIFF_CHANGE call HandlerSniff
waitevent
```

```
//Disable sniff mode
rc = BtcSniffDisable(mac$)

//Wait for a confirmation from the remote device has disable sniff mode
Waitevent
```

#### Expected Output:

```
Connecting to device 0016A4093ABF
--- Connect :      130049 0016A4093ABF
Result:      00000000
Enabling sniff mode with 0016A4093ABF
Sniff mode enabled: 0016A4093ABF Interval 1600
Sniff mode disabled: 0016A4093ABF
```

### BtcQuerySniffSubrating

#### FUNCTION

This function initiates sniff subrate negotiations with a remote device. There must be an active connection with the device specified.

**BTCQUERYSNIFFSUBRATING**(*strBDAAddr\$, maxTxLatency, maxRxLatency, minRemoteTimeout, minLocalTimeout*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>strBDAAddr\$</i></b>	<b>byREF <i>strBDAAddr\$</i> AS STRING</b> Returns the address of the last device to have negotiated sniff subrating.
<b><i>maxTxLatency</i></b>	<b>byREF <i>maxTxLatency</i> AS INTEGER</b> Returns the maximum transmit latency.
<b><i>maxRxLatency</i></b>	<b>byREF <i>maxRxLatency</i> AS INTEGER</b> Returns the maximum receive latency.
<b><i>minRemoteTimeout</i></b>	<b>byREF <i>minRemoteTimeout</i> AS INTEGER</b> Returns the minimum remote timeout.
<b><i>minLocalTimeout</i></b>	<b>byREF <i>minLocalTimeout</i> AS INTEGER</b> Returns the minimum local timeout.

#### Example:

```
dim rc, i

'//BT address of device to connect to. You will have to change this
dim mac$
```

```
mac$ = "\00\16\A4\09\3A\BF"

//Array with handles for spp connections
dim hSpp

#define LINKMODE_ACTIVE (0)
#define LINKMODE_SNIFF (2)

//Handler function called when the link mode is changed
function HandlerSniff(nStatus, nMode) as integer
    dim bdaddr$, link_mode, interval

    if(nStatus==0) then
        // Successful mode change - Query the link
        rc = BtcQuerySniffChange(bdaddr$, link_mode, interval)

        if(nMode == LINKMODE_SNIFF) then
            print "\nSniff mode enabled: "
            print strhexize$(bdaddr$); " Interval "; interval
        elseif(nMode == LINKMODE_ACTIVE) then
            print "\nSniff mode disabled: "
            print strhexize$(bdaddr$);
        endif
    endif
endfunc 0

//Handler function called when sniff subrating parameters are changed
function HandlerSubrate(status)
    dim bdaddr$, bdaddrhex$
    dim maxTxLatency, maxRxLatency, minRemoteTimeout, minLocalTimeout
    dim rc

    // Successful Sniff subrate mode change - Query the link
    rc = BtcQuerySniffSubrating(bdaddr$, maxTxLatency, maxRxLatency, minRemoteTimeout,
minLocalTimeout)

    //If the address is zero then subrating has been disabled due to a closed connection
    bdaddrhex$ = StrHexize$(bdaddr$)
    if(strcmp(bdaddrhex$, "000000000000") == 0) then
        exitfunc 1
    end if
endfunc 0
```

```
else
    print "\nSubrating changed: "; StrHexize$(bdaddr$); " - "
    print maxTxLatency; " "; maxRxLatency; " "; minRemoteTimeout; " "; minLocalTimeout
endif
endfunc 0

//Handler function called when SPP is connected
function HandlerSppConn(portHndl, result) as integer
    hSpp = portHndl
    print "\n --- Connect : ",hSpp, StrHexize$(mac$)
    print "\nResult: ",integer.h' result
endfunc 0

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

//Make spp connection
rc=BtcSppConnect(mac$)
print "\nConnecting to device ";StrHexize$(mac$)

onevent EvSppConn call HandlerSppConn
waitevent

//Enable sniff parameters with the active Bluetooth link
//Values are defined in timeslots
print "\nEnabling sniff mode with ";StrHexize$(mac$)
rc = BtcSniffEnable(mac$, 80, 128, 544, 1600)

//Wait for a confirmation from the remote device that parameters were successfully negotiated
onevent EVBTC_SNIFF_CHANGE call HandlerSniff
waitevent

//Enable sniff subrating parameters
rc = BtcSniffSubratingEnable(mac$, 5000, 2000, 2000)

//Wait for a confirmation from the remote device that parameters were successfully negotiated
onevent EVBTC_SNIFF_SUBRATING call HandlerSubrate
waitevent
```

## Expected Output:

```
Connecting to device 0016A4093ABF
--- Connect :      130049 0016A4093ABF
Result:      00000000
Enabling sniff mode with 0016A4093ABF
Sniff mode enabled: 0016A4093ABF Interval 1600
Subrating changed: 0016A4093ABF - 4800 1600 2000 2000
```

## BtcQueryModeChange

### FUNCTION

This function requests the address of the last device to have changed mode of operation, which can be sniff, park, hold or active. The function also returns the interval negotiated if applicable. This is called after a EVBTC\_MODE\_CHANGE event.

### BTCQUERYMODECHANGE(strBAddr\$, mode, interval)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>strBAddr\$</b>	<b>byREF strBAddr\$ AS STRING</b> Returns the address of the last device to change mode.
<b>mode</b>	<b>byREF mode AS INTEGER</b> Returns the mode entered by the device returned in the address.
<b>interval</b>	<b>byREF interval AS INTEGER</b> Returns the interval negotiated by the device returned in the address.

### Example:

```
dim rc, i

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

//Array with handles for spp connections
dim hSpp

#define LINKMODE_ACTIVE (0)
#define LINKMODE_SNIFF (2)

//Handler function called when the link mode is changed
```



```
function HandlerSniff(nStatus, nMode) as integer
    dim bdaddr$, link_mode, interval

    if(nStatus==0) then
        // Successful mode change - Query the link
        rc = BtcQuerySniffChange(bdaddr$, link_mode, interval)

        if(nMode == LINKMODE_SNIFF) then
            print "\nSniff mode enabled: "
            print strhexize$(bdaddr$); " Interval "; interval
        elseif(nMode == LINKMODE_ACTIVE) then
            print "\nSniff mode disabled: "
            print strhexize$(bdaddr$);
        endif
    endif
endfunc 0

//Handler function called when SPP is connected
function HandlerSppConn(portHndl, result) as integer
    hSpp = portHndl
    print "\n --- Connect : ",hSpp, StrHexize$(mac$)
    print "\nResult: ",integer.h' result
endfunc 0

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

//Make spp connection
rc=BtcSppConnect(mac$)
print "\nConnecting to device ";StrHexize$(mac$)

onevent EvSppConn call HandlerSppConn
waitevent

//Enable sniff parameters with the active Bluetooth link
//Values are defined in timeslots
print "\nEnabling sniff mode with ";StrHexize$(mac$)
rc = BtcSniffEnable(mac$, 80, 128, 544, 1600)
```

```
//Wait for a confirmation from the remote device that parameters were successfully negotiated
onevent EVBTC_SNIFF_CHANGE call HandlerSniff
waitevent
```

#### Expected Output:

```
Connecting to device 0016A4093ABF
--- Connect :      130049 0016A4093ABF
Result:      00000000
Enabling sniff mode with 0016A4093ABF
Sniff mode enabled: 0016A4093ABF Interval 1600
Sniff mode disabled: 0016A4093ABF
```

### BtcSniffSubratingEnable

#### FUNCTION

This function initiates sniff subrate negotiations with a remote device. There must be an active connection with the device specified. The values taken in this function are in the number of baseband slots (0.625ms).

**BTCSNIFFSUBRATINGENABLE(*strBDAddr*\$, *maxLatency*, *minRemoteTimeout*, *minLocalTimeout*)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>strBDAddr</i>\$</b>	<b>byREF <i>strBDAddr</i>\$ AS STRING</b> This sets the address of the remote device for which sniff subrating should be negotiated.
<b><i>maxLatency</i></b>	<b>byVAL <i>maxLatency</i> AS INTEGER</b> Sets the maximum sniff subrate that the remote device may use.
<b><i>minRemoteTimeout</i></b>	<b>byVAL <i>minRemoteTimeout</i> AS INTEGER</b> Sets the minimum base sniff subrate timeout that the remote device may use.
<b><i>minLocalTimeout</i></b>	<b>byVAL <i>minLocalTimeout</i> AS INTEGER</b> Sets the minimum base sniff subrate timeout that the local device may use.

#### Example:

```
dim rc, i

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

//Array with handles for spp connections
dim hSpp
```

```
#define LINKMODE_ACTIVE (0)
#define LINKMODE_SNIFF (2)

//Handler function called when the link mode is changed
function HandlerSniff(nStatus, nMode) as integer
    dim bdaddr$, link_mode, interval

    if(nStatus==0) then
        // Successful mode change - Query the link
        rc = BtcQuerySniffChange(bdaddr$, link_mode, interval)

        if(nMode == LINKMODE_SNIFF) then
            print "\nSniff mode enabled: "
            print strhexize$(bdaddr$); " Interval "; interval
        elseif(nMode == LINKMODE_ACTIVE) then
            print "\nSniff mode disabled: "
            print strhexize$(bdaddr$);
        endif
    endif
endfunc 0

//Handler function called when sniff subrating parameters are changed
function HandlerSubrate(status)
    dim bdaddr$
    dim maxTxLatency, maxRxLatency, minRemoteTimeout, minLocalTimeout
    dim rc

    // Successful Sniff subrate mode change - Query the link
    rc = BtcQuerySniffSubrating(bdaddr$, maxTxLatency, maxRxLatency, minRemoteTimeout,
minLocalTimeout)

    print "\nSubrating changed: "; StrHexize$(bdaddr$); " - "
    print maxTxLatency; " "; maxRxLatency; " "; minRemoteTimeout; " "; minLocalTimeout
endfunc 0

//Handler function called when SPP is connected
function HandlerSppConn(portHndl, result) as integer
    hSpp = portHndl
    print "\n --- Connect : ",hSpp, StrHexize$(mac$)
```

```
print "\nResult: ",integer.h' result
endfunc 0

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

//Make spp connection
rc=BtcSppConnect(mac$)
print "\nConnecting to device ";StrHexize$(mac$)

onevent EvSppConn call HandlerSppConn
waitevent

//Enable sniff parameters with the active Bluetooth link
//Values are defined in timeslots
print "\nEnabling sniff mode with ";StrHexize$(mac$)
rc = BtcSniffEnable(mac$, 80, 128, 544, 1600)

//Wait for a confirmation from the remote device that parameters were successfully negotiated
onevent EVBTC_SNIFF_CHANGE call HandlerSniff
waitevent

//Enable sniff subrating parameters
rc = BtcSniffSubratingEnable(mac$, 5000, 2000, 2000)

//Wait for a confirmation from the remote device that parameters were successfully negotiated
onevent EVBTC_SNIFF_SUBRATING call HandlerSubrate
waitevent
```

#### Expected Output:

```
Connecting to device 0016A4093ABF
--- Connect :      130049 0016A4093ABF
Result:      00000000
Enabling sniff mode with 0016A4093ABF
Sniff mode enabled: 0016A4093ABF Interval 1600
Subrating changed: 0016A4093ABF - 4800 1600 2000 2000
```

## Human Interface Device

### Events and Messages

#### *EVHIDCONN*

This event is thrown when a new HID connection has been established or an error has occurred. The message is passed to a handler, which should be registered in the *smart*BASIC 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:

HID_CONNECTION_TIMEOUT	0x01
HID_CONNECTION_REFUSED	0x02
HID_UNKNOWN_ERROR	0x03

#### Example:

```

dim rc

'//Dummy BT address
dim mac$
mac$ = "\00\11\22\33\44\55"

\
//=====
// Called after a connection attempt
//=====

function HandlerHIDConn(portHndl, result) as integer
    if result == 0 then
        print "\n-- Connection Successful"
    elseif result == 1 then
        print "\n-- Connection Timeout"
    elseif result == 2 then
        print "\n-- Connection Refused"
    else
        print "\n-- Unknown Error"
    endif
endfunc 1

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

ONEVENT EvHIDConn      CALL HandlerHIDConn

```

```
'// We must open HID device or HID host before initiating connection
rc = BtcHIDHostOpen()

'//make hid connection
rc=BtcHIDConnect(mac$)

print "\nConnecting to device ";StrHexize$(mac$)

waitevent
```

**Expected Output:**

```
Connecting to device 001122334455
-- Connection Timeout
```

### *EVHIDDISCON*

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

**Example:**

```
dim rc

'//BT address of device to connect to. You will have to change this
dim mac$
mac$ = "\00\16\A4\09\3A\64"

function HandlerHIDConn(portHndl, result) as integer
    print "\n --- Connect : ";integer.h' result

    // Disconnect immediately for the purpose of this demo
    rc = BtcHIDDisconnect(portHndl)
    if rc==0 then
        print "\nDisconnecting ..."
    else
        print "\nError:", integer.h'rc
    endif
endfunc 1

function HandlerHidDiscon(portHndl) as integer
```

```

print "\n --- Disconnected"

endfunc 1

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

ONEVENT EvHidConn          CALL HandlerHidConn
OnEvent EvHidDiscon        call HandlerHidDiscon

'// We must open HID device or HID host before initiating connection
rc = BtcHidHostOpen()
'//make hid connection
rc=BtcHIDConnect(mac$)

print "\nConnecting to device ";StrHexize$(mac$)

waitevent

```

#### Expected Output:

```

Connecting to device 0016A4093A64
--- Connect : 00000000
Disconnecting ...
--- Disconnected

```

### EVHIDCONTROL

This event is thrown when the local HID Host or Device receives a Control event. The message contains **nHandle**, the handle of the connection and **nControl**, the event received out of the following:

HID_CONTROL_NOP	0x00
HID_CONTROL_HARDRESET	0x01
HID_CONTROL_SOFTRESET	0x02
HID_CONTROL_SUSPEND	0x03
HID_CONTROL_EXITSUSPEND	0x04
HID_CONTROL_VCABLEUNPLUG	0x05

On receiving a HID\_CONTROL\_VCABLEUNPLUG control event, the connection is automatically disconnected.

### EVHIDTXEMPTY

This event is generated when the last report has been sent to the baseband from a call to BtcHIDWrite() with the specified **nHandle**, use this event to trigger sending more reports.

## *EVBTCHID\_DATA\_RECEIVED*

This event is thrown when data is received via the Human Interface Device. Usage is as shown in the example given for [BtcHIDRead\(\)](#).

### Example :: BtcHIDWrite.sb

```
dim rc

function HandlerHIDConn(portHndl, result) as integer
    dim s$
    print "\n --- Connect : ";integer.h' result

endfunc 1

function HandlerHidData() as integer
    dim hPort, report$

    '//read and print out a report
    rc = BtcHIDRead(hPort, report$)

    print"\nHandle: ";hPort; " Data: ";StrHexize$(report$)

endfunc 1

ONEVENT EvHidConn          CALL HandlerHidConn
ONEVENT EvBtc_HID_data_received  CALL HandlerHidData

'// We must open HID device or HID host before initiating connection
rc = BtcHidHostOpen()
'// Set device to be connectable
rc = BtcSetConnectable(1)

print "\nWaiting for a HID device to connect and write to us"

waitevent
```

### Expected Output:

```
Waiting for a HID device to connect and write to us
--- Connect : 00000000
Handle: 130050 Data: 0000040000000000
```



## BtcHIDDeviceOpen

### FUNCTION

Open a local HID device, this function also registers the SDP records with default settings tunable with BtcHIDConfig and the descriptor provided.

**Note:** A descriptor file must exist in the BT900 module before attempting to call this function, otherwise it fails with error code 0x1806: FSA\_OPEN\_FAIL

### BTCHIDDEVICEOPEN (*descriptor\$, nFeatures, nSubclass, name\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
<b>Arguments</b>		
<b>descriptor\$</b>	<i>byRef</i>	<b>descriptor\$ AS STRING</b> The HID report descriptor string that specifies how data is interpreted
<b>nFeatures</b>	<i>byVal</i>	<b>nFeatures AS INTEGER</b> The device features flags that are entered into the SDP table. This value is a bit field:
	0	HID_VIRTUAL_CABLE_BIT
	1	HID_RECONNECT_INITIATE_BIT
	2	HID_SDP_DISABLE_BIT
	3	HID_BATTERY_POWER_BIT
	4	HID_REMOTE_WAKE_BIT
	5	HID_NORMALLY_CONNECTABLE_BIT
<b>nSubclass</b>	6	HID_BOOT_DEVICE_BIT
	<i>byVal</i>	<b>nSubclass AS INTEGER</b> The device subclass that is entered into the SDP, if the device is not a composite device this value should be the last 8 bits of the Class of Device.
<b>nVersion</b>	<i>byVal</i>	<b>nVersion AS INTEGER</b> The device release number which is included in the HID SDP record.
<b>name\$</b>	<i>byRef</i>	<b>name\$ AS STRING</b> The name of the service to be entered into the SDP table.

### Example :: BtcHIDDeviceOpen.sb

```
dim rc, devName$

// Device settings
#define DEVICE_VID_SRC    0x2
#define DEVICE_VID        0x0077
#define DEVICE_PID        0x1234
#define DEVICE_VERSION    0x1

// Device class and subclass
#define DEVICE_CLASS_OF_DEVICE 0x2540
#define DEVICE_SUBCLASS      0x40

// HID descriptor
```

```
#define DEVICE_DESCRIPTOR
"\05\01\09\06\A1\01\05\07\19\E0\29\E7\15\00\25\01\75\01\95\08\81\02\95\01\75\08\81\03\95\05\7
5\01\05\08\19\01\29\05\91\02\95\01\75\03\91\03\95\06\75\08\15\00\25\65\05\07\19\00\29\65\81\0
0\C0"

// Bit field for device flags
#define HID_VIRTUAL_CABLE_BIT (0x00000001)
#define HID_RECONNECT_INITIATE_BIT (0x00000002)
#define HID_SDP_DISABLE_BIT (0x00000004)
#define HID_BATTERY_POWER_BIT (0x00000008)
#define HID_REMOTE_WAKE_BIT (0x00000010)
#define HID_NORMALLY_CONNECTABLE_BIT (0x00000020)
#define HID_BOOT_DEVICE_BIT (0x00000040)

devName$ = "BT900 Keyboard"

rc = BtcHidDeviceOpen(DEVICE_DESCRIPTOR, HID_NORMALLY_CONNECTABLE_BIT |
HID_BATTERY_POWER_BIT, DEVICE_SUBCLASS, DEVICE_VERSION, devName$)

if rc == 0 then
    print "\nHID Device Opened"
else
    print "\nError: ";rc
endif

rc = BtcHidClose()
```

#### Expected Output:

```
HID Device Opened
```

## BtcHIDHostOpen

### FUNCTION

Opens a local HID Host. This enables accepting connections from devices or making connections to devices.

### BTCHIDHOSTOPEN()

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	None

#### Examples:

```
dim rc
```

```
rc = BtcHIDHostOpen()  
if rc == 0 then  
    print "\nBtc HID Host opened"  
else  
    print "\nError: ";rc  
endif  
  
rc=BtcHidClose()
```

#### Expected Output:

```
Btc HID Host opened
```

## BtcHIDClose

### FUNCTION

Closes the local HID Host or HID Device, this function disconnects all existing connections and flushes any reports with a read pending.

#### BTCHIDCLOSE()

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	None

BTCHIDCLOSE is a built-in function.

## BtcHIDConnect

### FUNCTION

Connects to a device or host at the specified Bluetooth address. The type of connection depends on the local mode, if a local Host is opened the connect command connects to a remote Device.

#### BTCHIDCONNECT(*bdaddr*\$)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>bdaddr</i>\$</b>	<b>byRef <i>bdaddr</i>\$ AS STRING</b> The Bluetooth address of the remote device or host to connect too.

#### Example:

```
dim rc  
  
'//BT address of device to connect to. You will have to change this
```

```
dim mac$
mac$ = "\00\16\A4\09\3A\64"

//=====
// Called after a connection attempt
//=====

function HandlerHIDConn(portHndl, result) as integer
    print "\n --- Connect : ";integer.h' result
endfunc 1

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

ONEVENT EvHIDConn      CALL HandlerHidConn

'// We must open HID device or HID host before initiating connection
rc = BtcHIDHostOpen()

'//make hid connection
rc=BtcHIDConnect(mac$)

print "\nConnecting to device ";StrHexize$(mac$)

waitevent
```

#### Expected Output:

```
Connecting to device 0016A4093A64
--- Connect : 00000000
```

## BtcHIDDisconnect

### FUNCTION

Request a disconnect from a remote HID Host or Device. If there are reports waiting to be read, the disconnect event is not thrown until the queue is empty, despite the host or device being disconnected.

#### *BTCHIDDISCONNECT(nHandle)*

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

<b><i>nHandle</i></b>	<b><i>BYVAL nHandle AS INTEGER</i></b> The handle of the connection to be dropped.
-----------------------	---

### Examples:

```

dim rc

'//BT address of device to connect to. You will have to change this
dim mac$
mac$ = "\00\16\A4\09\3A\64"

//=====
// Called after a connection attempt
//=====
function HandlerHIDConn(portHndl, result) as integer
    print "\n --- Connect : ";integer.h' result

    // Disconnect immediately for the purpose of this demo
    rc = BtcHIDDisconnect(portHndl)
    if rc==0 then
        print "\n\nDisconnecting ..."
    else
        print "\nError:", integer.h'rc
    endif
endfunc 1

//=====
// Called after a disconnection attempt
//=====
function HandlerHidDiscon(portHndl) as integer
    print "\n --- Disconnected"

endfunc 1

//*****
// Equivalent to main() in C
//*****
ONEVENT EvHidConn          CALL HandlerHidConn
OnEvent EvHidDiscon        call HandlerHidDiscon

```

```
// We must open HID device or HID host before initiating connection
rc = BtcHidHostOpen()

//make hid connection
rc=BtcHIDConnect(mac$)

print "\nConnecting to device ";StrHexize$(mac$)

waitevent
```

### Expected Output:

```
Connecting to device 0016A4093A64
--- Connect : 00000000

Disconnecting ...
--- Disconnected
```

## BtcHIDRead

### FUNCTION

A function to read queued reports indicated by the reception of `EVBTC_HID_DATA_RECEIVED`. This function returns the handle of the device or host the report was received from and the report itself.

#### *BTCHIDREAD(nConnHandle , nReportHandle, nDropped)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nConnHandle</i></b>	<b><i>BYREF nConnHandle AS INTEGER</i></b> The handle of the connection that the report has been read from.
<b><i>nReportHandle</i></b>	<b><i>BYREF nReportHandle AS INTEGER</i></b> The report handle that is used to extract the report
<b><i>nDropped</i></b>	<b><i>BYREF nDropped AS INTEGER</i></b> The number of reports that were dropped

### Example :: BtcHIDWrite.sb

```
dim rc

//=====
// Called after a pairing attempt
//=====

function HandlerHIDConn(portHndl, result) as integer
    dim s$
    print "\n --- Connect : ";integer.h' result
```

```
endfunc 1

//=====
// Called after receiving HID data
//=====

function HandlerHidData() as integer
    dim hConnHndl, hRprtHndl, nDropped, nBitLen
    dim report$

    '// get the handle of the connection and the report
    rc = BtcHIDRead(hConnHndl, hRprtHndl, nDropped)
    '// now export the report and print it
    rc = HIDReportExport(hRprtHndl, nBitLen, report$)

    print"\nHandle: ";hConnHndl; " Data: ";report$

endfunc 1

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

ONEVENT EvHidConn          CALL HandlerHidConn
OnEvent EvBtc_HID_data_received CALL HandlerHidData

'// We must open HID device or HID host before initiating connection
rc = BtcHidHostOpen()
'// Set device to be connectable
rc = BtcSetConnectable(1)

print "\nWaiting for a HID device to connect and write to us"

waitevent
```

### Expected Output:

```
Waiting for a HID device to connect and write to us
--- Connect : 00000000
Handle: 130050 Data: 000005000000000000
```

## BtchIDWrite

### FUNCTION

A function to send a report to the Host or Device specified by the connection handle. Upon successful submission of this function the event, **EVHIDTXEMPTY** event is called with the handle specified in this function. Use the reception of this event to write more reports.

#### **BTCHIDWRITE**(*nConnHandle*, *nReportHandle*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nConnHandle</i></b>	<b>BYVAL <i>nHandle</i> AS INTEGER</b> The handle of the connection that the report shall be written too.
<b><i>nReportHandle</i></b>	<b>BYVAL <i>nReportHandle</i> AS INTEGER</b> The handle of the report we want to write

#### Example :: BtchIDRead.sb

```
dim rc

//=====
// Called after a pairing attempt
//=====

function HandlerHIDConn(hConnHndl, result) as integer
    dim s$
    dim hRprtHndl, nBitLen
    print "\n --- Connect : ";integer.h' result

    // Write data as soon as we connect
    s$ = "1"
    // Get the bit length of the report
    nBitLen = (StrLen(s$) * 8)
    // Create a new HID report
    rc = HIDReportInit(nBitLen, hRprtHndl)
    // Add the string to it
    rc = HIDReportAppendStr(hRprtHndl, 0, nBitLen, s$)
    // Now write
    rc = BtchIDWrite(hConnHndl, hRprtHndl)

    if rc==0 then
        print "\nHid Write Successful"
    else
        print "\nError: "; integer.h'rc
    endif
end if
```



```
// Finally destroy the report
rc = HIDReportDestroy(hRprtHndl)

endfunc 1

//*****
// Equivalent to main() in C
//*****
ONEVENT EvHidConn          CALL HandlerHidConn

'// We must open HID device or HID host before initiating connection
rc = BtcHidHostOpen()
'// Set device to be connectable
rc = BtcSetConnectable(1)

print "\nWaiting for a HID device to connect to us"

waitevent
```

#### Expected Output:

```
Waiting for a HID device to connect and write to us
--- Connect : 00000000
Hid Write Successful
```

## BtcHIDControl

### FUNCTION

A function to send a control event to the Host or Device specified by the connection handle. Local Hosts bay send all control events however local devices may only send the **HID\_CONTROL\_VCABLEUNPLUG** control event, upon which the connection is terminated.

#### **BTCHIDCONTROL**(*nHandle*, *nControl*)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
Arguments:		
nHandle	BYVAL nHandle AS INTEGER The handle of the connection that the report shall be written to.	
nControl	BYVAL nControl AS INTEGER The control event to send to the remote host or device specified by the connection handle.	
	0	Nop

1	Hard Reset
2	Soft Reset
3	Suspend
4	Exit Suspend
5	Virtual cable unplug

#### Example :: BtcHIDDisconnect.sb

```

dim rc

#define HID_CONTROL_NO_OPERATION      0
#define HID_CONTROL_HARD_RESET       1
#define HID_CONTROL_SOFT_RESET       2
#define HID_CONTROL_SUSPEND          3
#define HID_CONTROL_EXIT_SUSPEND     4
#define HID_CONTROL_VIRTUAL_CABLE_UNPLUG  5

//=====
// Called after a connection attempt
//=====

function HandlerHIDConn(portHndl, result) as integer
    print "\n --- Connect : ";integer.h' result

    // Immediately send a control event to the device upon connecting
    rc = BtcHIDControl(portHndl, HID_CONTROL_SOFT_RESET)

    if rc==0 then
        print "\nHID control: soft reset ..."
    else
        print "\nError:", integer.h'rc
    endif

endfunc 1

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

ONEVENT EvHidConn      CALL HandlerHidConn

'// We must open HID device or HID host before initiating connection
rc = BtcHidHostOpen()
'// Set device to be connectable
rc = BtcSetConnectable(1)

```

```
print "\nWaiting for a HID device to connect and write to us"

waitevent
```

#### Expected Output:

```
Waiting for a HID device to connect and write to us
--- Connect : 00000000
HID control: soft reset ...
```

## BtcHIDConfig

### FUNCTION

A function to configure various HID settings.

#### *BTCHIDCONFIG(nKey, nValue)*

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
Arguments:		
nKey	BYVAL nKey AS INTEGER The key of the configuration value to be modified from the following list.	
	0	Incoming HID Report internal buffer queue size in number of reports.
nValue	BYVAL nValue AS INTEGER The value that the configuration value should be changed too.	

## Serial Port Profile

Serial Port Profile (abbreviated SPP) is used 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 *smart*BASIC application, and contains **nHandle** (the handle of the connection) and **result** (a result code). **nHandle** is only valid on a successful result code (0).

**Note:** When using a 4 MHz clock speed and bridging the UART to SPP, the fastest supported baud rate is 115200.

Possible errors are:

SPP_CONNECTION_TIMEOUT	0x01
SPP_CONNECTION_REFUSED	0x02
SPP_UNKNOWN_ERROR	0x03
SDP_TIMEOUT	0x10

SDP_CONNECTION_ERROR	0x11
SDP_ERROR_RESPONSE	0x12
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\(\)](#).

### *EVSPDISCON*

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

#### **Example:**

```

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 1

function HandlerSppDiscon(portHndl) as integer
    print "\n --- Disconnect : ", portHndl
endfunc 0

```

```
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 BT900 : 000016A4093A5F
Module is Discoverable. Make an SPP connection to the module.

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

#### *EVSPSTATUS*

This event from the SPP manager informs the application of a status event or break. The event contains:

**nHandle** = The handle of the connection.

**nStatus** = The status of the modem control signaling lines as a bitmask (see **BtcSppSendStatus()**).

**nBreak** = 0 for no Break signal and 1 for Break signal present.

**nBreakTime** = Time in milliseconds to issue break for.

## BtcSppSendStatus

### FUNCTION

The BT900 supports RFCOMM Modem Control Signaling, which is the transfer of RS232 port status over-the-air using TS 07.10 signals RTC, RTR, IC, and DV. See the table below for mapping of these signals to their RS232 controls:

TS 07.10 Signals	Corresponding RS-232 Control Signals
RTC	DSR, DTR
RTR	RTS, CTS
IC	RI
DV	DCD

This function is used to send RFCOMM modem control signaling status to the remote connected device.

### *BTCSPSENDSTATUS (nHandle, nStatus)*

<b>Returns</b>	INTEGER, indicating the success of command
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVal nHandle AS INTEGER</b> This specifies the connection handle on which to send status events.
<b>nStatus</b>	<b>byVal nStatus AS INTEGER</b> The serial status as a bitmask: RTS/CTS (RTR), Bit 0 DTR/DSR (RTC), Bit 1 RING (IC), Bit 2 DCD (DV), Bit 3.

## BtcSppSendBreak

### FUNCTION

This function sends a BREAK event to the remote connected device with the time specified.

### *BTCSPSEENDBREAK(nHandle, nBreakTime)*

<b>Returns</b>	INTEGER, indicating the success of command
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVal nHandle AS INTEGER</b> This specifies the connection handle on which to send break events.
<b>nBreakTime</b>	<b>byVal nBreakTime AS INTEGER</b> The amount of time in ms to send the break event for

## 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 affects the maximum number of SPP connections achievable.

#### *BTCSPSETPARAMS (nFrameSize, nReceiveCreds)*

<b>Returns</b>	<p>INTEGER, indicating the success of command:</p> <ul style="list-style-type: none"> <li>Opened successfully</li> </ul>
<b>Arguments:</b>	
<b><i>nFrameSize</i></b>	<p><b><i>byRef nFrameSize AS INTEGER</i></b> The maximum frame size supported on new SPP connections. Default 192 Bytes, Range is from 23-1011 bytes.</p>
<b><i>nReceiveCreds</i></b>	<p><b><i>byRef nReceiveCreds AS INTEGER</i></b> Number of receive packets to queue. Default 3, Range is from 1-10 packets.</p>

#### Example:

```
dim rc
rc=BtcSppSetParams (256,6)

if rc == 0 then
    print "\nSPP Parameters updated."
else
    print "\nError: ";rc
endif
```

#### Expected Output:

```
SPP Parameters updated.
```

## BtcSPPOpen

### FUNCTION

This function is used to open the serial port service and listen for SPP connections.

#### *BTCSPPOPEN (nHandle)*

<b>Returns</b>	<p>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</p>
<b>Arguments:</b>	
<b><i>nHandle</i></b>	<p><b><i>byRef nHandle AS INTEGER</i></b> On return this contains the handle for the SPP service.</p>

#### Example:

```
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
```

## BtcSPPClose

### FUNCTION

Close the Serial Port being expedited by SPP Service.

#### *BTCSPPCLOSE* (*nHandle*)

<b>Returns</b>	INTEGER, indicating the success of command: Opened successfully
<b>Arguments:</b>	
<i>nHandle</i>	<b>byVal nHandle AS INTEGER</b> The handle of the SPP connection to close

#### Example:

```
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
```

## BtcSPPWrite

### FUNCTION

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

#### *BTCSPWRITE (nHandle, data\$, nLen)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<i>byVal nHandle AS INTEGER</i> This contains the handle for the applicable SPP connection (the BT900 can be in a connection with multiple devices).
<b>data\$</b>	<i>byRef data\$ AS STRING</i> This contains the data to send over SPP
<b>nLen</b>	<i>byRef nLen AS INTEGER</i> On return this contains the number of bytes written.

**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.

### Example:

```
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
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

```

## BtcSPPRead

### FUNCTION

Read data from the oldest SPP data event. Since the event EVBTC\_SPP\_DATA\_RECEIVED is invoked 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.

#### **BTCSPPREAD** (*nHandle*, *data\$*, *nLen*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<i>byRef</i> <b>nHandle AS INTEGER</b> On return, this contains the handle of the SPP connection from which the data came.
<b>data\$</b>	<i>byRef</i> <b>data\$ AS STRING</b> On return, this contains the data received from the connection identified by the handle above.

<i>nLen</i>	<i>byRef nLen AS INTEGER</i> On return this contains the number of bytes read.
-------------	---

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

**Example:**

```

dim rc
dim hSpp
dim n$, a$

function HandlerSppConn(portHandle, result)
    print "\n --- Connect : ",portHandle
    print "\nResult: ";integer.h' result
endfunc 1

'//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 BT900 : 000016A4093A5F
Module is Discoverable. Make an SPP connection

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

## BtcSPPConnect

### FUNCTION

Connect to an SPP device defined by `btaddr$`. In the event of one device of a connected pair being forcibly reset, the partner may not establish a new connection until the link supervision timeout expires which is typically 20 seconds. In this event a timeout event is returned and the action would be to try again.

### *BTCSPPCONNECT (btaddr\$)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>btaddr\$</i></b>	<b><i>byRef btaddr\$ AS STRING</i></b> The Bluetooth address of the device for connection

### Example:

```
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$)
print "\nConnecting to device ";StrHexize$(BTA$)

function HandlerSppConn(portHndl, result) as integer
    hSpp = portHndl
    print "\n --- Connect : ",hSpp, StrHexize$(BTA$)
    print "\nResult: ",integer.h' result
endfunc 0

onevent EvSppConn call HandlerSppConn

waitevent

print "\nExiting..."

```

#### Expected Output:

```

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

```

## BtcSPPDisconnect

### FUNCTION

Disconnect from an SPP device.

#### **BTCSPPDISCONNECT**(*nHandle*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<i>nHandle</i>	<b>BYREF nHandle AS INTEGER</b> The handle of the connection to be dropped

**Example:**

```
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 1

//-----
// Called on an SPP disconnection
//-----

function HandlerSppDiscon(hConn) as integer
    print "\n --- Disconnected :", hConn
    // rc=BtcSppClose(hPort)
endfunc 0

onevent EvSppConn call HandlerSppConn
onevent EvSppDiscon call HandlerSppDiscon

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

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

waitevent
```

### Expected Output:

```
LAIRD BT900 : 000016A4093A5F
Module is Discoverable. Make an SPP connection

--- Connect :          40449
Result: 00000000

Disconnecting...
--- Disconnected : 40449
```

## Stream Functions

Stream functions are used to combine different types of streams (e.g. UART, SPP) together so that data transmission between these streams is handled in the background without using smartBASIC to manually move the data to another stream. This functionality is useful when no processing of the received data from a stream is needed as data between streams is sent unaltered. At any point, a stream bridge connection can be unbridged and control returned to smartBASIC.

### Events and Messages

#### *EVSTREAMIDLE*

This event is thrown when a stream bridge goes idle, this information can be useful in disconnecting a stalled stream. The timeout is configured with `StreamBridgeConfig()`.

### StreamGetUartHandle

#### FUNCTION

Returns the stream handle of the UART.

#### *STREAMGETUARTHANDLE(nStreamHandle)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<i>nHandle</i>	<b>BYREF bStreamHandle AS INTEGER</b> Returns the handle of the UART

See example for [StreamBridge](#).

## StreamGetSPPHandle

### FUNCTION

Get the stream handle of an SPP connection.

**STREAMGETSPPHANDLE(*nHandle*, *nStreamHandle*)**

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

See example for [StreamBridge](#).

## StreamBridge

### FUNCTION

Bridges two stream connections together.

**StreamBridge(*nHandleOne*, *nHandleTwo*, *nHandle*)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nHandle</i></b>	<b>BYVAL <i>nHandleOne</i> AS INTEGER</b> First stream port to bridge
<b><i>nHandle</i></b>	<b>BYVAL <i>nHandleTwo</i> AS INTEGER</b> Second stream port to bridge
<b><i>nHandle</i></b>	<b>BYREF <i>nHandle</i> AS INTEGER</b> Returns the handle of the bridged connection

### Example :: StreamBridge.sb

(See at: <https://github.com/LairdCP/BT900-Applications/tree/master/Manual%20Code>)

```
dim rc, nSHandleG, nHandleB, nSppHandle

#define UseStreamIdle 1 //When set to 1 will utilise the stream idle event (after 60 seconds
without data being received, the stream connection will be released and the SPP device
disconnected), when set to 0 the timeout event will not be enabled.
#define StreamIdleTimeout 60 //Timeout (in seconds) that will cause a stream idle event to
occur after this time has passed without receiving any SPP data (UseStreamIdle, above, must
be set to 1)

SUB AssertRC(rc,line)
    IF rc != 0 THEN
        PRINT "Error at line ";line;" , code: ";INTEGER.H'rc;"&"\n"
    ENDIF
ENDSUB
```



```
FUNCTION HandlerPairReq()  
    //Pair request  
    dim BTA$  
    rc=BtcGetPairRequestBDAddr(BTA$)  
    AssertRC(rc, 17)  
    PRINT "\nPairing requested from device: "; StrHexize$(BTA$)  
    PRINT "\nAccepting pair request"  
    rc=BtcSendPairResp(1)  
    AssertRC(rc, 21)  
ENDFUNC 1  
  
FUNCTION SPPConnect(nHandle, Result)  
    //SPP connected  
    dim UARTStream, SPPStream  
    nSppHandle = nHandle  
    PRINT "Connected\n"  
  
    //Bridge to UART  
    rc = StreamGetUartHandle(UARTStream)  
    AssertRC(rc, 32)  
    rc = StreamGetSPPHandle(nSppHandle, SPPStream)  
    AssertRC(rc, 34)  
    rc = StreamBridge(UARTStream, SPPStream, nHandleB)  
    AssertRC(rc, 36)  
    IF (UseStreamIdle == 1) THEN  
        //Stream idle timeout enabled  
        rc = StreamBridgeConfig(nHandleB, 0, StreamIdleTimeout)  
        AssertRC(rc, 40)  
    ENDIF  
ENDFUNC 1  
  
FUNCTION SPPTimeout()  
    //SPP connection timeout  
    PRINT "Timeout\n"  
ENDFUNC 1  
  
FUNCTION SPPDisconnect(nHandle)  
    //SPP disconnected  
    PRINT "Disconnected\n"  
ENDFUNC 1
```

```

FUNCTION HandlrStreamIdle(nHandle)
    //SPP Stream timeout (Only enabled if UseStreamIdle is set to 1)
    rc = StreamUnBridge(nHandleB)
    AssertRC(rc, 57)
    PRINT "Stream stalled, stream bridge released\n"
    rc = BtcSppDisconnect(nSppHandle)
    AssertRC(rc, 60)
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
IF (UseStreamIdle == 1) THEN
    ONEVENT EVSTREAMIDLE CALL HandlrStreamIdle //Stream idle timeout
ENDIF

WAITEVENT
    
```

### Expected Output:

```

Connected
Test Data from another BT900
Disconnected
    
```

## StreamUnBridge

### FUNCTION

Unbridges a stream connection created using StreamBridge.

#### *STREAMUNBRIDGE(nHandle)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
----------------	---

**Arguments:**

<b><i>nHandle</i></b>	<b><i>BYVAL nHandle AS INTEGER</i></b> The handle of the bridged connection to unbridge
-----------------------	--

See example for [StreamBridge](#).

## StreamBridgeConfig

### FUNCTION

Configures an existing stream bridge with a key value pair.

#### *STREAMBRIDGECONFIG(nHandle, nKey, nValue)*

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
Arguments:		
nHandle	BYVAL nHandle AS INTEGER The handle of the existing stream bridge to apply the config too.	
nKey	BYVAL nKey AS INTEGER The key of the configuration value to be modified from the following list.	
	0	Stream idle timeout (in seconds) of the specified bridge.
nValue	BYVAL nValue AS INTEGER The value that the configuration value should be changed to.	

See example for [StreamBridge](#).

## Pairing, Bonding, and Security Manager Functions

This section describes routines which manage all aspects of BTC security such as Pairing and Bonding, IO capabilities, OOB data, Passkey submission, and Just Works config. 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\_OOB\_AVAILABLE\_REQUEST*

This event is thrown when `BtcSecMngrOOBPref` is set to 2, prompt me, and requires the user respond with the availability of OOB data for a device with `BtcSecMngrOOBAvailable`.

### Example:

```
dim rc

//=====
// Called when there is a OOB data availability request
//=====
function HandlerOOBAvail()

    print "OOB data available ? \n"

endfunc 1

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

OnEvent EVBTC_OOB_AVAILABLE_REQUEST    call HandlerOOBAvail

// set device to be pairable and connectable
rc = BtcSetPairable(1)
rc = BtcSetConnectable(1)

// When pairing is in progress, ask me if Oob data is available
rc = BtcSecMngrOobPref(2)

print "Waiting for authentication request\n"

WAITEVENT
```

### Expected output:

```
Waiting for authentication request
OOB data available ?
```

### *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](#).

If you receive “BT\_HCI\_STATUS\_CODE\_PIN\_OR\_LINKKEY\_MISSING”, the device has a stale link key. The module does not delete a stale key automatically due to security concerns, please remove the key manually and re-pair.

#### **Example:**

```
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
```

```
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 1

OnEvent EVBTC_PIN_REQUEST      call HandlerPinReq
OnEvent EVBTC_PAIR_REQUEST     call HandlerPairReq
OnEvent EVBTC_PAIR_RESULT      call HandlerPairRes

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 BT900 : 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 BT900 : 000016A4093A5F
Module is Discoverable and Pairable. Pair with the module.

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

### *EBBTC\_AUTHREQ*

This message is thrown after a Secure Simple Pairing or Legacy authentication request comes. Its parameter denotes the type of authentication from the following list.

AuthType	Description
0	A device is requesting Secure Simple Pairing - JustWorks, use <a href="#">BtcSendPAIRResp</a> to respond.
1	A device is requesting Secure Simple Pairing - Passkey, use <a href="#">BtcSecMgrPasskey</a> to respond.
2	A device is requesting Secure Simple Pairing - OOB, use <a href="#">BtcSecMgrOOBKey</a> to respond.
3	A device is requesting Legacy PIN entry, use <a href="#">BtcSendPINResp</a> to respond.

### Example:

```
#define EBTCAUTHKEYTYPE_NONE      0
#define EBTCAUTHKEYTYPE_PASSKEY  1
#define EBTCAUTHKEYTYPE_OOB      2
#define EBTCAUTHKEYTYPE_PIN      3

dim rc

//=====
// Called after a pairing attempt
//=====

function HandlerAuthReq(reqType)

    print "Authentication type: "

    if reqType == EBTCAUTHKEYTYPE_NONE then
        print "NONE\n"
    elseif reqType == EBTCAUTHKEYTYPE_PASSKEY then
        print "PASSKEY\n"
    elseif reqType == EBTCAUTHKEYTYPE_OOB then
        print "OOB\n"
    elseif reqtype == EBTCAUTHKEYTYPE_PIN then
```

```
    print "PIN\n"
else
    print "UNKNOWN\n"
endif

endfunc 1

//=====
// Called after receiving a pair response
//=====

function HandlerPairResp(res)

    if res == 0 then
        print "Successfully paired\n"
    endif

endfunc 1

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

OnEvent EVBTC_AUTHREQ          call HandlerAuthReq

// Set the device to be pairable and connectable
rc = BtcSetPairable(1)
rc = BtcSetConnectable(1)

// Set JustWorks configuration to 1 to get EVBTC_AUTHREQ event (default)
rc = BtcSecMngrJustWorksConf(1)

print "Waiting for authentication request\n"

WAITEVENT
```

#### Expected Output:

```
Waiting for authentication request
Authentication type: NONE
```



## *EVBTC\_PASSKEY*

This message is thrown after a Secure Simple Pairing requiring the BT900 to display a passkey comes in, its parameter denotes the passkey to display, the passkey must be zero padded to 6 digits if shorter.

### **BtcGetPAIRRequestBDAddr**

#### **FUNCTION**

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

#### *BTCGETPAIRREQUESTBDADDR (strBDAddr\$)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>strBDAddr\$</b>	<b>byREF strBDAddr\$ AS STRING</b> On return this string contains the Bluetooth address of the device that the pairing request came from.

See examples given for [EVBTC\\_PAIR\\_RESULT](#) and [BtcPair](#).

### **BtcGetPINRequestBDAddr**

#### **FUNCTION**

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

#### *BTCGETPINREQUESTBDADDR (strBDAddr\$)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>strBDAddr\$</b>	<b>byREF strBDAddr\$ AS STRING</b> On return, this string contains the Bluetooth address of the device requesting a PIN.

See examples given for [EVBTC\\_PAIR\\_RESULT](#) and [BtcPair](#).

### **BtcSendPAIRResp**

#### **FUNCTION**

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

#### *BTCSENDPAIRRESP (nAccept)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nAccept</b>	<b>byVAL nAccept AS INTEGER</b> Decline Accept

See example given for [EVBTC\\_PAIR\\_RESULT](#).

## BtcSendPINResp

### FUNCTION

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

#### BTSENDPINRESP (*strPIN\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<i>strPIN\$</i>	<b>byVAL <i>strPIN\$</i> AS STRING</b> This is the PIN that is used. For example: 1234

See examples given for [EVBTC\\_PAIR\\_RESULT](#) and [BtcPair](#).

## 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.

#### BTSAVEPAIRINGS(*fSave*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<i>fSave</i>	<b>byVal <i>fSave</i> AS INTEGER</b> If this flag is: 0 – Pairing information is not stored in the bonding manager 1 – Pairing information is stored in the bonding manager

#### Example:

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

#### Expected Output:

```
0
```

## 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 BT900 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:		
strBDAddr\$	<b>byREF strBDAddr\$ AS STRING</b> The Bluetooth address of the device to pair with. Must be 6 bytes long.	
nSave	<b>byVal nSave AS INTEGER</b> This flag sets whether or not to bond.	
	Value	Description
	0	Do not store pairing information (don't bond)
	1	Store pairing information (bond)
	2	Use default as specified by BtcSavePairings()

#### Example:

```

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 1

//=====
// Called on a PIN request from another device
//=====

function HandlerPINReq()
    rc=BtcGetPinRequestBDAddr(adr$)
    print "\nPIN Req. Sending pin " + PIN
    rc=BtcSendPinResp(PIN)
endfunc 1

//=====
// Called after a pairing attempt
//=====

function HandlerPairRes(res)
    dim i : i=res
    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 BT900 : 000016A4093A5F
Connectable
Pairable
Initiating Pairing...
Pair Req: 94350AA99A3C
Accepted, Pairing...
--- Pair Result: (00000000) 94350AA99A3C
```

## BtcBondingStats

### FUNCTION

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

#### **BTCBONDINGSTATS** (*nRolling*, *nPersistent*)

<b>Returns</b>	The total capacity of the database
<b>Arguments:</b>	
<b><i>nRolling</i></b>	<b>byREF <i>nRolling</i> AS INTEGER</b> On return, this integer contains the total number of bonds in the rolling database.
<b><i>nPersistent</i></b>	<b>byREF <i>nPersistent</i> AS INTEGER</b> On return, this integer contains the total number of bonds in the persistent database.

#### Example:

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

#### 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\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>btaddr\$</i></b>	<b>byREF <i>btaddr\$</i> AS STRING</b> Bluetooth address in big endian. Must be exactly six bytes long.

#### Example:

```
dim rc, BTA$

'//-----
'// 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$(BTA$); " erased"
    endif
EndSub
```

```
BTA$ = "\00\80\98\04\4e\91"
rc=BtcBondingEraseKey(BTA$)
AssertRC(rc,17)
```

#### Expected Output:

```
Link key for device 008098044E91 erased
```

## BtcBondingEraseAll

### FUNCTION

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

#### *BTCBONDINGERASEALL ()*

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

#### Example:

```
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
```

## 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\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<i>btaddr\$</i>	<b>byREF</b> <i>btaddr\$</i> <b>AS STRING</b> Bluetooth address in big endian. Must be exactly six bytes long.

### Example:

```
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
```

## BtcBondingGetFirst

### FUNCTION

This function is used to retrieve details about the first classic Bluetooth bond in the BT900'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 format. Exactly six bytes long.
btkey\$	byREF btkey\$ AS STRING	Bluetooth bond key. Exactly 16 bytes long.
keytype	byREF keytype AS INTEGER	Returns the type of the key; values include the following:
	Value	Description
	0	Combination key
	1	Local unit key
	2	Remote unit key
	3	Debug combination key
	4	Unauthenticated combination key
	5	Authenticated combination key

	6	Changed combination key
	7	Illegal key
	<b>byREF <i>bonddb</i> AS INTEGER</b> Which database the key is in; values include the following:	
<b><i>bonddb</i></b>	<b>Value</b>	<b>Description</b>
	0	Persistent database
	1	Rolling database

**Example:**

```

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.\n"

    //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
            PRINT "persistent"
        ENDIF
        PRINT " database.\n"
    ELSE
        //No additional bonds
        PRINT "No additional bonds\n"
    ENDIF
ELSE
    //No bonds

```

```
PRINT "No bonds to output.\n"
ENDIF
```

#### Expected Output:

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

## BtcBondingGetNext

### FUNCTION

This function is used to retrieve details about the next classic Bluetooth bond in the BT900's 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)*

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
Arguments:		
btaddr\$	<b>byREF btaddr\$ AS STRING</b> Bluetooth address in big endian. It is exactly six bytes long.	
btkey\$	<b>byREF btkey\$ AS STRING</b> Bluetooth bond key. It is exactly sixteen bytes long.	
keytype	<b>byREF keytype AS INTEGER</b> Returns the type of the key; values include the following:	
	Value	Description
	0	Combination key
	1	Local unit key
	2	Remote unit key
	3	Debug combination key
	4	Unauthenticated combination key
	5	Authenticated combination key
	6	Changed combination key
bonddb	7	Illegal key
	<b>byREF bonddb AS INTEGER</b> Which database the key is in;	
	Value	Description
	0	Persistent database
	1	Rolling database

See example for `BtcBondingGetFirst`.

## BtcSecMngrPasskey

### FUNCTION

This function submits a passkey to the underlying stack during a pairing procedure when prompted by the EVBTC\_AUTHREQ with AuthType set to 1. See [Events and Messages](#).

#### *BTCSECMNGRPASSKEY(nPassKey)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nPassKey</b>	byVal nPassKey AS INTEGER. The passkey to submit to the stack. Submit a value outside the range 0 to 999999 to reject the pairing.

### Example:

```
#define EBTCAUTHKEYTYPE_PASSKEY      1

dim rc

//=====
// Called after a pairing attempt
//=====

function HandlerAuthReq(reqType)

    if reqType == EBTCAUTHKEYTYPE_PASSKEY then
        // We got a passkey request, send the passkey here
        print "Got a passkey request. Please enter the passkey\n> "
    endif

endfunc 1

//=====
// Called after receiving a pair response
//=====

function HandlerPairResp(res)

    if res == 0 then
        print "Successfully paired\n"
    endif

endfunc 1
```

```
//=====
// Called after data has been recieved via Uart
//=====
function HandlerUartRx()

    dim nMatch, nPassKey, stRsp$

    // read UART data until carriage return and save it into stRsp$
    nMatch=UartReadMatch(stRsp$,13)
    if nMatch!=0 then
        //Extract passkey from input string
        rc = ExtractIntToken(stRsp$, nPassKey)
        // Submit passkey
        rc = BtcSecMngrPassKey(nPassKey)
    endif

endfunc 1

//*****
// Equivalent to main() in C
//*****
OnEvent EVBTC_AUTHREQ      call HandlerAuthReq
OnEvent EVBTC_PAIR_RESULT  call HandlerPairResp
OnEvent EVUARTRX           call HandlerUartRx

// Set the device to be pairable and connectable
rc = BtcSetPairable(1)
rc = BtcSetConnectable(1)

// Set the io capability to 'Keyboard'
// This means we can enter passkey
rc = BtcSecMngrIoCap(2)
print "Waiting for authentication request\n"

WAITEVENT
```

**Expected output:**

```
Waiting for authentication request
Got a passkey request. Please enter the passkey
> 492585
Successfully paired
```

## BtcSecMngrJustWorksConf

### FUNCTION

This function is used to set the default action for when a pairing is in progress and the I/O Capability negotiation results in Just Works.

#### **BTCSECMNGRJUSTWORKSCONF**(*nJustWorksConf*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nJustWorksConf</i></b>	<p><b>byVal nJustWorksConf AS INTEGER.</b></p> <p>If set to 0, pairing <i>just works</i> without confirmation. If set to 1, when pairing is in progress, you get an <b>EVBTC_AUTHREQ</b> with AuthType 0, and <b>EVBTC_PAIR_REQUEST</b> event. In this case you accept or decline the pairing request with <b>BtcSendPAIRResp</b>.</p>

### Example:

```
dim rc

//=====
// Called when receiving a pair response
//=====
function HandlerPairRes(Resp)

    if Resp == 0 then
        print "Paired successfully\n"
    endif

endfunc 1

//*****
// Equivalent to main() in C
//*****
OnEvent EVBTC_PAIR_RESULT    call HandlerPairRes

// Set the device to be pairable and connectable
rc = BtcSetPairable(1)
rc = BtcSetConnectable(1)
```

```
// Set to 0 so that pairing just works
rc = BtcSecMngrJustWorksConf(0)

print "Waiting for authentication request\n"

WAITEVENT
```

#### Expected output:

```
Waiting for authentication request
Paired successfully
```

## BtcSecMngrOOBAvailable

### FUNCTION

This function is used indicate that OOB data is available for the requested connection, called as a result of [EVBTC\\_OOB\\_AVAILABLE\\_REQUEST](#).

#### *BTCSECMNGROOBAVAILABLE(nOobAvail)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nOobAvail</i></b>	<b>byVal nOobAvail AS INTEGER.</b> If set to 0, we do not have OOB data available. If set to 1, OOB data is available.

#### Example:

```
#define EBTCAUTHKEYTYPE_NONE      0
#define EBTCAUTHKEYTYPE_PASSKEY  1
#define EBTCAUTHKEYTYPE_OOB      2
#define EBTCAUTHKEYTYPE_PIN      3

dim rc

//=====
// Called when there is a OOB data availability request
//=====

function HandlerOOBAvail()

    print "\nOOB data available ? \n"

    // specify that oob data is not available
    rc = BtcSecMngrOobAvailable(0)

    if rc == 0 then
        print "Specified that oob data is not available\n"
```

```
endif

endfunc 1

//=====
// Called after a pairing attempt
//=====
function HandlerAuthReq(reqType)

    print "Authentication type: "

    if reqType == EBTCAUTHKEYTYPE_NONE then
        print "NONE\n"
    elseif reqType == EBTCAUTHKEYTYPE_PASSKEY then
        print "PASSKEY\n"
    elseif reqType == EBTCAUTHKEYTYPE_OOB then
        print "OOB\n"
    elseif reqType == EBTCAUTHKEYTYPE_PIN then
        print "PIN\n"
    else
        print "UNKNOWN\n"
    endif

endfunc 1

//*****
// Equivalent to main() in C
//*****
OnEvent EVBTC_AUTHREQ          call HandlerAuthReq
OnEvent EVBTC_OOB_AVAILABLE_REQUEST call HandlerOOBAvail

// set device to be pairable and connectable
rc = BtcSetPairable(1)
rc = BtcSetConnectable(1)

// When pairing is in progress, ask me if Oob data is available
rc = BtcSecMgrOobPref(2)

print "Waiting for authentication request\n"

WAITEVENT
```



### Expected output:

```
Waiting for authentication request
OOB data available ?
Specified that oob data is not available
Authentication type: NONE
```

## BtcSecMngrOOBPref

### FUNCTION

This function is used to specify if OOB data is available for the pairing process, this applies to all incoming and outgoing pairings. If a mix of devices may attempt pairing, “prompt me” is recommended.

#### *BTCSECMNGROOBPREF*(*nOobPreferred*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nOobPreferred</i></b>	<b>byVal nOobPreferred AS INTEGER.</b> 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 the prompt event is <a href="#">EVBTC_OOB_AVAILABLE_REQUEST</a> .

```
#define EBTCAUTHKEYTYPE_NONE      0
#define EBTCAUTHKEYTYPE_PASSKEY  1
#define EBTCAUTHKEYTYPE_OOB      2
#define EBTCAUTHKEYTYPE_PIN      3

dim rc

//=====
// Called after a pairing attempt
//=====

function HandlerAuthReq(reqType)

    print "Authentication type: "

    if reqType == EBTCAUTHKEYTYPE_NONE then
        print "NONE\n"
    elseif reqType == EBTCAUTHKEYTYPE_PASSKEY then
        print "PASSKEY\n"
    elseif reqType == EBTCAUTHKEYTYPE_OOB then
        print "OOB\n"
    elseif reqType == EBTCAUTHKEYTYPE_PIN then
        print "PIN\n"
```

```

else
    print "UNKNOWN\n"
endif

endfunc 1

//*****
// Equivalent to main() in C
//*****
OnEvent EVBTC_AUTHREQ    call HandlerAuthReq

// set device to be pairable and connectable
rc = BtcSetPairable(1)
rc = BtcSetConnectable(1)

// Specify that the oob data is not available
rc = BtcSecMngrOobPref(0)

print "Waiting for authentication request\n"

WAITEVENT

```

#### Expected Output:

```

Waiting for authentication request
Authentication type: NONE

```

## BtcSecMngrRetrieveLocalOOBKey

### FUNCTION

This function retrieves the local OOB hash and randomiser from the baseband, this must be transmitted to the pairing partner device out of band and applied using [BtcSecMngrOOBKey](#).

#### **BTCSECMNGRRETRIEVELOOBKEY(oobHash\$, oobRand\$)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>oobHash\$</b>	<b>byRef oobHash\$ AS STRING.</b> This is the OOB hash from the baseband and is a 16 byte string.
<b>oobRand\$</b>	<b>byRef oobKey\$ AS STRING.</b> This is the OOB randomiser from the baseband and is a 16 byte string.

**Example:**

```
dim rc, oobHash$, oobRand$

// Set device to be pairable
rc = BtcSetPairable(1)

// Retrieve local OOB Hash and Randomiser
rc = BtcSecMngrRetrieveLocalOobKey(oobHash$, oobRand$)

// Display the hash and the randomiser
print "Hash      : ";StrHexize$(oobHash$);"\n"
print "Randomiser : ";StrHexize$(oobRand$);"\n"
```

**Expected output:**

```
Hash      : 7457E8356F66DBB28887CA9D20C52348
Randomiser : 18B662B0E19A344E897E705FF4E986EF
```

## BtcSecMngrOOBKey

### FUNCTION

This function submits an OOB (Out Of Band) hash and randomiser received from the pairing partner to the underlying stack during a pairing procedure when prompted by the [EVBTC\\_AUTHREQ](#) with AuthType set to 2. See [Events & Messages](#).

#### **BTCSECMNGROOBKEY(oobHash\$, oobRand\$)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>oobHash\$</b>	<b>byVal oobHash\$ AS STRING.</b> This is the OOB hash to provide to the baseband and is a 16 byte string.
<b>oobRand\$</b>	<b>byRef oobRand \$ AS STRING.</b> This is the OOB randomiser to provide to the baseband and is a 16 byte string.

**Example:**

```
#define EBTCAUTHKEYTYPE_OOB      2

dim rc, uart$, stRsp$

//=====
// Called after a pairing attempt
//=====

function HandlerAuthReq(reqType)
```

```
if reqType == EBTCAUTHKEYTYPE_OOB then
    // We got a passkey request, send the passkey here
    print "Got an OOB request."
    print "Please set the hash and randomiser followed by a carriage return\n"
    print "> "
endif

endfunc 1

//=====
// Called after data has been recieved via Uart
//=====
function HandlerUartRx()

    dim nMatch, oobHash$, oobRand$

    // read UART data until carriage return and save it into stRsp$
    nMatch=UartReadMatch(stRsp$,13)

    if nMatch!=0 then

        // Get the hash and randomiser from the input string
        uart$ = strsplitleft$(stRsp$, nMatch)
        rc = ExtractStrToken(uart$,oobHash$)
        rc = ExtractStrToken(uart$,oobRand$)

        // Dehexize the data first
        oobHash$ = StrDeHexize$(oobHash$)
        oobRand$ = StrDeHexize$(oobRand$)

        // Submit oob data
        rc = BtcSecMngrOobKey(oobHash$, oobRand$)
    endif

endfunc 1

//=====
// Called after receiving a pair response
```

```
//=====
function HandlerPairResp(res)

    if res == 0 then
        print "Successfully paired\n"
    endif
endfunc 1

//-----
// Enable synchronous event handlers
//-----

OnEvent EVBTC_AUTHREQ      call HandlerAuthReq
OnEvent EVBTC_PAIR_RESULT  call HandlerPairResp
OnEvent EVUARTRX           call HandlerUartRx

// Set device to be pairable and connectable
rc = BtcSetPairable(1)
rc = BtcSetConnectable(1)

// Specify that the oob data is available
rc = BtcSecMngrOobPref(1)

print "Waiting for authentication request\n"

WAITEVENT
```

#### Expected output:

```
Waiting for authentication request
Got an OOB request.Please set the hash and randomiser followed by a carriage return
> 63F6E834009C368612724FBC3253DDE2 8311CD946F30C785DD7EA83038A5221D
Successfully paired
```

## BtcSecMngrIoCap

### 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:

[https://www.Bluetooth.org/docman/handlers/DownloadDoc.ashx?doc\\_id=86174](https://www.Bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=86174)

[https://www.Bluetooth.org/docman/handlers/DownloadDoc.ashx?doc\\_id=86173](https://www.Bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=86173)

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

### BTCSECMNGRIOCAP (*nloCap*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.		
<b>Arguments:</b>			
<b><i>nloCap</i></b>	<b>byVal <i>nloCap</i> AS INTEGER.</b>		
	The user I/O capability for all subsequent pairings.		
	0	None; also known as <i>Just Works</i> (unauthenticated pairing)	
	1	Display with Yes/No input capability (authenticated pairing)	
	2	Keyboard Only (authenticated pairing)	
	3	Display Only (authenticated pairing – if other end has input cap)	

### Example:

```

dim rc

//=====
// Called after receiving a pair response
//=====
function HandlerPairResp(res)

    if res == 0 then
        print "Successfully paired\n"
    endif

endfunc 1

//*****
// Equivalent to main() in C
//*****
OnEvent EVBTC_PAIR_RESULT    call HandlerPairResp

// Set device to be connectable and pairable
rc = BtcSetPairable(1)
rc = BtcSetConnectable(1)

// Set default action to JustWorks so that we don't get EVBTC_AUTHREQ event
rc = BtcSecMngrJustWorksConf(0)

// Set IO capability as Just Works

```

```
rc = BtcSecMngrIoCap(0)

print "Waiting for authentication request\n"

WAITEVENT
```

#### Expected Output:

```
Waiting for authentication request
Successfully paired
```

See also examples for [BtcSecMngrPasskey\(\)](#) and [BtcPair\(\)](#).

## Miscellaneous Functions

### Events and Messages

#### *EVBTC\_DISCOV\_TIMEOUT*

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

See example given for [BtcSetDiscoverable\(\)](#).

#### *EVBTC\_REMOTENAME\_RECEIVED*

This event is thrown when the module receives a response to a [BtcQueryRemoteFriendlyName](#) command, after this event has been received the values can be read by issuing [BtcGetRemoteFriendlyName](#).

#### *EVBTC\_MODE\_CHANGE*

This message from the LMP indicates that a link has changed its power mode. The mode change must be queried with [BtcQueryModeChange](#) to return the address of the device that is associated with that link. Status returns 0 on success. On a non-zero value, the link mode parameter is invalid.

#### *EVBTC\_SNIFF\_SUBRATING*

This message from the LMP indicates that a link has changed its sniff subrating parameters. The mode change must be queried with [BtcQuerySniffSubrating](#) to return the address of the device that is associated with that link. Status returns 0 on success.

### BtcTxPowerSet

#### FUNCTION

This function **does not exist**; its sole purpose in this manual is to draw attention the following note.

Bluetooth Core specification Volume 2, Part C, Section 4.1.3, Power Control describes the LMP dynamic power ranging method employed by the BT900. As such, when link quality varies the transmission power used ranges

up and down to try and achieve a consistent link quality. It is therefore counter intuitive to allow setting of an arbitrary transmit power for Classic, unlike BLE.

## BtcSetPNPInformation

### FUNCTION

Sets the PNP information in the SDP record.

#### *BTCSETPNPINFORMATION(IDSrc, VendorID, ProductID, Version, description)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>IDSrc</b>	<b>byVAL IDSrc AS INTEGER</b> Identifies the source of the Vendor ID field, 1 Bluetooth SIG assigned Company Identifier, 2 USB Implementor Forum assigned VID.
<b>VendorID</b>	<b>byVAL VendorID AS INTEGER</b> Identifies the product vendor from the namespace in the Vendor ID Source.
<b>ProductID</b>	<b>byVAL ProductID AS INTEGER</b> Manufacturer managed identifier for this product
<b>Version</b>	<b>byVAL Version AS INTEGER</b> Manufacturer managed version for this product
<b>description\$</b>	<b>byREF description\$ AS STRING</b> Service description string.

### Example:

```
//Example :: BtcSetPNPInformation.sb

// Device settings
#define DEVICE_VID_SRC      0x2
#define DEVICE_VID          0x0077
#define DEVICE_PID          0x1234
#define DEVICE_VERSION      0x1

dim rc, devName$

devName$ = "BT900 BTC"

// Set the PNP information
rc = BtcSetPNPInformation(DEVICE_VID_SRC, DEVICE_VID, DEVICE_PID, DEVICE_VERSION, devName$)
if rc == 0 then
    print "PNP information set successfully\n"
else
    print "Failed to set PNP information\n"
endif
```



### Expected Output:

```
PNP information set successfully
```

## BtcGetBDAddrFromHandle

### FUNCTION

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

#### *BTCGETBDADDRFROMHANDLE (connHandle, strBDAddr\$)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>connHandle</b>	<b>byREF connHandle AS INTEGER</b> Handle of the connection from which to obtain the Bluetooth address
<b>strBDAddr\$</b>	<b>byREF strBDAddr\$ AS STRING</b> On return, this string contains the Bluetooth address of the device on the other end of the connection

See example for [BtcGetHandleFromBDAddr](#).

## BtcGetHandleFromBDAddr

### FUNCTION

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

#### *BTCGETHANDLEFROMBDADDR (strBDAddr\$, connHandle)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>strBDAddr\$</b>	<b>byREF strBDAddr\$ AS STRING</b> Bluetooth address of the device on the other end of the connection for which you want to obtain the handle.
<b>connHandle</b>	<b>byREF connHandle AS INTEGER</b> On return, this integer contains the connection handle.

### Example:

```
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
    rc=BtcGetHandleFromBDAddr(addr$, h)
    print "\nConnection Handle obtained from BT Address: ";h
else
    print "\nError obtaining Bluetooth address: "; integer.h'rc
endif
rc=BtcSppDisconnect(hConn)
endfunc 1

onevent EvSppConn call HandlerSppCon

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

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

waitevent
```

#### Expected Output:

```
LAIRD BT900 : 000016A4093A5F
Module is Discoverable. Make an SPP connection

--- Connect :      40449
Result:      00000000
Connected to device: 0016A4093A92
Connection Handle obtained from BT Address: 40449
```

## 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:

0	Public
1	Random Static
2	Random Private Resolvable
3	Random Private Non-Resolvable
All other values are illegal	

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:

DIM addr	addr="\00\11\22\33\44\55\66"
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 appear in little endian format, hence reverse order – and you don't see seven bytes, but a bit in the packet somewhere which specifies it to be public or random.

## BleSetAddressType

### FUNCTION

This functions 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*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
<b>Arguments:</b>		
<b><i>nAddrType</i></b>	<b>byVal <i>nAddrType</i> AS INTEGER.</b> Specifies the type of the LE address as follows:	
	0	Public address, same as Classic.
	1	Random static address, generated first boot.
	2	Random address, resolvable with IRK, generated on call.
	3	Random address, non resolvable, generated on call

### Example:

```
DIM rc

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

### Expected Output:

```
rc = 0
```

## Events and Messages

### EVBLE\_ADV\_TIMEOUT

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

### Example:

```
//Example :: EvBle_Adv_Timeout.sb (See in BT900CodeSnippets.zip)
DIM peerAddr$

//handler to service an advert timeout
FUNCTION HndlrBleAdvTimOut()
    PRINT "\nAdvert stopped via timeout"
    //DbgMsg( "\n - could use SystemStateSet(0) to switch off" )

    //-----
```

```
// Switch off the system - requires a power cycle to recover
//-----
// rc = SystemStateSet(0)
ENDFUNC 0

//start adverts
//rc = BleAdvertStart(0,"",100,5000,0)
IF BleAdvertStart(0,peerAddr$,100,2000,0)==0 THEN
    PRINT "\n Advert Started"
ELSE
    PRINT "\n\nAdvert not successful"
ENDIF

ONEVENT EVBLE_ADV_TIMEOUT CALL HndlrBleAdvTimOut

WAITEVENT
```

#### Expected Output:

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

### 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 *smartBASIC* 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.

MsgID	Description
0	A BLE connection is established and msgCtx is the connection handle.
1	A BLE disconnection event and msgCtx identifies the handle.
4	A BLE Service Error. The second parameter contains the error code.
9	Pairing in progress and displayed Passkey supplied in msgCtx.
10	A new bond has been successfully created.
11	Pairing in progress and authentication key requested. msgCtx is key type.
14	Connection parameters update and msgCtx is the conn handle.
15	Connection parameters update fail and msgCtx is the conn handle.
16	Connected to a bonded master and msgCtx is the conn handle.
17	A new pairing has replaced old key for the connection handle specified.
18	The connection is now encrypted and msgCtx is the conn handle.
20	The connection is no longer encrypted and msgCtx is the conn handle
21	The device name characteristic in the GAP service of the local GATT table has been written by the remote GATT client.
22	Attempt to add a new bonding to the bonding database failed
23	On a BLE connection to a bonded device, if the current GATT table schema does not match what existed at the last connection, then a GATT Service Change Indication is automatically sent and the app is informed via this event
24	On a BLE connection to a bonded device, if the current GATT table schema does not match what existed at the last connection, then a GATT Service Change Indication is automatically sent and the app is informed when the client acknowledges that indication

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

### Example:

```
//Example :: EvBleMsg.sb (See in BT900CodeSnippets.zip)
DIM addr$ : addr$=""
```

```
DIM rc

//=====
// This handler is called when there is a BLE message
//=====

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

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

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

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

// start adverts
IF BleAdvertStart(0,addr$,100,10000,0)==0 THEN
    PRINT "\nAdverts Started"
```

```
PRINT "\nPress any key to exit\n"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

#### Expected Output (When connection made with the module):

```
Adverts Started
Press any key to exit

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

Exiting...
```

#### Expected Output (When no connection made):

```
Adverts Started
Press any key to exit

Advert stopped via timeout
Exiting...
```

## EVDISCON

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

- Connection handle
- The reason for the disconnection.

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

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

#### Example:

```
//Example :: EvDiscon.sb (See in BT900CodeSnippets.zip)
DIM addr$ : addr$=""
```



```
FUNCTION HndlrBleMsg (BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER)
    IF nMsgID==0 THEN
        PRINT "\nNew Connection ";nCtx
    ENDIF
ENDFUNC 1

FUNCTION Btn0Press()
    PRINT "\nExiting..."
ENDFUNC 0

FUNCTION HndlrDiscon (BYVAL hConn AS INTEGER, BYVAL nRsn AS INTEGER) AS INTEGER
    PRINT "\nConnection ";hConn;" Closed: 0x";nRsn
ENDFUNC 0

ONEVENT EVBLEMSG CALL HndlrBleMsg
ONEVENT EVDISCON CALL HndlrDiscon

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

WAITEVENT
```

### Expected Output:

```
Adverts Started

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

## EVCONNPARAMREQ

This event is thrown when a peripheral requests an update to the connection parameters. The user must turn manual parameter control on to receive this message by using `BleConnectConfig(8,1)`. In this case autoaccept is disabled and full control is given to the user.

The parameters given are: Conn\_handle, Interval\_Min, Interval\_Max, Supervision\_Timeout, Slave\_Latency.

## EVCHARVALEX

This event is thrown when a characteristic is written to by a remote GATT client **and the config key 51 is set to 1 (not default)**. It comes with four parameters:

- Characteristic handle that was returned when the characteristic was registered using the function `BleCharCommit()`
- Offset
- Length of the data from the characteristic value
- The new characteristic value data as a string variable

*Please note: The event messages EVCHARVAL and EVCHARVALEX are mutually exclusive and controlled via the new non-volatile config key 51. The config key can be changed in command mode using “AT+CFG 51 nn” or in run mode using the functions `NvCfgKeyGet()`/`NvCfgKeySet()`. The use of `NvCfgKeyGet/Set` is recommended in your application as that eliminates an extra step of configuring the module in your end device production. See the example below.*

### Example:

```
DIM hMyChar,rc,at$,conHndl

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, kv, hSvc, attr$, adRpt$, addr$, scRpt$ : attr$="Hi"
    //Enable Strings in Event messages
    rc=NvCfgKeyGet(51,kv)
    IF rc==0 THEN
        IF kv==0 THEN
            rc = NvCfgKeySet(51,1)
            IF rc==0 THEN
                Reset(0)
            ELSE
                PRINT "\nFailed to update config key 51"
            ENDIF
        ELSE
            PRINT "\nOld firmware, Key 51 could not be read"
        ENDIF
    ELSE
        //commit service
        rc=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
        rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
```

```
//initialise char, write/read enabled, accept signed writes
rc=BleCharNew(0x0A,BleHandleUuid16(1),BleAttrMetaData(1,1,20,0,rc),0,0)
//commit char initialised above, with initial value "hi" to service 'hSvc'
rc=BleCharCommit(hSvc,attr$,hMyChar)
//commit changes to service
rc=BleServiceCommit(hSvc)
rc=BleScanRptInit(scRpt$)
//Add 1 service handle to scan report
//rc=BleAdvRptAddUuid16(scRpt$,0x18EE,-1,-1,-1,-1,-1)
//commit reports to GATT table - adRpt$ is empty
rc=BleAdvRptsCommit(adRpt$,scRpt$)
rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC rc

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

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

//=====
// New char value handler
//=====
FUNCTION HandlerCharValEx(BYVAL charHandle, BYVAL offset, BYVAL len, BYVAL data$)
    DIM s$
```

```
IF charHandle == hMyChar THEN
    PRINT "\n";len;" byte(s) have been written to char value attribute from offset ";offset
    PRINT "\nNew Char Value: ";strhexize$(data$)
ENDIF

CloseConnections()

ENDFUNC 1

ONEVENT EVCHARVALEX CALL HandlerCharValEx
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...
```

## EVCHARVAL

This event is thrown when a characteristic is written to by a remote GATT client **and the config key 51 is set to 0 (default)**. 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

**Example:**

```
//Example :: EvCharVal.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, hSvc, attr$, adRpt$, addr$, scRpt$ : attr$="Hi"

    //commit service
    rc=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    //initialise char, write/read enabled, accept signed writes
    rc=BleCharNew(0x0A,BleHandleUuid16(1),BleAttrMetaData(1,1,20,0,rc),0,0)
    //commit char initialised above, with initial value "hi" to service 'hSvc'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc=BleServiceCommit(hSvc)
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    //rc=BleAdvRptAddUuid16(scRpt$,0x18EE,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC rc

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//=====
// Ble event handler
//=====

FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
```

```

conHndl=nCtx
IF nMsgID==1 THEN
    PRINT "\n\n--- Disconnected from client"
    EXITFUNC 0
ELSEIF nMsgID==0 THEN
    PRINT "\n--- Connected to client"
ENDIF
ENDFUNC 1

//=====
// New char value handler
//=====

FUNCTION HandlerCharVal(BYVAL charHandle, BYVAL offset, BYVAL len)
    DIM s$
    IF charHandle == hMyChar THEN
        PRINT "\n";len;" byte(s) have been written to char value attribute from offset ";offset

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

ONEVENT EVCHARVAL CALL HandlerCharVal
ONEVENT EVBLEMSG CALL HndlrBleMsg

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nThe characteristic's value is ";at$
    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\(\)](#)

### Example:

```
// 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

### Example:

```
//Example :: EvCharCccd.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, hSvc, metaSuccess, at$, attr$, adRpt$, addr$, scRpt$
    attr$="Hi"
```

```
DIM svcUuid : svcUuid=0x18EE
DIM charUuid : charUuid = BleHandleUuid16(1)
DIM charMet : charMet = BleAttrMetadata(0,0,20,1,metaSuccess)
DIM hSvcUuid : hSvcUuid = BleHandleUuid16(svcUuid)
DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc) //CCCD metadata for char

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

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

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

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

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

//=====
// Close connections so that we can run another app without problems
//=====
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    rc=GpioUnbindEvent(1)
ENDSUB

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



```
ENDIF
ENDFUNC 1

//=====
// Indication acknowledgement from client handler
//=====
FUNCTION HndlrCharHvc(BYVAL charHandle AS INTEGER) AS INTEGER
    IF charHandle == hMyChar THEN
        PRINT "\nGot confirmation of recent indication"
    ELSE
        PRINT "\nGot confirmation of some other indication: ";charHandle
    ENDIF
ENDFUNC 1

//=====
// Called when data received via the UART
//=====
FUNCTION HndlrUartRx() AS INTEGER
ENDFUNC 0

//=====
// CCCD descriptor written handler
//=====
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$
    IF charHandle==hMyChar THEN
        IF nVal & 0x02 THEN
            PRINT "\nIndications have been enabled by client"
            value$="hello"
            IF BleCharValueIndicate(hMyChar,value$)!=0 THEN
                PRINT "\nFailed to indicate new value"
            ENDIF
        ELSE
            PRINT "\nIndications have been disabled by client"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1
```

```
ONEVENT EVBLEMSG CALL HndlrBleMsg
ONEVENT EVCHARHVC CALL HndlrCharHvc
ONEVENT EVCHARCCCD CALL HndlrCharCccd
ONEVENT EVUARTRX CALL HndlrUartRx

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nValue of the characteristic ";hMyChar;" is: ";at$
    PRINT "\nYou can write to the CCCD characteristic."
    PRINT "\nThe BT900 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 BT900 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:**

```
//Example :: EvCharSccd.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc,chVal$,conHndl

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

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

    //initialise broadcast capable, readable, writeable
    rc=BleCharNew(0x0B,BleHandleUuid16(1),charMet,0,BleAttrMetadata(1,1,1,0,rc2))

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

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

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

//=====
// Close connections so that we can run another app without problems
//=====
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    rc=GpioUnbindEvent(1)
ENDSUB

//=====
// Broadcast characterstic value
//=====
FUNCTION PrepAdvReport()
```

```
dim adRpt$, scRpt$, svcDta$

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

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

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

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

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

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

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

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

//=====
// Ble event handler
//=====
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n--- Disconnected from client"
        dim addr$
        rc=BleAdvertStart(0,addr$,20,300000,0)
        IF rc==0 THEN
            PRINT "\nYou should now see the new characteristic value in the advertisment data"
```

```
ENDIF
ELSEIF nMsgID==0 THEN
    PRINT "\n--- Connected to client"
ENDIF
ENDFUNC 1

//=====
// Called when data arrives via UART
//=====

FUNCTION HndlrUartRx()
ENDFUNC 0

//=====
// CCCD descriptor written handler
//=====

FUNCTION HndlrCharSccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$
    IF charHandle==hMyChar THEN
        IF nVal & 0x01 THEN
            PRINT "\nBroadcasts have been enabled by client"
            IF PrepAdvReport()==0 THEN
                rc=BleDisconnect(conHndl)
                PRINT "\nDisconnecting..."
            ELSE
                PRINT "\nError Committing advert reports: ";integer.h'rc
            ENDIF
        ELSE
            PRINT "\nBroadcasts have been disabled by client"
            IF ResetAdvReport()==0 THEN
                PRINT "\nAdvert reports reset"
            ELSE
                PRINT "\nError Resetting advert reports: ";integer.h'rc
            ENDIF
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

//=====
// New char value handler
//=====
```

```
FUNCTION HndlrCharVal (BYVAL charHandle, BYVAL offset, BYVAL len)
    DIM s$
    IF charHandle == hMyChar THEN
        rc=BleCharValueRead(hMyChar,chVal$)
        PRINT "\nNew Char Value: ";chVal$
    ENDIF
ENDFUNC 1

//=====
// Called after a disconnection
//=====

FUNCTION HndlrDiscon(hConn, nRsn)
    dim addr$
    rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC 1

ONEVENT EVBLEMSG CALL HndlrBleMsg
ONEVENT EVCHARSCCD CALL HndlrCharSccd
ONEVENT EVUARTRX CALL HndlrUartRx
ONEVENT EVCHARVAL CALL HndlrCharVal
ONEVENT EVDISCON CALL HndlrDiscon

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,chVal$)
    PRINT "\nCharacteristic Value: ";chVal$
    PRINT "\nWrite a new value to the characteristic, then enable broadcasting.\nThe module will then
    disconnect and broadcast the new characteristic value."
    PRINT "\n--- Press any key to exit\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

CloseConnections()

PRINT "\nExiting..."
```

### Expected Output:

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

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

--- Disconnected from client
You should now see the new characteristic value in the advertisement 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:

```
//Example :: EvCharDesc.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl, hOtherDescr

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====
FUNCTION OnStartup$()
    DIM rc, hSvc, at$, adRpt$, addr$, scRpt$, hOtherDscr,attr$, attr2$, rc2
    attr$="Hi"
    DIM charMet : charMet = BleAttrMetaData(1,0,20,0,rc)

    //Commit svc with handle 'hSvcUuid'
    rc=BleServiceNew(1,BleHandleUuid16(0x18EE),hSvc)
    //initialise characteristic - readable
    rc=BleCharNew(0x02,BleHandleUuid16(1),charMet,0,0)
```

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

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

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

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

//=====
// Close connections so that we can run another app without problems
//=====
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    rc=GpioUnbindEvent(1)
ENDSUB

//=====
// Ble event handler
//=====
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n--- Disconnected from client"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n\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 descriptor ";hDesc;" with uuid ";integer.h'duid;" is ";a$
        ELSE
            PRINT "\nCould not read the descriptor value"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT EVBLEMSG CALL HndlrBleMsg
ONEVENT EVCHARDESC CALL HndlrCharDesc
ONEVENT EVUARTRX CALL HndlrUartRx

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

WAITEVENT

CloseConnections()

PRINT "\nExiting..."
```

### Expected Output:

```
Other Descriptor Value: my char desc
Write a new value
--- Press any key to exit

--- Connected to client
New value for desriptor 0 with uuid FE012901 is hello
```

## EVNOTIFYBUF

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

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

---

**Note:** When sending data using Indications, this event is not thrown because those messages have to be confirmed by the client which results in a `EVCHARHVC` message to the *smart*BASIC application. Likewise, writes which are acknowledged also do not consume these buffers.

---

### Example:

```
//Example :: EvNotifyBuf.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl,ntfyEnabled

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, hSvc, at$, attr$, adRpt$, addr$, scRpt$
    attr$="Hi"
    DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc) //CCCD metadata for char

    //Commit svc with handle 'hSvc'
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    rc=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
    //initialise char, write/read enabled, accept signed writes, notifiable
    rc=BleCharNew(0x12,BleHandleUuid16(1),BleAttrMetaData(1,0,20,0,rc),mdCccd,0)
    //commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc=BleServiceCommit(hSvc)
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,0x18EE,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,50,0,0)
```

```
ENDFUNC rc

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

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

//=====
// Ble event handler
//=====

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

//=====
// Tx Buffer free handler
```

```
//=====
FUNCTION HndlrNtfyBuf ()
    SendData ()
ENDFUNC 0

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

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

IF OnStartup ()==0 THEN
    rc = BleCharValueRead (hMyChar, at$)
    PRINT "\nYou can connect and write to the CCCD characteristic."
    PRINT "\nThe BT900 will then send you data until buffer is full\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
```

```
CloseConnections()  
PRINT "\nExiting..."
```

#### Expected Output:

```
You can connect and write to the CCCD characteristic.  
The BT900 will then send you data until buffer is full  
  
--- Connected to client  
Notifications have been disabled by client : Notifications have been enabled by client  
--- Notifying  
--- Buffer full  
Notified 1818505336 times  
Exiting...
```

## Miscellaneous Functions

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

### BleTxPowerSet

#### FUNCTION

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

The actual value is determined in the radios internal power table and accepts values between 10 and -20 in 1dB steps. At any time SYSINFO(2008) returns the actual transmit power setting (when in command mode, 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*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nTxPower</i></b>	<b>byVal <i>nTxPower</i> AS INTEGER.</b> Specifies the new transmit power in dBm units to be used for all subsequent tx packets. The actual value is determined by the radios internal power table.

#### Example:

```
//Example :: BleTxPowerSet.sb (See in BT900CodeSnippets.zip)  
DIM rc,dp  
  
dp=1000 : rc = BleTxPowerSet (dp)
```

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

#### Expected Output:

```
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
```

## BleTxPwrWhilePairing

### FUNCTION

This function sets the transmit power of all packets that are transmitted while a pairing is in progress. This mode of pairing is referred to as Whisper Mode Pairing. The actual value is clipped to the transmit power for normal operation which is set using BleTxPowerSet() function.

The actual value is determined in the radios internal power table and accepts values between 10 and -20 in 1dB steps.

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

Although this function can accept any value 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.

### BLETXPWRWHILEPAIRING(*nTxPower*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nTxPower</i></b>	<b>byVal <i>nTxPower</i> AS INTEGER.</b> Specifies the new transmit power in dBm units to be used for all subsequent tx packets. The actual value is determined by the radios internal power table.

#### Example:

```
//Example :: BleTxPwrWhilePairing.sb (See in BT900CodeSnippets.zip)
DIM rc,dp

dp=1000 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nrc = ";rc
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
dp=8 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," ", " actual= "; SysInfo(2018)
dp=2 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," ", " actual= "; SysInfo(2018)
dp=-10 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
dp=-25 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
dp=-45 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
dp=-1000 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
```

#### Expected Output:

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

## BleGetConnHandleFromAddr

### FUNCTION

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

**BLEGETCONNHANDLEFROMADDR(*macAddrBE\$, nConnHandle*)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>macAddrBE\$</i></b>	<b>byRef <i>macAddrBE\$</i> AS STRING.</b> The Bluetooth address of the connected remote device.
<b><i>nConnHandle</i></b>	<b>byRef <i>nConnHandle</i> AS INTEGER.</b> Returned connection handle.

### Example:

```
DIM rc, periphAddr$

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

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

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

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

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



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

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

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

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

WAITEVENT
```

### Expected Output:

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

## BleGetAddrFromConnHandle

### FUNCTION

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

**BLEGETADDRFROMCONNHANDLE(*nConnHandle*, *macAddrBE\$*)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nConnHandle</i></b>	<b>byRef <i>nConnHandle</i> AS INTEGER.</b> Connection handle from which to get Bluetooth address
<b><i>macAddrBE\$</i></b>	<b>byRef <i>macAddrBE\$</i> AS STRING.</b> Returned Bluetooth address.

### Example:

```
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
```

## 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 **BLEADVPTINIT**, **BLEADVPTADDxxx**, and **BLEADVPTCOMMIT** 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 not currently 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, nFilterHandle*)

<b>Returns</b>	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 <code>nFlags</code> argument to <code>BleAdvRptInit()</code> 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.		
<b>Arguments:</b>			
<b>nAdvType</b>	<b>byVal nAdvType AS INTEGER.</b> Specifies the advertisement type as follows:		
	0	ADV_IND	Invites connection requests

	1	ADV_DIRECT_IND	Invites connection from addressed device
	2	ADV_SCAN_IND	Invites scan request for more advert data
	3	ADV_NONCONN_IND	Does not accept connections/active scans
<b>peerAddr\$</b>	<b>byRef peerAddr\$ AS STRING</b> It can be an empty string that is omitted if the advertisement type is not ADV_DIRECT_IND. This is only required when nAdvType == 1. When not empty, a valid address string is exactly seven octets long (for example: \00\11\22\33\44\55\66) where the first octet is the address type and the rest of the six octets is the usual Bluetooth address in big endian format (so the most significant octet of the address is at offset 1), whether public or random.		
	0	Public	
	1	Random Static	
	2	Random Private Resolvable	
	3	Random Private Non-Resolvable	
	All other values are illegal.		
<b>nAdvInterval</b>	<b>byVal nAdvInterval AS INTEGER.</b> 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. When using ADV_NONCONN_IND or ADV_SCAN_IND advert types the advertising interval must be atleast 100ms.		
<b>nAdvTimeout</b>	<b>byVal nAdvTimeout AS INTEGER.</b> 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. <b>WARNING: To save power, do not mistakenly set this to e.g. 100ms.</b>		
<b>nFilterHandle</b>	<b>byVal nFilterHandle AS INTEGER.</b> Specifies the whitelist handle to use with advertising, passing 0 will disable the use of whitelist.		

**Example:**

```
//Example :: BleAdvertStart.sb (See in BT900CodeSnippets.zip)
DIM addr$ : addr$=""

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

//The advertising interval is set to 25 milliseconds. The module will stop
//advertising after 60000 ms (1 minute)
IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
  PRINT "\nAdverts Started"
  PRINT "\nIf you search for Bluetooth devices on your device, you should see 'Laird BT900'"
ELSE
  PRINT "\n\nAdvertisement not successful"
```

```
ENDIF

ONEVENT EVBLE_ADV_TIMEOUT CALL HndlrBlrAdvTimOut

WAITEVENT
```

### Expected Output:

```
Adverts Started

If you search for Bluetooth devices on your device, you should see 'Laird BT900'

Advert stopped via timeout
Exiting...
```

## BleAdvertStop

### FUNCTION

This function causes the BLE module to stop advertising.

### *BLEADVERTSTOP ()*

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

### Example:

```
//Example :: BleAdvertStop.sb (See in BT900CodeSnippets.zip)
DIM addr$ : addr$=""
DIM rc

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

FUNCTION Btn0Press ()
    IF BleAdvertStop() == 0 THEN
        PRINT "\nAdvertising Stopped"
    ELSE
        PRINT "\n\nAdvertising failed to stop"
    ENDIF

    PRINT "\nExiting..."
ENDFUNC 0
```

```
IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
  PRINT "\nAdverts Started. Press button 0 to stop.\n"
ELSE
  PRINT "\n\nAdvertisement not successful"
ENDIF

rc = GpioSetFunc(16,1,2)
rc = GpioBindEvent(0,16,1)

ONEVENT EVBLE_ADV_TIMEOUT CALL HndlrBlrAdvTimOut
ONEVENT EVGPIOCHAN0      CALL Btn0Press

WAITEVENT
```

#### Expected Output:

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

Advertising Stopped
Exiting...
```

## 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:

<b>Advert Channel Mask</b>	Bit field detailing the channels to advertise on.
----------------------------	---

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

### **BLEADVERTCONFIG** (*configID,configValue*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
<b>Arguments:</b>		
<b>configID</b>	<b>byVal configID AS INTEGER.</b>	
	This identifies the value to update as follows:	
	0	Unused
	1	Unused
	2	Unused
	3	Advert Channel Mask
For all other configID values the function returns an error.		
<b>configValue</b>	<b>byVal configValue AS INTEGER.</b>	

This contains the new value to set in the parameters identified by configID.

## 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 BLEADVREPORTCOMMIT is called.

This report is for use with advertisement packets.

**BLEADVREPORTINIT(advRpt\$, nFlagsAD, nAdvAppearance, nMaxDevName)**

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
Arguments:		
advRpt\$	byRef advRpt\$ AS STRING. This contains an advertisement report.	
nFlagsAD	byVal nFlagsAD AS INTEGER. Specifies the flags AD bits where bit 0 is set for limited discoverability and bit 1 is set for general discoverability. Bit 2 will be forced to 1 and bits 3 & 4 will be forced to 0. Bits 3 to 7 are reserved for future use by the BT SIG and must be set to 0.	
nAdvAppearance	byVal nAdvAppearance AS INTEGER. Determines whether the appearance advert should be added or omitted as follows:	
	0	Omit appearance advert
	1	Add appearance advert as specified in the GAP service which is supplied via the BleGapSvcInit() function
nMaxDevName	byVal nMaxDevName AS INTEGER. The n leftmost characters of the device name specified in the GAP service. If this value is set to zero (0) then the device name is not included.	

### Example:

```
//Example :: BleAdvRptInit.sb (See in BT900CodeSnippets.zip)
DIM advRpt$ : advRpt$=""
DIM discovMode : discovMode=0
DIM advAppearance : advAppearance = 1
DIM maxDevName : maxDevName = 10

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

### Expected Output:

```
Advert report initialised
```



## 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)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>scanRpt</b>	<b>byRef scanRpt AS STRING.</b> This contains a scan report.

#### Example:

```
//Example :: BleScanRptInit.sb (See in BT900CodeSnippets.zip)
DIM scnRpt$ : scnRpt$=""

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

#### Expected Output:

```
Scan report initialised
```

## BleAdvRptGetSpace

### FUNCTION

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

#### BLEADVRPTGETSPACE(advRpt)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>advRpt\$</b>	<b>byRef advRpt\$ AS STRING.</b> This contains an advert/scan report.

#### Example:

```
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
```

## 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*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>AdvRpt</b>	<b>byRef AdvRpt AS STRING.</b> The advert report onto which the 16-bit uuids AD record is added.
<b>Uuid1</b>	<b>byVal uuid1 AS INTEGER</b> 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.
<b>Uuid2</b>	<b>byVal uuid2 AS INTEGER</b> 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.
<b>Uuid3</b>	<b>byVal uuid3 AS INTEGER</b> 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.
<b>Uuid4</b>	<b>byVal uuid4 AS INTEGER</b> 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.
<b>Uuid5</b>	<b>byVal uuid5 AS INTEGER</b> 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.
<b>Uuid6</b>	<b>byVal uuid6 AS INTEGER</b> UUID in the range 0 to FFFF; if the value is outside that range, it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.

### Example:

```
//Example :: BleAdvAddUuid16.sb (See in BT900CodeSnippets.zip)
DIM advRpt$, rc
DIM discovMode : discovMode=0
DIM advAppearance : advAppearance = 1
DIM maxDevName : maxDevName = 10

rc = BleAdvRptInit(advRpt$, discovMode, advAppearance, maxDevName)

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

IF BleAdvRptAddUuid16(advRpt$,0x180F,0x180A, -1, -1, -1, -1)==0 THEN
```

```
PRINT "\nUUID Service List AD added"
ENDIF

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

#### Expected Output:

```
UUID Service List AD added
```

## 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.

#### BLEADVVRPTADDUUID128 (*advRpt*, *nUuidHandle*)

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

#### Example:

```
//Example :: BleAdvAddUuid128.sb (See in BT900CodeSnippets.zip)
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
```

## 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.

**BLEADVVRTAPPENDAD** (*advRpt, nTag, stData\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>AdvRpt</b>	<b>byRef AdvRpt AS STRING.</b> The advert report onto which the AD record is to be appended.
<b>nTag</b>	<b>byVal nTag AS INTEGER</b> nTag should be in the range 0 to FF and is the TAG field for the record.
<b>stData\$</b>	<b>byRef stData\$ AS STRING</b> This is an octet string which can be 0 bytes long. The maximum length is governed by the space available in AdvRpt, a maximum of 31 bytes long.

#### Example:

```
//Example :: BleAdvRptAppendAD.sb (See in BT900CodeSnippets.zip)
DIM scnRpt$,ad$
ad$="\01\02\03\04"

PRINT BleScanRptInit(scnRpt$)

IF BleAdvRptAppendAD(scnRpt$,0x31,ad$)==0 THEN //6 bytes will be used up in the report
    PRINT "\nAD with data '";ad$;"' was appended to the advert report"
ENDIF
```

#### Expected Output:

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

## BleAdvRptsCommit

### FUNCTION

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

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

#### *BLEADVRPTSCOMMIT(advRpt, scanRpt)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>advRpt</b>	<b>byRef advRpt AS STRING.</b> The most recent advert report.
<b>scanRpt</b>	<b>byRef scanRpt AS STRING.</b> The most recent scan report.

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

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

### Example:

```
//Example :: BleAdvRptsCommit.sb (See in BT900CodeSnippets.zip)
DIM advRpt$ : advRpt$=""
DIM scRpt$ : scRpt$=""
DIM discovMode : discovMode = 0
DIM advApprnce : advApprnce = 1
DIM maxDevName : maxDevName = 10

PRINT BleAdvRptInit(advRpt$, discovMode, advApprnce, maxDevName)
PRINT BleAdvRptAddUuid16(advRpt$, 0x180F,0x180A, -1, -1, -1, -1)
PRINT BleAdvRptsCommit(advRpt$, scRpt$)

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

### Expected Output:

000

## Scanning Functions

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

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

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

The scan procedure ends when it times out (timeout parameter is supplied when scanning is initiated) or when 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:

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

- **EVBLE\_ADV\_REPORT** – Received when an advert has been successfully cached in a ring buffer. The handler should call the function BleScanGetAdvReport() repeatedly to read all the advert reports that have been cached until the cache is empty, otherwise there is a risk that advert reports will be discarded. The output parameter nDiscarded returns the number of discarded reports, if any.
- **EVBLE\_FAST\_PAGED** – Received when a peripheral has sent an advert with the address of this module. The handler should stop scanning using BleScanStop() and then initiate a connection using BleConnect().

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

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

Active scanning means that for each advert received (if it is ADV\_IND or ADV\_DISCOVER\_IND) a SCAN\_REQ is sent to the advertising device so that the data in the scan response can be appended to the data that has already been received for the advert.

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

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

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

### BLESCANSTART (*scanTimeoutMs*, *nFilterHandle*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>scanTimeoutMs</i></b>	<b>byVAL <i>scanTimeoutMs</i> AS INTEGER.</b> 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 <i>smart</i> BASIC application. Valid range is 0 to 65535000 milliseconds (about 18 hours). If 0 is supplied, a timer is not started and scanning can only be stopped by calling either BleScanAbort() or Ble ScanStop().
<b><i>nFilterHandle</i></b>	<b>byVAL <i>nFilterHandle</i> AS INTEGER</b> Specifies the whitelist handle to use when scanning, passing 0 will disable the use of whitelist.

#### Example:

```
//Example :: BleScanStart.sb (See in BT900CodeSnippets.zip)
DIM rc

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

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

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

ONEVENT EVBLE_SCAN_TIMEOUT CALL HndlrScanTO

WAITEVENT
```

#### Expected Output:

```
Scanning
Scan timeout
```

## BleScanAbort

### FUNCTION

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

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

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

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

### BLESCANABORT()

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

### Example:

```
//Example :: BleScanAbort.sb (See in BT900CodeSnippets.zip)
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
```

## BleScanStop

### FUNCTION

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

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

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

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

### BLESCANSTOP()

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

#### Example:

```
//Example :: BleScanStop.sb (See in BT900CodeSnippets.zip)
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 GetTickCountSince(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
ENDIF
```

#### Expected Output:

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

## BleScanFlush

### FUNCTION

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

#### BLESCANFLUSH()

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

#### Example:

```
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 GetTickCountSince(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

'//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.
```

## 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:

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

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

### BLESCANCONFIG (configID,configValue)

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

#### Example:

```
//Example :: BleScanConfig.sb (See in BT900CodeSnippets.zip)
DIM rc, startTick

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

PRINT "\n\nSetting new parameters..."

rc = BleScanConfig(0, 100) //set scan interval to 100
rc = BleScanConfig(1, 50) //set scan window to 50
rc = BleScanConfig(2, 0) //set scan type to passive
rc = BleScanConfig(3, 3) //set report cache size

PRINT "\n\n--- New Parameters:"
PRINT "\nScan Interval: "; SysInfo(2150) //get current scan interval
PRINT "\nScan Window: "; SysInfo(2151) //get current scan window
PRINT "\nScan Type: ";
IF SysInfo(2152)==0 THEN //get current scan type
    PRINT "Passive"
ELSE
```

```
PRINT "Active"
ENDIF
PRINT "\nReport Cache Size: "; SysInfo(2153) //get report cache size
```

#### Expected Output:

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

Setting new parameters..

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

## BleScanGetAdvReport

### FUNCTION

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

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

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

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

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

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>periphAddr\$</b>	<b>byREF periphAddr\$ AS STRING</b> On return, this parameter is updated with the address of the peripheral that sent the advert.
<b>advData\$</b>	<b>byREF advData \$ AS STRING</b> On return, this parameter is updated with the data payload of the advert which consists of multiple AD elements.

<b><i>nDiscarded</i></b>	<b>byREF nDiscarded AS INTEGER</b> On return, this parameter is updated with the number of adverts that were discarded because there was no space in the internal queue.
<b><i>nRssi</i></b>	<b>byREF nRssi AS INTEGER</b> On return, this parameter is updated with the RSSI as reported by the stack for that advert. <b>Note:</b> This is NOT a value that is sent by the peripheral but a value that is calculated by the receiver in this module.

**Note:** This code snippet was tested with another BT900 running the iBeacon app (see in smartBASIC\_Sample\_Apps folder) on peripheral firmware.

**Example:**

```
//Example :: BleScanGetAdvReport.sb (See in BT900CodeSnippets.zip)
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
ENDFUNC
```

```
ENDWHILE

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

ONEVENT EVBLE_SCAN_TIMEOUT CALL HndlrScanTO
ONEVENT EVBLE_ADV_REPORT CALL HndlrAdvRpt

WAITEVENT
```

### Expected Output:

```
Scanning

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

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

--- No more adverts in cache

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

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

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

## BleScanGetAdvReportEx

As with BleScanGetAdvReport but with a extra channel parameter, in the BT900 the channel is unknown and always a constant '3' value.

### BLESCANGETADVREPORTEX (*periphAddr\$, advData\$, nDiscarded, nRssi, nChannel*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>periphAddr\$</i></b>	<b>byREF <i>periphAddr\$</i> AS STRING</b> On return, this parameter is updated with the address of the peripheral that sent the advert.
<b><i>advData\$</i></b>	<b>byREF <i>advData \$</i> AS STRING</b> On return, this parameter is updated with the data payload of the advert which consists of multiple AD elements.
<b><i>nDiscarded</i></b>	<b>byREF <i>nDiscarded</i> AS INTEGER</b> On return, this parameter is updated with the number of adverts that were discarded because there was no space in the internal queue.
<b><i>nRssi</i></b>	<b>byREF <i>nRssi</i> AS INTEGER</b> On return, this parameter is updated with the RSSI as reported by the stack for that advert. <b>Note:</b> This is NOT a value that is sent by the peripheral but a value that is calculated by the receiver in this module.
<b><i>nChannel</i></b>	<b>byREF <i>nChannel</i> AS INTEGER</b> On return, this parameter is always '3' for unknown as the BT900 does not have access to channel information.

## 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\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nIndex</i></b>	<b>byVAL <i>nIndex</i> AS INTEGER</b> This is a zero-based index of the AD element that is copied into the output data parameter ADval\$.
<b><i>rptData\$</i></b>	<b>byREF <i>rptData\$</i> AS STRING.</b> 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).
<b><i>nADTag</i></b>	<b>byREF <i>nADTag</i> AS INTEGER</b>



	When the nth index is found, the single byte tag value for that AD element is returned in this parameter.
<b>ADval\$</b>	<b>byREF ADval\$ AS STRING</b> When the nth index is found, the data excluding single byte the tag value for that AD element is returned in this parameter.

**Example:**

```
//Example :: BleAdvGetADbyIndex.sb (See in BT900CodeSnippets.zip)
DIM rc, ad1$, ad2$, fullAD$, nADTag, ADval$

'//AD with length = 6 bytes, tag = 0xDD
ad1$="\06\DD\11\22\33\44\55"

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

fullAD$ = ad1$ + ad2$
PRINT "\n\n"; Strhexize$(fullAD$);"\n"

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

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

'//Will fail because there are only 2 AD elements
rc=BleGetADbyIndex(2, fullAD$ , nADTag, ADval$)
IF rc==0 THEN
    PRINT "\nThird AD element with tag 0x"; INTEGER.H'nADTag ;" is ";StrHexize$(ADval$)
ELSE
    PRINT "\nError reading AD: "; INTEGER.H'rc
ENDIF
```

### Expected Output:

```
06DD112233445507EEAABBCCDDEEFF

First AD element with tag 0x000000DD is 1122334455
Second AD element with tag 0x000000EE is AABBCCDDEEFF
Error reading AD: 00006060
```

## 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.

### BLEGETADBYTAG (*rptData\$*, *nADtag*, *ADval\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>rptData\$</i></b>	<b>byREF <i>rptData\$</i> AS STRING.</b> 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).
<b><i>nADtag</i></b>	<b>byVAL <i>nADtag</i> AS INTEGER</b> This parameter specifies the single byte tag value for the AD element that is to returned in the <i>ADval\$</i> parameter. Only the first instance can be catered for. If multiple instances are suspected, then use BleAdvADbyIndex() to extract it.
<b><i>ADval\$</i></b>	<b>byREF <i>ADval\$</i> AS STRING</b> When the nth index is found, the data excluding single byte the tag value for that AT element is returned in this parameter.

### Example:

```
//Example :: BleAdvGetADbyIndex.sb (See in BT900CodeSnippets.zip)
DIM rc, ad1$, ad2$, fullAD$, nADtag, ADval$

'//AD with length = 6 bytes, tag = 0xDD
ad1$="\06\DD\11\22\33\44\55"

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

```
fullAD$ = ad1$ + ad2$
PRINT "\n\n"; Strhexize$(fullAD$);"\n"

nADTag = 0xDD

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

nADTag = 0xEE

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

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

#### Expected Output:

```
06DD112233445507EEAABBCCDDEEFF

AD element with tag 0x000000DD is 1122334455
AD element with tag 0x000000EE is AABBCCDDEEFF
Error reading AD: 00006060
```

## BleScanGetPagerAddr

### FUNCTION3

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*)

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

#### Example:

```
//Example :: BleScanGetPagerAddr.sb (See in BT900CodeSnippets.zip)
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
```

## 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

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

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

## 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.

The BT900 has a resolution of ms for the Supervision timeout and Intervals, if you specify a value between whole ms it will be rounded.

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 )

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>periphAddr\$</b>	<b>byRef periphAddr\$ AS STRING</b> The Bluetooth address of the device to connect to which <b>MUST</b> be properly formatted and is exactly seven bytes long.
<b>connTimeoutMs</b>	<b>byVal connTimeoutMs AS INTEGER.</b> 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 <i>smart</i> BASIC application.
<b>minConnIntUs</b>	<b>byVal minConnIntUs AS INTEGER.</b> The minimum connection interval in microseconds.
<b>maxConnIntUs</b>	<b>byVal maxConnIntUs AS INTEGER.</b> The maximum connection interval in microseconds
<b>nSuprToutUs</b>	<b>byVal nSuprToutUs AS INTEGER.</b> The link supervision timeout for the connection in microseconds.

### Example:

```
//Example :: BleConnect.sb (See in BT900CodeSnippets.zip)
DIM rc, periphAddr$

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

IF rc==0 THEN
    PRINT "\nScanning"
```

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

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

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

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

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

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

'//This handler will be called when a disconnection happens
```

```
FUNCTION HndlrDiscon(nCtx, nRsn)
ENDFUNC 0

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

WAITEVENT
```

### Expected Output:

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

## 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 ()

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

### Example:

```
//Example :: BleConnectCancel.sb (See in BT900CodeSnippets.zip)
DIM rc, periphAddr$

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



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

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

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

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

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

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

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

ONEVENT EVBLE_ADV_REPORT CALL HndlrAdvRpt

WAITEVENT
```

### Expected Output:

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

## 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:

<b>Multi-link Connection Interval Periodicity</b>	30 milliseconds
<b>Scan Interval</b>	120 milliseconds
<b>Scan Window</b>	60 milliseconds
<b>Slave Latency</b>	0

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

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

### BLECONNECTCONFIG (configID,configValue)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
<b>Arguments:</b>		
<b>configID</b>	<b>byVal configID AS INTEGER.</b>	
	The following are the values to update:	
	0	Scan interval in milliseconds (range 0..10240)
	1	Scan Window in milliseconds (range 0..10240)
	2	Slave Latency (0..1000)
	5	Multi-Link Connection Interval Periodicity (20..200)
	6	Minimum connection length in ms
	7	Maximum connection length in ms
	8	Manual control of master connection parameters. 0 or 1.

	For all other configID values, the function returns an error.
<b>configValue</b>	<b>byVal configValue AS INTEGER.</b> This contains the new value to set in the parameters identified by configID.

**Example:**

```
//Example :: BleConnectConfig.sb (See in BT900CodeSnippets.zip)
DIM rc, startTick

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

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

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

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

## Expected Output:

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

Setting new parameters...

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

## BleDisconnect

### FUNCTION

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

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

### BLEDISCONNECT (nConnHandle)

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

### Example:

```
//Example :: BleDisconnect.sb (See in BT900CodeSnippets.zip)
DIM addr$ : addr$=""
DIM rc

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

```

PRINT "\nDisconnected ";nCtx;"\n"
EXITFUNC 0
ENDSELECT
ENDFUNC 1

ONEVENT EVBLEMSG      CALL HndlrBleMsg

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

WAITEVENT

```

#### Expected Output:

```

Adverts Started

New Connection 35800
Disconnected 3580

```

## BleSetCurConnParms

### FUNCTION

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

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

#### BLESETCURCONNPparms (nConnHandle, nMinIntUs, nMaxIntUs, nSuprToutUs, nSlaveLatency)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nConnHandle</b>	<b>byVal nConnHandle AS INTEGER.</b> Specifies the handle of the connection that must have the connection parameters changed.
<b>nMinIntUs</b>	<b>byVal nMinIntUs AS INTEGER.</b> The minimum acceptable connection interval in microseconds.
<b>nMaxIntUs</b>	<b>byVal nMaxIntUs AS INTEGER.</b> The maximum acceptable connection interval in microseconds.
<b>nSuprToutUs</b>	<b>byVal nSuprToutUs AS INTEGER.</b> The link supervision timeout for the connection in microseconds. It should be greater than the slave latency times that granted the connection interval.

<b><i>nSlaveLatency</i></b>	<b>byVal nSlaveLatency AS INTEGER.</b> The number of connection interval polls that the peripheral may ignore. This times the connection interval shall not be greater than the link supervision timeout.
-----------------------------	--

**Note:** The BT900 has a resolution of ms for the Supervision timeout and Intervals, if you specify a value between whole ms it will be rounded.

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

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

#### Example:

```
//Example :: BleSetCurConnParams.sb (See in BT900CodeSnippets.zip)
DIM rc
DIM addr$ : addr$=""

FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
  DIM intrvl, sprvTo, slat

  SELECT nMsgId
    CASE 0 //BLE_EVBLEMSGID_CONNECT
      PRINT "\n --- New Connection : ", nCtx
      rc=BleGetCurConnParams(nCtx, intrvl, sprvTo, slat)
      IF rc==0 THEN
        PRINT "\nConn Interval", intrvl
        PRINT "\nConn Supervision Timeout", sprvTo
        PRINT "\nConn Slave Latency", slat
        PRINT "\n\nRequest new parameters"
        //request connection interval in range 50ms to 75ms and link
        //supervision timeout of 4seconds with a slave latency of 19
        rc = BleSetCurConnParams(nCtx, 50000, 75000, 4000000, 19)
      ENDIF
    CASE 1 //BLE_EVBLEMSGID_DISCONNECT
      PRINT "\n --- Disconnected : ", nCtx
  EXITFUNC 0
```

```
CASE 14 //BLE_EVBLEMSGID_CONN_PARMS_UPDATE
    rc=BleGetCurConnParms (nCtx,intrvl,sprvto,slat)
    IF rc==0 THEN
        PRINT "\n\nConn Interval",intrvl
        PRINT "\nConn Supervision Timeout",sprvto
        PRINT "\nConn Slave Latency",slat
    ENDIF
CASE 15 //BLE_EVBLEMSGID_CONN_PARMS_UPDATE_FAIL
    PRINT "\n ??? Conn Parm Negotiation FAILED"
CASE ELSE
    PRINT "\nBle Msg",nMsgId
ENDSELECT
ENDFUNC 1

ONEVENT EVBLEMSG CALL HandlerBleMsg

IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "\nAdverts Started\n"
    PRINT "\nMake a connection to the BT900"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

### Expected Output (Unsuccessful Negotiation):

```
Adverts Started

Make a connection to the BT900
--- 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 BT900
--- 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.

## BleGetCurConnParams

### 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.

### BLEGETCURCONNP\_PARAMS (nConnHandle, nIntervalUs, nSuprToutUs, nSlaveLatency)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nConnHandle</b>	<b>byVal nConnHandle AS INTEGER.</b> Specifies the handle of the connection to read the connection parameters of
<b>nIntervalUs</b>	<b>byRef nIntervalUs AS INTEGER.</b> The current connection interval in microseconds
<b>nSuprToutUs</b>	<b>byRef nSuprToutUs AS INTEGER.</b> The current link supervision timeout in microseconds for the connection.
<b>nSlaveLatency</b>	<b>byRef nSlaveLatency AS INTEGER.</b> 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. <b>Note:</b> See <a href="#">Note on Slave Latency</a> .

See previous example.



## BleConnMgrUpdCfg

### FUNCTION

This function is used to initialise the connection manager for slave/peripheral role. Setting all values to zero will disable the retry statemachine for manual control.

#### BLECONNMNGRUPDCFG (*nConnUpdateFirstDelay*, *nConnUpdateNextDelay*, *nConnUpdateMaxRetry*)

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

#### Example:

```
dim rc
#define CONN_UPD_FIRST_DELAY 500
#define CONN_UPD_NEXT_DELAY 800
#define CONN_UPD_MAX_RETRY 8

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

#### Expected Output:

```
Connection manager successfully initialised
```

## Whitelist Management Functions

This section describes routines which are used to manage whitelists.

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

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

A trusted device by definition will have supplied its IRK key when the pairing and bonding happened in the past. Hence treat this group of functions as a means of creating, maintaining and destroying that list of addresses and IRKS.

The operation that enables whitelisting is the function that starts advertising and scanning. So refer to the functions `BleAdvertStart()` and `BleScanStart()`, although note that both these functions are only available depending on the role capability.

---

**Note:** The BT900 currently does not resolve IRKs, it is advisable only to rely on filtering IEEE and Random Static addresses.

---

## BleWhitelistCreate

### FUNCTION

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

#### BLEWHITELISTCREATE(*hWlist*, *nMaxAddrs*, *nMaxIrks*, *nPktFilterMask*)

<b>Returns</b>	<p>INTEGER, a result code.</p> <p><b>Typical value:</b></p> <p>0x0000 indicates a successful operation</p> <p>0x605E indicates too many whitelists already created.</p>
<b>Arguments</b>	
<b><i>hWlist</i></b>	<p><b>byRef <i>hWlist</i> AS INTEGER.</b></p> <p>If an empty whitelist is successfully created then this will be updated with a valid handle. If not then this will contain -1 (0xFFFFFFFF)</p>
<b><i>nMaxAddrs</i></b>	<p><b>byVal <i>nMaxAddrs</i> AS INTEGER.</b></p> <p>Maximum addresses that will be stored in this whitelist</p>
<b><i>nMaxIrks</i></b>	<p><b>byVal <i>nMaxIrks</i> AS INTEGER.</b></p> <p>Maximum Identity Resolving Keys (IRKs) that will be stored in this whitelist</p>
<b><i>nPktFilterMask</i></b>	<p><b>byVal <i>nPktFilterMask</i> AS INTEGER.</b></p> <p>This is a bit mask which specifies what type of incoming packets this list will apply to, as follows:-</p> <p>Bit 0 : Set to 1 for Scan Request packets</p> <p>Bit 1 : Set to 1 for Connection Request packets</p> <p>Bit 2 : Set to 1 for Advert Report Packets</p> <p>Bits 3 to 31 : reserved for future use</p> <p><b>Note:</b> If all bits are 0, then a default mask of 3 is used for the BL600</p>
<b>Interactive Command</b>	No

```
//Example :: BleWhitelist.sb
```

```
DIM rc,conHndl,hWlist, val
```

```
DIM addr$ : addr$=""
```

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

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

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

//=====
// Ble event handler
//=====

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

//=====
// This handler is called when there is an advert report waiting to be read
//=====

function HandlerAdvRpt () as integer
    dim ad$,dta$,ndisc,rsi
    rc = BleScanGetAdvReport(ad$,dta$,ndisc,rsi)
    while rc==0
        print "\nADV: ";strhexize$(ad$); " ";strhexize$(dta$); " ";ndisc; " ";rsi
        rc = BleScanGetAdvReport(ad$,dta$,ndisc,rsi)
    endwhile
endfunc 1

//=====
// This handler is called when there is an advert report waiting to be read
//=====

sub WhiteListInit()
    //set invalid whitelist handle
    hWlist=-1
    //now check maximum whitelists that can be defined and for that valid handle
    //is not required
endsub
```

```
rc=BleWhiteListInfo(hWlist,0, val) //get max number of whitelists allows
AssertRC(100)
print "\n Max allowed whitelists = "; val

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

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

ONEVENT EVBLEMSG          CALL HndlrBleMsg
OnEvent EVBLE_ADV_REPORT  CALL HandlerAdvRpt

//Initiliase a whitelist
WhiteListInit()
```

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

//Wait for events
WAITEVENT

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

BLEWHITELISTCREATE is an extension function.

## BleWhitelistDestroy

### FUNCTION

This function is used to clear an existing whitelist identified by a valid handle previously returned from BleWhitelistCreate() so that new addresses and Identity Resolving Keys (IRKs) can be added.

#### BLEWHITELISTDESTROY (hWlist)

<b>Returns</b>	INTEGER, a result code. <b>Typical value:</b> 0x0000 (indicates a successful operation)
<b>Arguments</b>	
<b>hWlist</b>	<b>byRef hWlist AS INTEGER.</b> This is the handle of the whitelist and is passed as a reference so that on exit it will have an invalid handle value so cannot be used inadvertently. The handle will have been returned by BleWhitelistCreate()
<b>Interactive Command</b>	No

For example, see description of [BleWhitelistCreate\(\)](#) above.

BLEWHITELISTDESTROY is an extension function.

## BleWhitelistClear

### FUNCTION

This function is used to clear an existing whitelist identified by a valid handle previously returned from BleWhitelistCreate() so that new addresses and Identity Resolving Keys (IRKs) can be added.

#### BLEWHITELISTCLEAR (hWlist)

<b>Returns</b>	INTEGER, a result code. <b>Typical value:</b> 0x0000 (indicates a successful operation)
<b>Arguments</b>	

<b><i>hWlist</i></b>	<b>byVal <i>hWlist</i> AS INTEGER.</b> This is the handle of the whitelist to clear and will have been returned by BleWhitelistCreate()
<b>Interactive Command</b>	No

For example, see description of [BleWhitelistCreate\(\)](#) above.

BLEWHITELISTCLEAR is an extension function.

### BleWhitelistSetFilter

#### FUNCTION

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

#### BLEWHITELISTSETFILTER (*hWlist*, *nPktFilterMask*)

<b>Returns</b>	INTEGER, a result code. <b>Typical value:</b> 0x0000 (indicates a successful operation)
<b>Arguments</b>	
<b><i>hWlist</i></b>	<b>byRef <i>hWlist</i> AS INTEGER.</b> This is the handle of the whitelist and will have been returned by BleWhitelistCreate()
<b><i>nPktFilterMask</i></b>	<b>byVal <i>nPktFilterMask</i> AS INTEGER.</b> This is a bit mask which specifies what type of incoming packets this list will apply to, as follows:- Bit 0 : Set to 1 for Scan Request packets Bit 1 : Set to 1 for Connection Request packets Bit 2 : Set to 1 for Advert Report Packets Bits 3 to 31 : reserved for future use
<b>Interactive Command</b>	No

For example, see description of [BleWhitelistCreate\(\)](#) above.

BLEWHITELISTSETFILTER is an extension function.

### BleWhitelistAddAddr

#### FUNCTION

This function is used to add a 7 byte mac address to the whitelist identified by the handle supplied. The function will automatically check if the mac address is trusted by interrogating the trusted device database and if it is, then the address stored there along with the IRK is added instead of the address supplied.

#### BLEWHITELISTADDADDR (*hWlist*, *addr\$*)

<b>Returns</b>	INTEGER, a result code. <b>Typical value:</b> 0x0000 (indicates a successful operation)
<b>Arguments</b>	

<b><i>hWlist</i></b>	<b>byVal <i>hWlist</i> AS INTEGER.</b> This is the handle of the whitelist and will have been returned by <code>BleWhitelistCreate()</code>
<b><i>addr\$</i></b>	<b>byRef <i>addr\$</i> AS STRING.</b> This is the address that is to be added to the whitelist. It will be checked for presence in trusted device database and if trusted, the IRK will also be added automatically to the whitelist
<b>Interactive Command</b>	No

For example, see description of [BleWhitelistCreate\(\)](#) above.

BLEWHITELISTADDADDR is an extension function.

## BleWhitelistAddIndex

### FUNCTION

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

#### BLEWHITELISTADDINDEX (*hWlist*, *nIndex*)

<b>Returns</b>	INTEGER, a result code. <b>Typical value:</b> 0x0000 (indicates a successful operation)
<b>Arguments</b>	
<b><i>hWlist</i></b>	<b>byVal <i>hWlist</i> AS INTEGER.</b> This is the handle of the whitelist and will have been returned by <code>BleWhitelistCreate()</code>
<b><i>nIndex</i></b>	<b>byVal <i>nIndex</i> AS INTEGER.</b> This is the Nth index (zero based) of the record in the trusted device database to add to the whitelist. The IRK will also be added automatically to the whitelist. The index is the same entity per the function <a href="#">BleBondMngrGetInfo()</a>
<b>Interactive Command</b>	No

For example, see description of [BleWhitelistCreate\(\)](#) above.

BLEWHITELISTADDINDEX is an extension function.

## BleWhitelistInfo

### FUNCTION

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

#### BLEWHITELISTINFO (*hWlist*, *nInfoID*, *nValue*)

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

<b><i>nInfoID</i></b>	<p><b>byVal <i>nInfoID</i> AS INTEGER.</b> This is ID of the information to be returned as follow:-  0 : maximum number of whitelists (hWlist is ignored)  1 : maximum number of mac addresses (hWlist is ignored)  2 : maximum number of IRKs (hWlist is ignored)  101 : current number of addresses added  102 : current number of IRKs added</p> <p><b>Note:</b> For 101 and 102, the values are cleared to 0 if BleWhitelistClear() is called.</p>
<b><i>nValue</i></b>	<p><b>byRef <i>nValue</i> AS INTEGER.</b> The information value is returned in this variable</p>
<b>Interactive Command</b>	No

For example, see description of [BleWhitelistCreate\(\)](#) above.

BLEWHITELISTINFO is an extension function.

## GATT Server Functions

This section describes all functions related to creating and managing services that collectively define a GATT table from a GATT server role perspective. These functions allow the developer to create any service that has is described and adopted by the Bluetooth SIG or any custom service that implements some custom unique functionality, within resource constraints such as the limited RAM and FLASH memory that is exist in the module.

A GATT table is a collection of adopted or custom services which, in turn, are a collection of adopted or custom characteristics. By definition, an adopted service cannot contain custom characteristics but the reverse is possible where a custom service can include both adopted and custom characteristics.

Descriptions of services and characteristics are available in the Bluetooth Specification v4.0 or newer. Because these descriptions are concise and difficult to understand, the following section attempts to familiarise you with these concepts using the *smart*BASIC programming environment perspective.

To help understand service and characteristic better, think of a characteristic as a container (or a pot) of data where the pot comes with space to store the data and a set of properties that are officially called Descriptors in the BT spec. In the pot analogy, think of a descriptor as the color of the pot, whether it has a lid, whether the lid has a lock, whether it has a handle or a spout, etc. For a full list of these descriptors online, see <http://developer.bluetooth.org/GATT/descriptors/Pages/DescriptorsHomePage.aspx> . These descriptors are assigned 16-bit UUIDs (value 0x29xx) and are referenced in some of the *smart*BASIC API functions if you decide to add those to your characteristic definition.

You can consider a service as a carrier bag to hold a group of related characteristics together where the printing on the carrier bag is a UUID. From a *smart*BASIC developer's perspective, a set of characteristics is what you need to manage and the concept of service is only required at GATT table creation time.

A GATT table can have many services, each containing one or more characteristics. The difference between services and characteristics is expedited using an identification number called a UUID (Universally Unique Identifier) which is a 128-bit (16-byte) number. Adopted services or characteristics have a 16-bit (2-byte) shorthand identifier (which is an offset plus a base 128-bit UUID defined and reserved by the Bluetooth SIG); custom service or characteristics have the full 128-bit UUID. The logic behind this is that a 16-bit UUID implies



that a specification has been published by the Bluetooth SIG whereas using a 128-bit UUID does NOT require any central authority to maintain a register of those UUIDs or specifications describing them.

The lack of the requirement for a central register is important to understand in the sense that, if a custom service or characteristic must be created, the developer can use any publicly available UUID (sometimes also known as GUID) generation utility.

These utilities use entropy from the real world to generate a 128-bit random number that has an extremely low probability to be the same as that generated by someone else at the same time or in the past or future.

As an example, at the time of writing this document, the following website

<http://www.guidgenerator.com/online-guid-generator.aspx> offers an immediate UUID generation service, although it uses the term GUID. From the GUID Generator website:

***How unique is a GUID?***

*128-bits is big enough and the generation algorithm is unique enough that if 1,000,000,000 GUIDs per second were generated for 1 year the probability of a duplicate would be only 50%. Or if every human on Earth generated 600,000,000 GUIDs there would only be a 50% probability of a duplicate.*

This extremely low probability of generating the same UUID is why there is no need for a central register maintained by the Bluetooth SIG for custom UUIDs.

Please note that Laird does not guarantee that the UUID generated by this website or any other utility is unique. It is left to the judgement of the developer whether to use it or not.

---

**Note:** If the developer intends to create custom services and/or characteristics then it is recommended that a single UUID is generated and used from then on as a 128-bit (16 byte) company/developer unique base along with a 16-bit (2-byte) offset, in the same manner as the Bluetooth SIG.

This allows up to 65536 custom services and characteristics to be created, with the added advantage that it is easier to maintain a list of 16-bit integers.

The main reason for avoiding more than one long UUID is to keep RAM usage down given that 16 bytes of RAM is used to store a long UUID. *smart*BASIC functions have been provided to manage these custom 2-byte UUIDs along with their 16-byte base UUIDs.

---

In this document, when a service or characteristic is described as adopted, it implies that the Bluetooth SIG published a specification which defines that service or characteristic and there is a requirement that any device claiming to support them has proof that the functionality has been tested and verified to behave as per that specification.

Currently there is no requirement for custom service and/or characteristics to have any approval. By definition, interoperability is restricted to the provider and implementer.

A service is an abstraction of some collectivised functionality which, if broken down further, would cease to provide the intended behaviour. Two examples in the BLE domain that have been adopted by the Bluetooth SIG are Blood Pressure Service and Heart Rate Service. Each have sub-components that map to characteristics.

Blood pressure is defined by a collection of data entities such as Systolic Pressure, Diastolic Pressure, and Pulse Rate. Likewise, a Heart Rate service has a collection which includes entities such as the Pulse Rate and Body Sensor Location.

A list of all the adopted services is at: <http://developer.Bluetooth.org/GATT/services/Pages/ServicesHome.aspx>. Laird recommends that, if you decide to create a custom service, it should be defined and described in a similar

fashion; your goal should be to get the Bluetooth SIG to adopt it for everyone to use in an interoperable manner.

These services are also assigned 16-bit UUIDs (value 0x18xx) and are referenced in some of the *smartBASIC* API functions described in this section.

Services, as described above, are a collection of one or more characteristics. A list of all adopted characteristics is found at: <http://developer.bluetooth.org/GATT/characteristics/Pages/CharacteristicsHome.aspx>. You should note that these descriptors are also assigned 16-bit UUIDs (value 0x2Axx) and are referenced in some of the API functions described in this section. Custom characteristics have 128-bit (16-byte) UUIDs and API functions are provided to handle those.

---

**Note:** If you intend to create a custom service or characteristic and adopt the recommendation of a single 16-byte base UUID so that the service can be identified using a 2-byte UUID, then allocate a 16-bit value which is not going to coincide with any adopted values to minimise confusion. Selecting a similar value is possible and legal given that the base UUID is different.

---

The remainder of this introduction focuses on the specifics of how to create and manage a GATT table from a perspective of the *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.

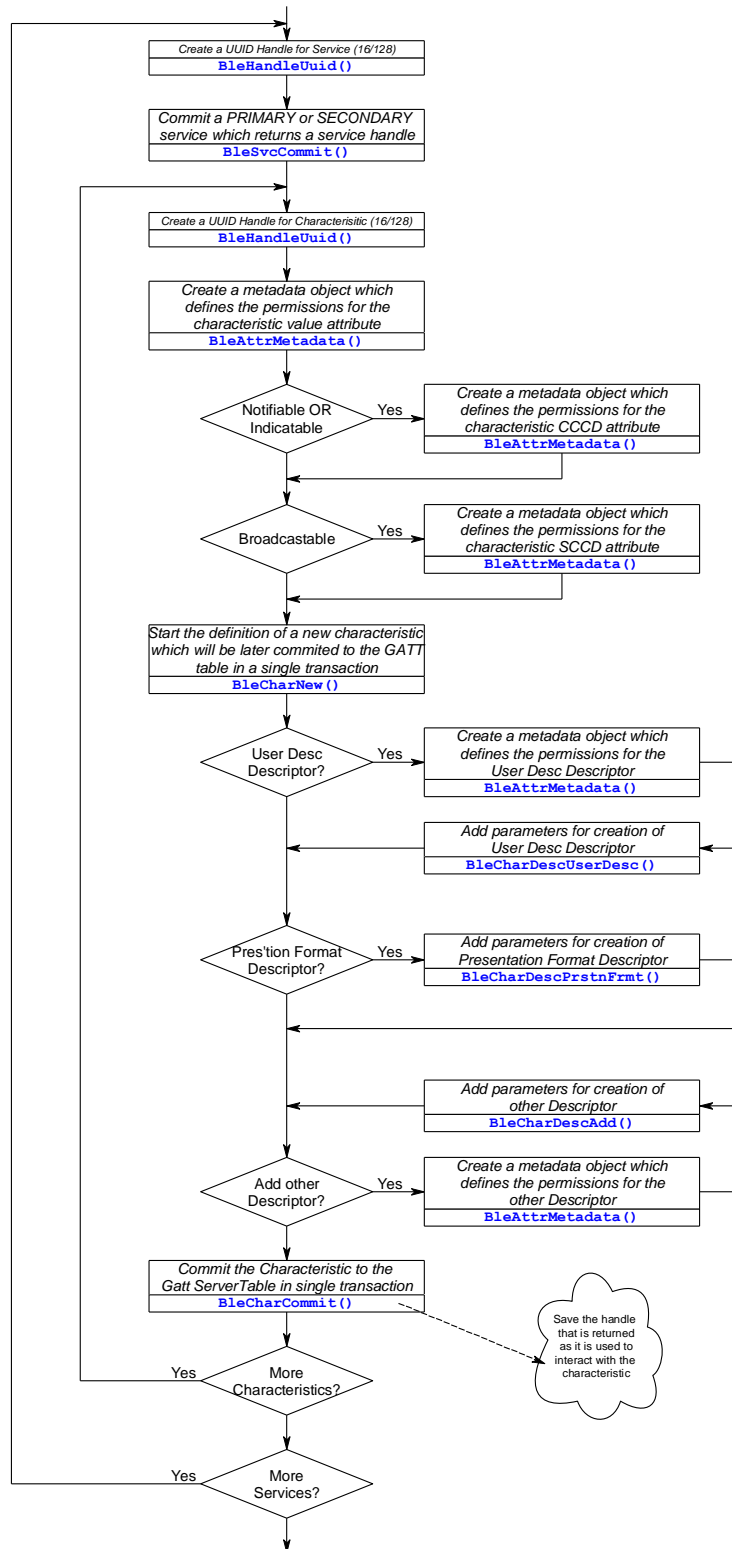
The GATT client (remote end of the wireless connection) must see those carrier bags to determine the groupings and, once it has identified the pots, it only needs to keep a list of references to the pots it is interested in. Once that list is made at the client end, it can 'throw away the carrier bag'.

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.

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:**

```
//Example :: ServicesAndCharacteristics.sb (See in BT900CodeSnippets.zip)

//=====
//Register two Services in the GATT Table. Service 1 with 2 Characteristics and
//Service 2 with 1 characteristic. This implies a total of 3 characteristics to
//manage.
//The characteristic 2 in Service 1 will not be readable or writable but only
//indicatable
//The characteristic 1 in Service 2 will not be readable or writable but only
//notifyable
//=====

DIM rc    //result code
DIM hSvc  //service handle
DIM mdAttr
DIM mdCccd
DIM mdSccd
DIM chProp
DIM attr$

DIM hChar11 // handles for characteristic 1 of Service 1
DIM hChar21 // handles for characteristic 2 of Service 1
DIM hChar12 // handles for characteristic 1 of Service 2

DIM hUuidS1 // handles for uuid of Service 1
DIM hUuidS2 // handles for uuid of Service 2
DIM hUuidC11 // handles for uuid of characteristic 1 in Service 1
DIM hUuidC12 // handles for uuid of characteristic 2 in Service 1
DIM hUuidC21 // handles for uuid of characteristic 1 in Service 2

//---Register Service 1
hUuidS1 = BleHandleUuid16(0x180D)
rc = BleServiceNew(BLE_SERVICE_PRIMARY, hUuidS1, hSvc)

//---Register Characteristic 1 in Service 1
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,10,0,rc)
mdCccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
mdSccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
```

```
chProp = BLE_CHAR_PROPERTIES_READ + BLE_CHAR_PROPERTIES_WRITE
hUuidC11 = BleHandleUuid16(0x2A37)
rc = BleCharNew(chProp, hUuidC11,mdAttr,mdCccd,mdSccd)
rc = BleCharCommit(shHrs,hrs$,hChar11)

//---Register Characteristic 2 in Service 1
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,10,0,rc)
mdCccd = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,2,0,rc)
mdSccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
chProp = BLE_CHAR_PROPERTIES_INDICATE
hUuidC12 = BleHandleUuid16(0x2A39)
rc = BleCharNew(chProp, hUuidC12,mdAttr,mdCccd,mdSccd)
attr$="\00\00"
rc = BleCharCommit(hSvc,attr$,hChar21)
rc = BleServiceCommit(hSvc)

//---Register Service 2 (can now reuse the service handle)
hUuidS2 = BleHandleUuid16(0x1856)
rc = BleServiceNew(BLE_SERVICE_PRIMARY, hUuidS2, hSvc)

//---Register Characteristic 1 in Service 2
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_NONE,BLE_ATTR_ACCESS_NONE,10,0,rc)
mdCccd = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,2,0,rc)
mdSccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
chProp = BLE_CHAR_PROPERTIES_NOTIFY
hUuidC21 = BleHandleUuid16(0x2A54)
rc = BleCharNew(chProp, hUuidC21,mdAttr,mdCccd,mdSccd)
attr$="\00\00\00\00"
rc = BleCharCommit(hSvc,attr$,hChar12)
rc = BleServiceCommit(hSvc)

//===The 2 services are now visible in the gatt table
```

Writes into a characteristic from a remote client are detected and processed as follows:

```
//-----
// To deal with writes from a GATT client into characteristic 1 of Service 1
// which has the handle hChar11
//-----

// This handler is called when there is a EVCHARVAL message
FUNCTION HandlerCharVal(BYVAL hChar AS INTEGER) AS INTEGER
```

```
DIM attr$
IF hChar == hChar11 THEN
    rc = BleCharValueRead(hChar11,attr$)
    print "Svc1/Char1 has been writen with = ";attr$

ENDIF
ENDFUNC 1

//enable characteristic value write handler
OnEvent EVCHARVAL      call HandlerCharVal

WAITEVENT
```

Assuming there is a connection and notify has been enabled, a value notification is expedited as follows:

```
//-----
// Notify a value for characteristic 1 in service 2
//-----
attr$="somevalue"
rc = BleCharValueNotify(hChar12,attr$)
```

Assuming there is a connection and indicate has been enabled, a value indication is expedited as follows:

```
//-----
// indicate a value for characteristic 2 in service 1
//-----

// This handler is called when there is a EVCHARHVC message
FUNCTION HandlerCharHvc(BYVAL hChar AS INTEGER) AS INTEGER
    IF hChar == hChar12 THEN
        PRINT "Svc1/Char2 indicate has been confirmed"
    ENDIF
ENDFUNC 1

//enable characteristic value indication confirm handler
OnEvent EVCHARHVC      CALL HandlerCharHvc

attr$="somevalue"
rc = BleCharValueIndicate(hChar12,attr$)
```

The rest of this section details all the *smart*BASIC functions that help create that framework.

## 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:

[http://developer.Bluetooth.org/GATT/services/Pages/ServiceViewer.aspx?u=org.Bluetooth.service.generic\\_access.xml](http://developer.Bluetooth.org/GATT/services/Pages/ServiceViewer.aspx?u=org.Bluetooth.service.generic_access.xml)

**BLEGAPSVCLINIT** (*deviceName*, *nameWritable*, *nAppearance*, *nMinConnInterval*, *nMaxConnInterval*, *nSupervisionTout*, *nSlaveLatency*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation
<b>Arguments:</b>	
<b><i>deviceName</i></b>	<p><b>byRef <i>deviceName</i> AS STRING</b> The name of the device (such as Laird_Thermometer) to store in the Device Name characteristic of the GAP service.</p> <p><b>Note:</b> When an advert report is created using BLEADVPTINIT(), 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.</p>
<b><i>nameWritable</i></b>	<p><b>byVal <i>nameWritable</i> AS INTEGER</b> If non-zero, the peer device is allowed to write the device name. Some profiles allow this to be made optional.</p>
<b><i>nAppearance</i></b>	<p><b>byVal <i>nAppearance</i> AS INTEGER</b> Field lists the external appearance of the device and updates the Appearance characteristic of the GAP service. Possible values: <a href="#">org.Bluetooth.characteristic.gap.appearance</a></p>
<b><i>nMinConnInterval</i></b>	<p><b>byVal <i>nMinConnInterval</i> AS INTEGER</b> 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 <i>nMaxConnInterval</i>.</p>
<b><i>nMaxConnInterval</i></b>	<p><b>byVal <i>nMaxConnInterval</i> AS INTEGER</b> 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 <i>nMinConnInterval</i>.</p>

<b><i>nSupervisionTimeout</i></b>	<p><b>byVal <i>nSupervisionTimeout</i> AS INTEGER</b></p> <p>The preferred link supervision timeout and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service.</p> <p>Range is between 100000 to 32000000 microseconds (rounded to the nearest 10000 microseconds).</p>
<b><i>nSlaveLatency</i></b>	<p><b>byVal <i>nSlaveLatency</i> AS INTEGER</b></p> <p>The preferred slave latency is the number of communication intervals that a slave may ignore without losing the connection and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service.</p> <p>This value must be smaller than (nSupervisionTimeout/ nMaxConnInterval) -1. i.e.  <math>nSlaveLatency &lt; (nSupervisionTimeout / nMaxConnInterval) - 1</math></p>

**Example:**

```
//Example :: BleGapSvcInit.sb (See in BT900CodeSnippets.zip)

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

dvcNme$= "Laird_TS"
nmeWrtble = 0           //Device name will not be writable by peer
apprnce = 768           //The device will appear as a Generic Thermometer
MinConnInt = 500000     //Minimum acceptable connection interval is 0.5 seconds
MaxConnInt = 1000000    //Maximum acceptable connection interval is 1 second
ConnSupTO = 4000000     //Connection supervisory timeout is 4 seconds
sL = 0                  //Slave latency--number of conn events that can be missed

rc=BleGapSvcInit(dvcNme$,nmeWrtble,apprnce,MinConnInt,MaxConnInt,ConnSupTO,sL)

IF !rc THEN
    PRINT "\nSuccess"
ELSE
    PRINT "\nFailed 0x"; INTEGER.H'rc //Print result code as 4 hex digits
ENDIF
```

**Expected Output:**

```
Success
```



## BleGetDeviceName\$

### FUNCTION

This function reads the device name characteristic value from the local GATT table. This value is the same as that supplied in BleGapSvcInit() if the 'nameWritable' parameter was 0, otherwise it may be different.

EVBLEMSG event is thrown with 'msgid' == 21 when the GATT client writes a new value and is the best time to call this function.

### BLEGETDEVICENAME\$ ()

<b>Returns</b>	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.
<b>Arguments</b>	None

### Example:

```
//Example :: BleGetDeviceName$.sb (See in BT900CodeSnippets.zip)

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

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

// Changing device name manually
dvcNme$= "My BT900"
nmeWrtble = 0
apprnce = 768
MinConnInt = 500000
MaxConnInt = 1000000
ConnSupTO = 4000000
sL = 0

rc = BleGapSvcInit(dvcNme$,nmeWrtble,apprnce,MinConnInt,MaxConnInt,ConnSupTO,sL)
PRINT "\n --- New DevName : "; BleGetDeviceName$()
```

### Expected Output:

```
--- DevName : LAIRD BT900
--- New DevName : My BT900
```

## BleSvcRegDevInfo

### FUNCTION

This function is used to register the Device Information service with the GATT server. The Device Information service contains nine characteristics as listed at the following website:

[http://developer.bluetooth.org/GATT/services/Pages/ServiceViewer.aspx?u=org.Bluetooth.service.device\\_information.xml](http://developer.bluetooth.org/GATT/services/Pages/ServiceViewer.aspx?u=org.Bluetooth.service.device_information.xml)

The firmware revision string is always set to **BT900:vW.X.Y.Z** where W,X,Y,Z are as per the revision information which is returned to the command AT I 4.

**BLESVCREGDEVINFO ( *manfName\$, modelNum\$, serialNum\$, hwRev\$, swRev\$, sysId\$, regDataList\$, pnpId\$* )**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>manfName\$</i></b>	<b>byVal <i>manfName\$</i> AS STRING</b> The device manufacturer. Can be set empty to omit submission.
<b><i>modelNum\$</i></b>	<b>byVal <i>modelNum\$</i> AS STRING</b> The device model number. Can be set empty to omit submission.
<b><i>serialNum\$</i></b>	<b>byVal <i>serialNum\$</i> AS STRING</b> The device serial number. Can be set empty to omit submission.
<b><i>hwRev\$</i></b>	<b>byVal <i>hwRev\$</i> AS STRING</b> The device hardware revision string. Can be set empty to omit submission.
<b><i>swRev\$</i></b>	<b>byVal <i>swRev\$</i> AS STRING</b> The device software revision string. Can be set empty to omit submission.
<b><i>sysId\$</i></b>	<b>byVal <i>sysId\$</i> AS STRING</b> 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.
<b><i>regDataList\$</i></b>	<b>byVal <i>regDataList\$</i> AS STRING</b> The device's regulatory certification data list as defined in the specification. It can be set as an empty string to omit submission.
<b><i>pnpId\$</i></b>	<b>byVal <i>pnpId\$</i> AS STRING</b> The device's plug and play ID as defined in the specification. Can be set empty to omit submission. Otherwise, it shall be exactly 7 octets long, where: Byte 0 := Vendor Id Source Byte 1,2 := Vendor Id (Byte 1 is LSB) Byte 3,4 := Product Id (Byte 3 is LSB) Byte 5,6 := Product Version (Byte 5 is LSB)

### Example:

```
//Example :: BleSvcRegDevInfo.sb (See in BT900CodeSnippets.zip)
```

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

manfNme$ = "Laird Technologies"
mdlNum$ = "BT900"
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
```

## BleHandleUuid16

### FUNCTION

This function takes an integer in the range 0 to 65535 and converts it into a 32-bit integer handle that associates the integer as an offset into the Bluetooth SIG 128-bit (16-byte) base UUID which is used for all adopted services, characteristics, and descriptors.

If the input value is not in the valid range, then an invalid handle (0) is returned.

The returned handle is treated by the developer as an opaque entity and no further logic is based on the bit content, apart from all zeros which represent an invalid UUID handle.

### BLEHANDLEUUID16 (nUuid16)

<b>Returns</b>	INTEGER, a nonzero handle shorthand for the UUID. Zero is an invalid UUID handle
<b>Arguments:</b>	
<b>nUuid16</b>	<b>byVal nUuid16 AS INTEGER</b> nUuid16 is first bitwise ANDed with 0xFFFF and the result is treated as an offset into the Bluetooth SIG 128 bit base UUID

#### Example:

```
//Example :: BleHandleUuid16.sb (See in BT900CodeSnippets.zip)
DIM uuid
```

```
DIM hUuidHRS

uuid = 0x180D //this is UUID for Heart Rate Service
hUuidHRS = BleHandleUuid16(uuid)

IF hUuidHRS == 0 THEN
    PRINT "\nFailed to create a handle"
ELSE
    PRINT "Handle for HRS Uuid is "; integer.h' hUuidHRS; "(";hUuidHRS;") "
ENDIF
```

#### Expected Output:

```
Handle for HRS Uuid is FE01180D (-33482739)
```

## 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\$*)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b><i>stUuid\$</i></b>	<b>byRef <i>stUuid\$</i> AS STRING</b> Any 16-byte string that was generated using a UUID generation utility that has enough entropy to ensure that it is random. The first byte of the string is the MSB of the UUID (big endian format).

#### Example:

```
//Example :: BleHandleUuid128.sb (See in BT900CodeSnippets.zip)
DIM uuid$, hUuidCustom

//create a custom uuid for my ble widget
uuid$ = "ced9d91366924a1287d56f2764762b2a"
```

```

uuid$ = StrDehexize$(uuid$)
hUuidCustom = BleHandleUuid128(uuid$)
IF hUuidCustom == 0 THEN
    PRINT "\nFailed to create a handle"
ELSE
    PRINT "Handle for custom Uuid is ";integer.h' hUuidCustom; "(";hUuidCustom;")"
ENDIF
// hUuidCustom now references an object which points to
// a base uuid = ced9d91366924a1287d56f2747622b2a (note 0's in byte position 2/3)
// and an offset = 0xd913

```

#### Expected Output:

```
Handle for custom Uuid is FC03D913 (-66856685)
```

## BleHandleUuidSibling

### FUNCTION

This function takes an integer in the range 0 to 65535 along with a UUID handle which had been previously created using BleHandleUuid16() or BleHandleUuid128() to create a new UUID handle. This handle references the same 128 base UUID as the one referenced by the UUID handle supplied as the input parameter.

The returned handle shall be treated by the developer as an opaque entity and no further logic shall be based on the bit content, apart from all zeroes (which represents an invalid UUID handle).

#### BLEHANDLEUUIDSIBLING (nUuidHandle, nUuid16)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>nUuidHandle</b>	<b>byVal nUuidHandle AS INTEGER</b> A handle that was previously created using either BleHandleUui16() or BleHandleUuid128().
<b>nUuid16</b>	<b>byVal nUuid16 AS INTEGER</b> A UUID value in the range 0 to 65535 which is treated as an offset into the 128-bit base UUID referenced by nUuidHandle.

#### Example:

```

//Example :: BleHandleUuidSibling.sb (See in BT900CodeSnippets.zip)
DIM uuid$ ,hUuid1, hUuid2 //hUuid2 will have the same base uuid as hUuid1

//create a custom uuid for my ble widget
uuid$ = "ced9d91366924a1287d56f2764762b2a"
uuid$ = StrDehexize$(uuid$)
hUuid1 = BleHandleUuid128(uuid$)
IF hUuid1 == 0 THEN

```

```
PRINT "\nFailed to create a handle"
ELSE
    PRINT "Handle for custom Uuid is ";integer.h' hUuid1;"(";hUuid1;")"
ENDIF
// hUuid1 now references an object which points to
// a base uuid = ced9000066924a1287d56f2747622b2a (note 0's in byte position 2/3)
// and an offset = 0xd913

hUuid2 = BleHandleUuidSibling(hUuid1,0x1234)
IF hUuid2 == 0 THEN
    PRINT "\nFailed to create a handle"
ELSE
    PRINT "\nHandle for custom sibling Uuid is ";integer.h'hUuid2;"(";hUuid2;")"
ENDIF
// hUuid2 now references an object which also points to
// the base uuid = ced9000066924a1287d56f2700004762 (note 0's in byte position 2/3)
// and has the offset = 0x1234
```

#### Expected Output:

```
Handle for custom Uuid is FC03D913 (-66856685)
Handle for custom sibling Uuid is FC031234 (-66907596)
```

## 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 `BleSvcAddIncludeSvc()` which is described immediately after this function.

---

This function now replaces BleSvcCommit() and marks the beginning of a service definition in the GATT server table. When the last descriptor of the last characteristic has been registered the service definition should be terminated by calling BleServiceCommit().

### BLESERVICENEW (nSvcType, nUuidHandle, hService )

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nSvcType</b>	<b>byVal nSvcType AS INTEGER</b> 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.
<b>nUuidHandle</b>	<b>byVal nUuidHandle AS INTEGER</b> 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().
<b>hService</b>	<b>byRef hService AS INTEGER</b> If the service attribute is created in the GATT table, then this contains a composite handle which references the actual attribute handle. This is then subsequently used when adding characteristics to the GATT table. If the function fails to install the service attribute for any reason, this variable will contain 0 and the returned result code will be non-zero.

#### Example:

```
//Example :: BleServiceNew.sb (See in BT900CodeSnippets.zip)

#DEFINE BLE_SERVICE_SECONDARY          0
#DEFINE BLE_SERVICE_PRIMARY            1

//-----
//Create a Health Thermometer PRIMARY service attribute which has a uuid of 0x1809
//-----

DIM hHtsSvc //composite handle for hts primary service
DIM hUuidHT : hUuidHT = BleHandleUuid16(0x1809) //HT Svc UUID Handle

IF BleServiceNew(BLE_SERVICE_PRIMARY,hUuidHT,hHtsSvc)==0 THEN
    PRINT "\nHealth Thermometer Service attribute written to GATT table"
    PRINT "\nUUID Handle value: ";hUuidHT
    PRINT "\nService Attribute Handle value: ";hHtsSvc
ELSE
    PRINT "\nService Commit Failed"
ENDIF

//-----
//Create a Battery PRIMARY service attribute which has a uuid of 0x180F
//-----

DIM hBatSvc //composite handle for battery primary service
```

```
//or we could have reused nHtsSvc
DIM hUuidBatt : hUuidBatt = BleHandleUuid16(0x180F) //Batt Svc UUID Handle

IF BleServiceNew(BLE_SERVICE_PRIMARY,hUuidBatt,hBatSvc)==0 THEN
    PRINT "\n\nBattery Service attribute written to GATT table"
    PRINT "\nUUID Handle value: ";hUuidBatt
    PRINT "\nService Attribute Handle value: ";hBatSvc
ELSE
    PRINT "\nService Commit Failed"
ENDIF
```

### Expected Output:

```
Health Thermometer Service attribute written to GATT table
UUID Handle value: -33482743
Service Attribute Handle value: 16

Battery Service attribute written to GATT table
UUID Handle value: -33482737
Service Attribute Handle value: 17
```

## BleServiceCommit

This function in the BT900 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/committed for that service.

### BLESERVICECOMMIT (hService)

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

See example for [BleCharCommit\(\)](#).

## BleSvcAddIncludeSvc

### FUNCTION

**Note:** This function is currently not available for use on this module

This function is used to add a reference to a service within another service. This is usually, but not necessarily, a SECONDARY service which is virtually identical to a PRIMARY service from the GATT server perspective. The only



difference is that, when a GATT client queries a device for all services, it does not receive mention of SECONDARY services.

When a GATT client encounters an INCLUDED SERVICE object when querying a particular service it performs a sub-procedure to get handles to all the characteristics that are part of that INCLUDED service.

This mechanism is provided to allow for a single set of characteristics to be shared by multiple primary services. This is most relevant if a characteristic is defined so that it can have only one instance in a GATT table but needs to be offered in multiple PRIMARY services. A typical implementation, where a characteristic is part of many PRIMARY services, installs that characteristic in a SECONDARY service ( see [BleSvcCommit\(\)](#) ) and then uses the function defined in this section to add it to all the PRIMARY services that want to have that characteristic as part of their group.

It is possible to include a service which is also a PRIMARY or SECONDARY service, which in turn can include further PRIMARY or SECONDARY services. The only restriction to nested includes is that there cannot be recursion.

**Note:** If a service has INCLUDED services, then they is installed in the GATT table immediately after a service is created using [BleSvcCommit\(\)](#) and before [BleCharCommit\(\)](#). The BT 4.0 specification mandates that any 'included service' attribute be present before any characteristic attributes within a particular service group declaration.

### ***BleSvcAddIncludeSvc (hService)***

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation
<b>Arguments:</b>	
<b><i>hService</i></b>	<b>byVal hService AS INTEGER</b> This argument contains a handle that was previously created using the function <a href="#">BleSvcCommit()</a> .

### **Example:**

```
//Example :: BleSvcAddIncludeSvc.sb (See in BT900CodeSnippets.zip)
#define BLE_SERVICE_SECONDARY          0
#define BLE_SERVICE_PRIMARY            1

//-----
//Create a Battery SECONDARY service attribure which has a uuid of 0x180F
//-----

dim hBatSvc //composite handle for batteru primary service
dim rc      //or we could have reused nHtsSvc
dim metaSuccess
DIM charMet : charMet = BleAttrMetaData(1,1,10,1,metaSuccess)
DIM s$ : s$ = "Hello" //initial value of char in Battery Service
DIM hBatChar

rc = 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)
```

## 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*, *flsVariableLen*, *resCode*)

<b>Returns</b>	INTEGER, a 32-bit opaque data object to be used in subsequent calls when adding Characteristics to a GATT table.	
<b>Arguments:</b>		
<b>nReadRights</b>	byVal nReadRights AS INTEGER This specifies the read rights and shall have one of the following values:	
	0	No access
	1	Open
	2	Encrypted with No Man-In-The-Middle (MITM) protection
	3	Encrypted with Man-In-The-Middle (MITM) protection
	4	Signed with No Man-In-The-Middle (MITM) protection (not available)
	5	Signed with Man-In-The-Middle (MITM) protection (not available)
	<b>Note:</b> In early releases of the firmware, 4 and 5 are not available.	
<b>nWriteRights</b>	byVal nWriteRights AS INTEGER This specifies the write rights and shall have one of the following values:	
	0	No access
	1	Open
	2	Encrypted with No Man-In-The-Middle (MITM) protection
	3	Encrypted with Man-In-The-Middle (MITM) protection
	4	Signed with No Man-In-The-Middle (MITM) protection (not available)
	5	Signed with Man-In-The-Middle (MITM) protection (not available)
	<b>Note:</b> In early releases of the firmware, 4 and 5 are not available.	
<b>nMaxDataLen</b>	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.	
<b>flsVariableLen</b>	byVal flsVariableLen 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	

<b>resCode</b>	<b>byRef resCode AS INTEGER</b> This variable is updated with a result code which is 0 if a metadata object was successfully returned by this call. Any other value implies a metadata object did not get created.
----------------	---

**Example:**

```
//Example :: BleAttrMetadata.sb (See in BT900CodeSnippets.zip)

DIM mdVal //metadata for value attribute of Characteristic
DIM mdCccd //metadata for CCCD attribute of Characteristic
DIM mdSccd //metadata for SCCD attribute of Characteristic
DIM rc

//++++
// Create the metadata for the value attribute in the characteristic
// and Heart Rate attribute has variable length
//++++

//There is always a Value attribute in a characteristic
mdVal=BleAttrMetadata(17,0,20,0,rc)
//There is a CCCD and SCCD in this characteristic
mdCccd=BleAttrMetadata(1,2,2,0,rc)
mdSccd=BleAttrMetadata(0,0,2,0,rc)

//Create the Characteristic object
IF BleCharNew(3,BleHandleUuid16(0x2A1C),mdVal,mdCccd,mdSccd)==0 THEN
    PRINT "\nSuccess"
ELSE
    PRINT "\nFailed"
ENDIF
```

**Expected Output:**

Success

## BleAttrMetadataEx

**BLEATTRMETADATAEX** (*nReadRights, nWriteRights, nMaxDataLen, nFlags, resCode*)

<b>Returns</b>	INTEGER, a 32-bit opaque data object to be used in subsequent calls when adding Characteristics to a GATT table.
<b>Arguments:</b>	

<b><i>nReadRights</i></b>	byVal nReadRights AS INTEGER	
	This specifies the read rights and shall have one of the following values:	
	0	No access
	1	Open
	2	Encrypted with No Man-In-The-Middle (MITM) protection
	3	Encrypted with Man-In-The-Middle (MITM) protection
	4	Signed with No Man-In-The-Middle (MITM) protection (not available)
<b><i>nWriteRights</i></b>	5	Signed with Man-In-The-Middle (MITM) protection (not available)
	<b>Note:</b> 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:	
	0	No access
	1	Open
	2	Encrypted with No Man-In-The-Middle (MITM) protection
<b><i>nMaxDataLen</i></b>	3	Encrypted with Man-In-The-Middle (MITM) protection
	4	Signed with No Man-In-The-Middle (MITM) protection (not available)
	5	Signed with Man-In-The-Middle (MITM) protection (not available)
	<b>Note:</b> 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.	

	<p><b>byVal nFlags AS INTEGER</b></p> <p>(Note The deprecated function BleAttrMetaData effectively behaves as if this parameter was either 0 or 1 and as if all others bits are 0)</p> <p>This is a bit mask where the bits are defined as follows:-</p> <p><b>Bit 0 :</b></p> <p>Set this to 1 only if you want the attribute to automatically shorten it's length according to the number of bytes written by the client. For example, if the initial length is 2 and the client writes only 1 byte, then if this is 0, then only the first byte gets updated and the rest remain unchanged. If this parameter is set to 1, then when a single byte is written the attribute will shorten it's length to accommodate. If the client tries to write more bytes than the initial maximum length, then the client will get an error response.</p> <p><b>Bit 1:</b></p> <p>Set this to 1 to ensure that the memory for the attribute is allocated from User space (and hence less memory available for smartBASIC) so that a larger gatt table can be created.</p> <p>This bit is ignored for all attributes other than for characteristic value.</p> <p><b>Bit 2:</b></p> <p>Set this to 1 to require authorisation for reads. When an attempt to read is made by the client then one of the events EVAUTHVAL, EVAUTHCCCD, EVAUTHSCCD or EVAUTHDESC is thrown to the app and in the handler for that event, either BleAuthorizeChar() or BleAuthorizeDesc() is called with appropriate parameters to grant or deny access.</p> <p><b>Bit 3</b></p> <p>Set this to 1 to require authorisation for writes. When an attempt to read is made by the client then one of the events EVAUTHVAL, EVAUTHCCCD, EVAUTHSCCD or EVAUTHDESC is thrown to the app and in the handler for that event, either BleAuthorizeChar() or BleAuthorizeDesc() is called with appropriate parameters to grant or deny access.</p>
<p><b>resCode</b></p>	<p><b>byRef resCode AS INTEGER</b></p> <p>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.</p>

BLEATTRMETADATAEX 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)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.																		
<b>Arguments:</b>																			
<b><i>nCharProps</i></b>	<p><b>byVal <i>nCharProps</i> AS INTEGER</b> This variable contains a bit mask to specify the following high level properties for the characteristic that is added to the GATT table:</p> <table border="1"> <thead> <tr> <th>Bit</th><th>Description</th></tr> </thead> <tbody> <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>	Bit	Description	0	Broadcast capable (SCCD descriptor must be present)	1	Can be read by the client	2	Can be written by the client without a response	3	Can be written	4	Can be notifiable (CCCD descriptor must be present)	5	Can be indicatable (CCCD descriptor must be present)	6	Can accept signed writes	7	Reliable writes
Bit	Description																		
0	Broadcast capable (SCCD descriptor must be present)																		
1	Can be read by the client																		
2	Can be written by the client without a response																		
3	Can be written																		
4	Can be notifiable (CCCD descriptor must be present)																		
5	Can be indicatable (CCCD descriptor must be present)																		
6	Can accept signed writes																		
7	Reliable writes																		
<b><i>nUuidHandle</i></b>	<p><b>byVal <i>nUuidHandle</i> AS INTEGER</b> 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().</p>																		
<b><i>mdVal</i></b>	<p><b>byVal <i>mdVal</i> AS INTEGER</b> 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().</p>																		
<b><i>mdCccd</i></b>	<p><b>byVal <i>mdCccd</i> AS INTEGER</b> 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 <i>nCharProps</i> specifies that the characteristic is notifiable or indicatable and this value contains 0, this function aborts with an appropriate result code.</p>																		
<b><i>mdSccd</i></b>	<p><b>byVal <i>mdSccd</i> AS INTEGER</b> 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 <i>nCharProps</i> specifies that the characteristic is broadcastable and this value contains 0, this function aborts with an appropriate resultcode.</p>																		

**Example:**

```
// Example :: BleCharNew.sb (See in BT900CodeSnippets.zip)
DIM rc
DIM charUuid : charUuid = BleHandleUuid16(2)    //Characteristic's UUID
DIM mdVal : mdVal = BleAttrMetadata(1,0,20,0,rc) //Metadata for value attribute
DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc) //Metadata for CCCD attribute of Characteristic

//=====
// Create a new char:
// --- Indicatable, not Broadcastable (so mdCccd is included, but not mdSccd)
```

```
// --- Can be read, not written (shown in mdVal as well)
//=====
IF BleCharNew(0x22,charUuid,mdVal,mdCccd,0)==0 THEN
    PRINT "\nNew Characteristic created"
ELSE
    PRINT "\nFailed"
ENDIF
```

#### Expected Output:

```
New Characteristic created
```

## BleCharDescUserDesc

### FUNCTION

This function adds an optional User Description Descriptor to a Characteristic and can only be called after BleCharNew() starts the process of describing a new characteristic.

The BT 4.0 specification describes the User Description Descriptor as “.. a UTF-8 string of variable size that is a textual description of the characteristic value.” It further stipulates that this attribute is optionally writable and so a metadata argument exists to configure it as such. The metadata automatically updates the Writable Auxilliaries properties flag for the characteristic. This is why that flag bit is NOT specified for the nCharProps argument to the BleCharNew() function.

#### BLECHARDESCUSERDESC(*userDesc\$, mdUser*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>userDesc\$</i></b>	<b>byRef <i>userDesc\$</i> AS STRING</b> The user description string with which to initilise the descriptor. If the length of the string exceeds the maximum length of an attribute then this function aborts with an error result code.
<b><i>mdUser</i></b>	<b>byVal <i>mdUser</i> AS INTEGER</b> This is a mandatory metadata that defines the properties of the User Description Descriptor attribute created in the characteristic and pre-created using the help of BleAttrMetadata(). If the write rights are set to 1 or greater, the attribute is marked as writable and the client is able to provide a user description that overwrites the one provided in this call.

#### Example:

```
//Example :: BleCharDescUserDesc.sb (See in BT900CodeSnippets.zip)
DIM rc, metaSuccess,usrDesc$ : usrDesc$="A description"
DIM charUuid : charUuid = BleHandleUuid16(1)
DIM charMet : charMet = BleAttrMetaData(1,1,20,0,metaSuccess)
DIM mdUsrDsc : mdUsrDsc = BleAttrMetaData(1,1,20,0,metaSuccess)
DIM mdSccd : mdSccd = BleAttrMetadata(1,1,2,0,rc) //CCCD metadata for char

//initialise char, write/read enabled, accept signed writes, indicatable
```



```
rc=BleCharNew(0x4B,charUuid,charMet,0,mdSccd)
rc=BleCharDescUserDesc(usrDesc$,mdUsrDsc)

IF rc==0 THEN
    PRINT "\nChar created and User Description '";usrDesc$;"' added"
ELSE
    PRINT "\nFailed"
ENDIF
```

#### Expected Output:

```
Char created and User Description 'A description' added
```

### 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)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.			
<b>Arguments:</b>				
<b>nFormat</b>	<b>byVal nFormat AS INTEGER</b>			
	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 <a href="http://developer.bluetooth.org/GATT/Pages/FormatTypes.aspx">http://developer.bluetooth.org/GATT/Pages/FormatTypes.aspx</a> 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	duint16	0x19	utf8s
	0x1A	utf16s	0x1B	struct
	0x1C-0xFF	RFU		
<b>nExponent</b>	<b>byVal nExponent AS INTEGER</b> 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: <i>actual value = Characteristic Value * 10 to the power of nExponent.</i> Valid range -128 to 127			
<b>nUnit</b>	<b>byVal nUnit AS INTEGER</b> 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: <a href="http://developer.bluetooth.org/GATT/units/Pages/default.aspx">http://developer.bluetooth.org/GATT/units/Pages/default.aspx</a> Valid range 0 to 65535.			
<b>nNameSpace</b>	<b>byVal nNameSpace AS INTEGER</b> The value identifies the organization, defined in the Assigned Numbers document published by the Bluetooth SIG, found at: <a href="https://developer.bluetooth.org/GATT/Pages/GATTNamespaceDescriptors.aspx">https://developer.bluetooth.org/GATT/Pages/GATTNamespaceDescriptors.aspx</a> Valid range 0 to 255.			
<b>nNSdesc</b>	<b>byVal nNSdesc AS INTEGER</b> This value is a description of the organisation specified by nNameSpace. Valid range 0 to 65535.			

#### Example:

```
//Example :: BleCharDescPrstnFrmt.sb (See in BT900CodeSnippets.zip)

DIM rc, metaSuccess, usrDesc$ : usrDesc$="A description"
DIM charUuid : charUuid = BleHandleUuid16(1)
DIM charMet : charMet = BleAttrMetadata(1,1,20,0,metaSuccess)
DIM mdUsrDsc : mdUsrDsc = BleAttrMetadata(1,1,20,0,metaSuccess)
DIM mdSccd : mdSccd = BleAttrMetadata(1,1,2,0,rc) //CCCD metadata for char

//initialise char, write/read enabled, accept signed writes, indicatable
rc=BleCharNew(0x4B,charUuid,charMet,0,mdSccd)
rc=BleCharDescUserDesc(usrDesc$,mdUsrDsc)

IF rc==0 THEN
    PRINT "\nChar created and User Description ";usrDesc$;" added"
ELSE
    PRINT "\nFailed"
```

```
ENDIF

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

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

IF BleCharDescPrstnFrmt(0x0E, 2, 0x271A, 0x01, 0x0000) == 0 THEN
    PRINT "\nPresentation Format Descriptor added"
ELSE
    PRINT "\nPresentation Format Descriptor not added"
ENDIF
```

#### Expected Output:

```
Char created and User Description 'A description' added
Presentation Format Descriptor added
```

## 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*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nUuid16</i></b>	<p><b>byVal <i>nUuid16</i> AS INTEGER</b></p> <p>This is a value in the range 0x2905 to 0x2999</p> <p><b>Note:</b> This is the actual UUID value, NOT the handle.</p> <p>The highest value at the time of writing is 0x2908, defined for the Report Reference Descriptor. See <a href="http://developer.bluetooth.org/GATT/descriptors/Pages/DescriptorsHomePage.aspx">http://developer.bluetooth.org/GATT/descriptors/Pages/DescriptorsHomePage.aspx</a> for a list of Descriptors defined and adopted by the Bluetooth SIG.</p>

<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This is the data that is saved in the Descriptor's attribute
<b>mdDesc</b>	<b>byVal n AS INTEGER</b> This is mandatory metadata that is used to define the properties of the Descriptor attribute that is created in the Characteristic and was pre-created using the help of the function BleAttrMetadata(). If the write rights are set to 1 or greater, then the attribute is marked as writable and the client is able to modify the attribute value.

**Example:**

```
//Example :: BleCharDescAdd.sb (See in BT900CodeSnippets.zip)

DIM rc, metaSuccess,usrDesc$ : usrDesc$="A description"
DIM charUuid : charUuid = BleHandleUuid16(1)
DIM charMet : charMet = BleAttrMetaData(1,1,20,0,metaSuccess)
DIM mdUsrDsc : mdUsrDsc = charMet
DIM mdSccd : mdSccd = charMet

//initialise char, write/read enabled, accept signed writes, indicatable
rc=BleCharNew(0x4B,charUuid,charMet,0,mdSccd)
rc=BleCharDescUserDesc(usrDesc$,mdUsrDsc)
rc=BleCharDescPrstnFrmt(0x0E,2,0x271A,0x01,0x0000)

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

//++++
//Add the other Descriptor 0x29XX -- first one
//++++
DIM mdChrDsc : mdChrDsc = BleAttrMetadata(1,0,20,0,metaSuccess)
DIM attr$ : attr$="some_value1"
rc=BleCharDescAdd(0x2905,attr$,mdChrDsc)

//++++
//Add the other Descriptor 0x29XX -- second one
//++++
attr$="some_value2"
rc=rc+BleCharDescAdd(0x2906,attr$,mdChrDsc)

//++++
//Add the other Descriptor 0x29XX -- last one
```

```
//++++
attr$="some_value3"
rc=rc+BleCharDescAdd(0x2907,attr$,mdChrDsc)

IF rc==0 THEN
    PRINT "\nOther descriptors added successfully"
ELSE
    PRINT "\nFailed"
ENDIF
```

#### Expected Output:

```
Other descriptors added successfully
```

## BleCharCommit

### FUNCTION

This function commits a characteristic which was prepared by calling BleCharNew() and optionally BleCharDescUserDesc(), BleCharDescPrstnFrmt() or BleCharDescAdd().

It is an instruction to the GATT table manager that all relevant attributes that make up the characteristic should appear in the GATT table in a single atomic transaction. If it successfully created, a single composite characteristic handle is returned which should not be confused with GATT table attribute handles. If the Characteristic was not accepted then this function returns a non-zero result code which conveys the reason and the handle argument that is returned has a special invalid handle of 0.

The characteristic handle that is returned references an internal opaque object that is a linked list of all the attribute handles in the characteristic which by definition implies that there is a minimum of 1 (for the characteristic value attribute) and more as appropriate. For example, if the characteristic's property specified is notifiable then a single CCCD attribute also exists.

**Note:** In the GATT table, when a characteristic is registered, there are actually a minimum of two attribute handles, one for the Characteristic Declaration and the other for the Value. However there is no need for the *smart*BASIC apps developer to access it, so it is not exposed. Access is not required because the characteristic was created by the application developer and so shall already know its content – which never changes once created.

### BLECHARCOMMIT (*hService,attr\$,charHandle*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>hService</i></b>	<b>byVal hService AS INTEGER</b> This is the handle of the service to which the characteristic belongs, which in turn was created using the function BleSvcCommit().

<b>attr\$</b>	<p><b>byRef attr\$ AS STRING</b> 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.</p>
<b>charHandle</b>	<p><b>byRef charHandle AS INTEGER</b> 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 must be placed in a global <i>smart</i>BASIC variable. When a significant event occurs as a result of action by a remote client, an event message is sent to the application which can be serviced using a handler. That message contains a handle field corresponding to this composite characteristic handle. Standard procedure is to select on that value to determine for which characteristic the message is intended. See event messages: EVCHARHVC, EVCHARVAL, EVCHARCCCD, EVCHARSCCD, EVCHARDESC.</p>

**Example:**

```
// Example :: BleCharCommit.sb (See in BT900CodeSnippets.zip)

#DEFINE BLE_SERVICE_SECONDARY          0
#DEFINE BLE_SERVICE_PRIMARY            1

DIM rc
DIM attr$,usrDesc$ : usrDesc$="A description"
DIM hHtsSvc //composite handle for hts primary service
DIM mdCharVal : mdCharVal = BleAttrMetaData(1,1,20,0,rc)
DIM mdCccd : mdCccd = BleAttrMetadadata(1,1,2,0,rc)
DIM mdUsrDsc : mdUsrDsc = BleAttrMetaData(1,1,20,0,rc)
DIM hHtsMeas //composite handle for htsMeas characteristic

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

//-----
//Create the Measurement Characteristic object, add user description descriptor
//-----
rc=BleCharNew(0x2A,BleHandleUuid16(0x2A1C),mdCharVal,mdCccd,0)
rc=BleCharDescUserDesc(usrDesc$,mdUsrDsc)

//-----
//Commit the characteristics with some initial data
//-----
attr$="hello\00worl\64"
```

```
IF BleCharCommit(hHtsSvc,attr$,hHtsMeas)==0 THEN
    PRINT "\nCharacteristic Committed"
ELSE
    PRINT "\nFailed"
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 Committed
```

## BleCharValueRead

### FUNCTION

This function reads the current content of a characteristic identified by a composite handle that was previously returned by the function `BleCharCommit()`.

In most cases a read will be performed when a GATT client writes to a characteristic value attribute. The write event is presented asynchronously to the *smart*BASIC application in the form of EVCHARVAL event. This function will most often be accessed from the handler that services that event.

#### **BLECHARVALUEREAD** (*charHandle*,*attr\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>charHandle</i></b>	<b>byVal <i>charHandle</i> AS INTEGER</b> This is the handle to the characteristic whose value must be read which was returned when <code>BleCharCommit()</code> was called.
<b><i>attr\$</i></b>	<b>byRef <i>attr\$</i> AS STRING</b> This string variable contains the new value from the characteristic.

#### Example:

```
//Example :: BleCharValueRead.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc, conHndl

//=====
// Initialise and instantiate service, characteristic,
//=====
```

```
FUNCTION OnStartup()  
    DIM rc, hSvc, scRpt$, adRpt$, addr$, attr$ : attr$="Hi"  
  
    //commit service  
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)  
    //initialise char, write/read enabled, accept signed writes  
    rc=BleCharNew(0x0A,BleHandleUuid16(1),BleAttrMetaData(1,1,20,0,rc),0,0)  
    //commit char initialised above, with initial value "hi" to service 'hSvc'  
    rc=BleCharCommit(hSvc,attr$,hMyChar)  
    //commit changes to service  
    rc=BleServiceCommit(hSvc)  
    //initialise scan report  
    rc=BleScanRptInit(scRpt$)  
    //Add 1 service handle to scan report  
    rc=BleAdvRptAddUuid16(scRpt$,0x18EE,-1,-1,-1,-1,-1)  
    //commit reports to GATT table - adRpt$ is empty  
    rc=BleAdvRptsCommit(adRpt$,scRpt$)  
    rc=BleAdvertStart(0,addr$,150,0,0)  
ENDFUNC rc  
  
//=====   
// New char value handler  
//=====   
FUNCTION HndlrChar(BYVAL chrHndl, BYVAL offset, BYVAL len)  
    dim s$  
    IF chrHndl == hMyChar THEN  
        PRINT "\n";len;" byte(s) have been written to char value attribute from offset ";offset  
  
        rc=BleCharValueRead(hMyChar,s$)  
        PRINT "\nNew Char Value: ";s$  
    ENDIF  
    rc=BleAdvertStop()  
    rc=BleDisconnect(conHndl)  
ENDFUNC 0  
  
//=====   
// Get the connection handle  
//=====   
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtn)
```



```

conHndl=nCtn
ENDFUNC 1

IF OnStartup()==0 THEN
    DIM at$ : rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic value attribute: ";at$;"\nConnect to BT900 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 BT900 and send a new value

New characteristic value: Laird
Exiting...

```

## 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\$*)

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

#### Example:

```
//Example :: BleCharValueWrite.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc

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

    //commit service
    rc = BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    //initialise char, write/read enabled, accept signed writes
    rc=BleCharNew(0x4A,BleHandleUuid16(1),BleAttrMetaData(1,1,20,0,rc),0,0)
    //commit char initialised above, with initial value "hi" to service 'hSvc'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc = BleServiceCommit(hSvc)
ENDFUNC rc

//=====
// Uart Rx handler - write input to characteristic
//=====
FUNCTION HndlrUartRx()
    TimerStart(0,10,0)
ENDFUNC 1

//=====
// Timer0 timeout handler
//=====
FUNCTION HndlrTmr0()
    DIM t$ : rc=UartRead(t$)
    rc = BleCharValueWrite(hMyChar,t$)
    IF rc==0 THEN
        PRINT "\nNew characteristic value: ";t$
    ELSE
        PRINT "\nFailed to write new characteristic value ";integer.h'rc;"\n"
    ENDIF
ENDFUNC 0
```

```
IF OnStartup()==0 THEN
    DIM at$ : rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic value attribute: ";at$;"\nType a new value\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

ONEVENT EVUARTRX CALL HndlrUartRx
ONEVENT EVTMR0 CALL HndlrTmr0

WAITEVENT

PRINT "\nExiting..."
```

#### Expected Output:

```
Characteristic value attribute: Hi
Send a new value
Laird

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

## 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\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>charHandle</b>	<b>byVal charHandle AS INTEGER</b> This is the handle to the characteristic whose value must be updated which is returned when <code>BleCharCommit()</code> is called.
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> String variable containing new value to write to the characteristic and then send as a notification to the client. If there is no connection, this function fails with an appropriate result code.

#### Example:

```
//Example :: BleCharValueNotify.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl
```

```
//=====
// Initialise and instantiate service, characteristic, start adverts
//=====
FUNCTION OnStartup()
    DIM rc, hSvc, at$, attr$, adRpt$, addr$, scRpt$
    attr$="Hi"
    DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc) //CCCD metadata for char

    //Commit svc with handle 'hSvcUuid'
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    //initialise char, write/read enabled, accept signed writes, notifiable
    rc=BleCharNew(0x12,BleHandleUuid16(1),BleAttrMetaData(1,0,20,0,rc),mdCccd,0)
    //commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc=BleServiceCommit(hSvc)
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,0x18EE,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,50,0,0)
ENDFUNC rc

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

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

```
ENDIF
ENDFUNC 1

//=====
// CCCD descriptor written handler
//=====
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$
    IF charHandle==hMyChar THEN
        PRINT "\nCCCD Val: ";nVal
        IF nVal THEN
            PRINT " : Notifications have been enabled by client"
            value$="hello"
            IF BleCharValueNotify(hMyChar,value$)!=0 THEN
                PRINT "\nFailed to notify new value :";INTEGER.H'rc
            ELSE
                PRINT "\nSuccessful notification of new value"
                EXITFUNC 0
            ENDIF
        ELSE
            PRINT " : Notifications have been disabled by client"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT EVBLEMSG CALL HndlrBleMsg
ONEVENT EVCHARCCCD CALL HndlrCharCccd

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic Value: ";at$
    PRINT "\nYou can connect and write to the CCCD characteristic."
    PRINT "\nThe BT900 will then notify your device of a new characteristic value\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

CloseConnections()
```

```
PRINT "\nExiting..."
```

#### Expected Output:

```
Characteristic Value: Hi
You can connect and write to the CCCD characteristic.
The BT900 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...
```

## BleCharValueIndicate

### FUNCTION

If there is BLE connection, this function is used to write new data into the VALUE attribute of a characteristic so that it can be sent as an indication to the GATT client. The characteristic is identified by a composite handle returned by the function `BleCharCommit()`.

An indication results in an acknowledgement from the client and that is presented to the *smart*BASIC application as the EVCHARHVC event.

#### **BLECHARVALUEINDICATE** (*charHandle*,*attr\$*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>charHandle</b>	<b>byVal charHandle AS INTEGER</b> This is the handle to the characteristic whose value must be updated which is returned when <code>BleCharCommit()</code> was called.
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> String variable containing new value to write to the characteristic and then to send as a notification to the client. If there is no connection, this function fails with an appropriate result code.

#### Example:

```
//Example :: BleCharValueIndicate.sb (See in BT900CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

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

    DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc) //CCCD metadata for char
```

```
//Commit svc with handle 'hSvcUuid'
rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
//initialise char, write/read enabled, accept signed writes, notifiable
rc=BleCharNew(0x22,BleHandleUuid16(1),BleAttrMetaData(1,0,20,0,rc),mdCccd,0)
//commit char initialised above, with initial value "hi" to service 'hMyChar'
rc=BleCharCommit(hSvc,attr$,hMyChar)
//commit changes to service
rc=BleServiceCommit(hSvc)
rc=BleScanRptInit(scRpt$)
//Add 1 service handle to scan report
rc=BleAdvRptAddUuid16(scRpt$,0x18EE,-1,-1,-1,-1,-1)
//commit reports to GATT table - adRpt$ is empty
rc=BleAdvRptsCommit(adRpt$,scRpt$)
rc=BleAdvertStart(0,addr$,50,0,0)
ENDFUNC rc

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

//=====
// CCCD descriptor written handler
//=====
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal)
    DIM value$
    IF charHandle==hMyChar THEN
        PRINT "\nCCCD Val: ";nVal
        IF nVal THEN
            PRINT " : Indications have been enabled by client"
```

```
value$="hello"
rc=BleCharValueIndicate(hMyChar,value$)
IF rc!=0 THEN
    PRINT "\nFailed to indicate new value :";INTEGER.H'rc
ELSE
    PRINT "\nSuccessful indication of new value"
    EXITFUNC 1
ENDIF
ELSE
    PRINT " : Indications have been disabled by client"
ENDIF
ELSE
    PRINT "\nThis is for some other characteristic"
ENDIF
ENDFUNC 1

//=====
// Indication Acknowledgement Handler
//=====
FUNCTION HndlrChrHvc(BYVAL charHandle)
    IF charHandle == hMyChar THEN
        PRINT "\n\nGot confirmation of recent indication"
    ELSE
        PRINT "\n\nGot confirmation of some other indication: ";charHandle
    ENDIF
ENDFUNC 0

ONEVENT EVBLEMSG CALL HndlrBleMsg
ONEVENT EVCHARCCCD CALL HndlrCharCccd
ONEVENT EVCHARHVC CALL HndlrChrHvc

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic Value: ";at$
    PRINT "\nYou can connect and write to the CCCD characteristic."
    PRINT "\nThe BT900 will then indicate a new characteristic value\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF
```



```
WAITEVENT
```

```
rc=BleDisconnect(conHndl)
rc=BleAdvertStop()
PRINT "\nExiting..."
```

### Expected Output:

```
Characteristic Value: Hi
You can connect and write to the CCCD characteristic.
The BT900 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...
```

## 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\$*)

<b>Returns</b>	INTEGER, a result code. The typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>charHandle</b>	<b>byVal charHandle AS INTEGER</b> 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.
<b>nDescHandle</b>	<b>byVal nDescHandle AS INTEGER</b> 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.
<b>nOffset</b>	<b>byVal nOffset AS INTEGER</b> This is the offset into the descriptor attribute from which the data should be read and copied into attr\$.

<b><i>nLength</i></b>	<b>byVal <i>nLength</i> AS INTEGER</b> This is the number of bytes to read from the descriptor attribute from offset <i>nOffset</i> and copied into <i>attr\$</i> .
<b><i>nDescUuidHandle</i></b>	<b>byRef <i>nDescUuidHandle</i> AS INTEGER</b> On exit, this is updated with the uuid handle of the descriptor that got updated.
<b><i>attr\$</i></b>	<b>byRef <i>attr\$</i> AS STRING</b> On exit, this string variable contains the new value from the characteristic descriptor.

**Example:**

```
//Example :: BleCharDescRead.sb (See in BT900CodeSnippets.zip)
DIM rc, conHndl, hMyChar

//-----
//Create some PRIMARY service attribute which has a uuid of 0x18FF
//-----

SUB OnStartup()
    DIM hSvc, attr$, scRpt$, adRpt$, addr$
    rc=BleSvcCommit(1, BleHandleUuid16(0x18FF), hSvc)
    // Add one or more characteristics
    rc=BleCharNew(0x0a, BleHandleUuid16(0x2AFF), BleAttrMetadata(1, 1, 20, 1, rc), 0, 0)

    //Add a user description
    DIM s$ : s$="You can change this"
    rc=BleCharDescAdd(0x2999, s$, BleAttrMetadata(1, 1, 20, 1, rc))

    //commit characteristic
    attr$="\00" //no initial alert
    rc = BleCharCommit(hSvc, attr$, hMyChar)
    rc=BleScanRptInit(scRpt$)
    //Add 1 char handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$, 0x2AFF, -1, -1, -1, -1, -1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$, scRpt$)
    rc=BleAdvertStart(0, addr$, 200, 0, 0)
ENDSUB

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
```

```
rc=BleAdvertStop()
ENDSUB

//=====
// Ble event handler - Just to get the connection handle
//=====
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
ENDFUNC 1

//=====
// Handler to service writes to descriptors by a GATT client
//=====
FUNCTION HandlerCharDesc(BYVAL hChar AS INTEGER, BYVAL hDesc AS INTEGER)
    DIM instnc,nUuid,a$, offset,duid

    IF hChar == hMyChar THEN
        rc = BleCharDescRead(hChar,hDesc,0,20,duid,a$)
        IF rc==0 THEN
            PRINT "\nRead 20 bytes from index ";offset;" in new char value."
            PRINT "\n ::New Descriptor Data: ";StrHexize$(a$);
            PRINT "\n ::Length=";StrLen(a$)
            PRINT "\n ::Descriptor UUID ";integer.h' duid
            EXITFUNC 0
        ELSE
            PRINT "\nCould not access the uuid"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

//install a handler for writes to characteristic values
ONEVENT EVCHARDESC CALL HandlerCharDesc
ONEVENT EVBLEMSG CALL HndlrBleMsg

OnStartup()
PRINT "\nWrite to the User Descriptor with UUID 0x2999"
```

```
//wait for events and messages
WAITEVENT

CloseConnections()
PRINT "\nExiting..."
```

#### Expected Output:

```
Write to the User Descriptor with UUID 0x2999
Read 20 bytes from index 0 in new char value.
::New Descriptor Data: 4C61697264
::Length=5
::Descriptor UUID FE012999
Exiting...
```

## GATT Client Functions

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

These GATT client functions allow the developer to discover services, characteristics and descriptors, read and write to characteristics and descriptors, and handle either notifications or indications.

To interact with a remote GATT server, it is important to have a good understanding of how it is constructed. It is best to see it as a table consisting of many rows and three visible columns (handle, type, value) and at least one more invisible column whose content affects access to the data column.

16 bit Handle	Type (16 or 128 bit)	Value (1 to 512 bytes)	Permissions
---------------	----------------------	------------------------	-------------

These rows are grouped into collections called services and characteristics. The grouping is achieved by creating a row with Type = 0x2800 or 0x2801 for services (primary and secondary respectively) and 0x2803 for characteristics.

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

When scanning, if a row is encountered with the value 0x2800 or 0x2801 in the Type column, then it is understood as the start of a primary or secondary service which in turn contains at least one characteristic or one 'included service' which have Type=0x2803 and 0x2802 respectively.

When a row with Type = 0x2803 (a characteristic) is encountered, then the next row contains the value for that characteristic; afterwards, there may be zero or more descriptors.

This means each characteristic consists of at least two rows in the table; and if descriptors exist for that characteristic, then a single row per descriptor.

Handle	Type	Value	Comments
0x0001	0x2800	UUID of the Service	Primary Service 1 Start
0x0002	0x2803	Properties, Value Handle, Value UUID1	Characteristic 1 Start
0x0003	Value UUID1	Value : 1 to 512 bytes	Actual data
0x0004	0x2803	Properties, Value Handle, Value UUID2	Characteristic 2 Start
0x0005	Value UUID2	Value : 1 to 512 bytes	Actual data
0x0006	0x2902	Value	Descriptor 1( CCCD)
0x0007	0x2903	Value	Descriptor 2 (SCCD)
0x0008	0x2800	UUID of the Service	Primary Service 2 Start
0x0009	0x2803	Properties, Value Handle, Value UUID3	Characteristic 1 Start
0x000A	Value UUID3	Value : 1 to 512 bytes	Actual data
0x000B	0x2800	UUID of the Service	Primary Service 3 Start
0x000C	0x2803	Properties, Value Handle, Value UUID3	Characteristic 3 Start
0x000D	Value UUID3	Value : 1 to 512 bytes	Actual data
0x000E	0x2902	Value	Descriptor 1( CCCD)
0x000F	0x2903	Value	Descriptor 2 (SCCD)
0x0010	0x2904	Value (presentation format data)	Descriptor 3
0x0011	0x2906	Value (valid range)	Descriptor 4 (Range)

A colour highlighted example of a GATT server table is shown above. There are three **services** (at handles 0x0001, 0x0008 and 0x000B) because there are three rows where the Type = 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 = 0x0004 has the mandatory value row and then two **descriptors**.

The Bluetooth specification allows for multiple instances of the same service or characteristics or descriptors and they are differentiated by the unique handle. This ensures no ambiguity.

Each GATT server table allocates the handle numbers, the only stipulation being that they be in ascending order (gaps are allowed). This is important to understand because two devices containing the same services and characteristic and in EXACTLY the same order may NOT allocate the same handle values, especially if one device increments handles by 1 and another with some other arbitrary random value. The specification does stipulate that once the handle values are allocated, they are fixed for all subsequent connections unless the device

exposes a GATT service which allows for indications to the client that the handle order has changed and thus force it to flush its cache and rescan the GATT table.

When a connection is first established, there is no prior knowledge as to which services exist or their handles. Therefore, the GATT protocol which is used to interact with GATT servers, provides procedures that allow for the GATT table to be scanned so that the client can ascertain which services are offered. This section describes *smart*BASIC functions which encapsulate and manage those procedures to enable a *smart*BASIC application to map the table.

These helper functions have been written to help gather the handles of all the rows which contain the value type for appropriate characteristics as those are the ones that will be read or written to. The *smart*BASIC internal engine also maintains data objects so that it is possible to interact with descriptors associated with the characteristic.

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

This approach ensures that the least amount of RAM resource is required to implement a GATT client and, given that these procedures operate at speeds many orders of magnitude slower compared to the speed of the CPU and energy consumption is to be kept as low as possible, the response to a command is delivered asynchronously as an event for which a handler must be specified in the user *smart*BASIC application.

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

## Events and Messages

The nature of GATT client operation consists of multiple queries and acting on the responses. Because the connection intervals are slower than the CPU speed, responses can arrive many tens of milliseconds after the 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

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

- Connection Handle

#### Example:

```
//Example :: EVGATTCTOUT.sb (See in BT900CodeSnippets.zip)
//

DIM rc, conHndl

//=====
```

```
// Initialise and instantiate service, characteristic, start adverts
//=====
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGATTcOpen(0,0) : ENDIF
ENDFUNC rc

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

'//=====
'//=====
FUNCTION HandlerGATTcTout(cHndl) AS INTEGER
    PRINT "\nEVGATTCTOUT connHandle=";cHndl
ENDFUNC 1

//=====
// Main() equivalent
//=====
ONEVENT EVBLEMSG      CALL HndlrBleMsg
OnEvent EVGATTCTOUT   call HandlerGATTcTout

rc = OnStartup()

WAITEVENT
```

### Expected Output:

```

. . .
. . .
EVGATTCTOUT connHandle=123
. . .
. . .

```

### *EVDISCPRIMSV*

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.

### *EVDISCCHAR*

This event message is thrown if either `BleDiscCharFirst()` or `BleDiscCharNext()` returns a success. The message contains the following INTEGER parameters:

- Connection Handle
- Characteristic UUID Handle
- Characteristic properties
- Handle for the value attribute of the characteristic
- Included Service UUID Handle

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

‘**Characteristic Uuid Handle**’ contains the UUID of the characteristic and supplied as a handle.

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

<b>Bit 0</b>	Set if BROADCAST is enabled
<b>Bit 1</b>	Set if READ is enabled
<b>Bit 2</b>	Set if WRITE_WITHOUT_RESPONSE is enabled
<b>Bit 3</b>	Set if WRITE is enabled
<b>Bit 4</b>	Set if NOTIFY is enabled
<b>Bit 5</b>	Set if INDICATE is enabled
<b>Bit 6</b>	Set if AUTHENTICATED_SIGNED_WRITE is enabled
<b>Bit 7</b>	Set if RELIABLE_WRITE is enabled

‘**Handle for the Value Attribute of the Characteristic**’ is the handle for the value attribute and is the value to store to keep track of important characteristics in a GATT server for later read/write operations.

‘**Included Service Uuid Handle**’ is for future use and is always 0.



### *EVDISCDESC*

This event message is thrown if either `BleDissDescFirst()` or `BleDiscDescNext()` returns a success. The message contains the following INTEGER parameters:

- Connection Handle
- Descriptor Uuid Handle
- Handle for the Descriptor in the remote GATT Table

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

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

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

### *EVFINDCHAR*

This event message is thrown if `BleGATTcFindChar()` returns a success. The message contains the following INTEGER parameters:

- Connection Handle
- Characteristic Properties
- Handle for the Value Attribute of the Characteristic
- Included Service Uuid Handle

If the specified instance of the service/characteristic is not present in the remote GATT server table, then all parameters contain zero apart from the Connection Handle.

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

Bit	Description
0	Set if BROADCAST is enabled
1	Set if READ is enabled
2	Set if WRITE_WITHOUT_RESPONSE is enabled
3	Set if WRITE is enabled
4	Set if NOTIFY is enabled
5	Set if INDICATE is enabled
6	Set if AUTHENTICATED_SIGNED_WRITE is enabled
7	Set if RELIABLE_WRITE is enabled
15	Set if the characteristic has extended properties

**‘Handle for the Value Attribute of the Characteristic’** is the handle for the value attribute and is the value to store to keep track of important characteristics in a GATT server for later read/write operations.

**‘Included Service Uuid Handle’** is for future use and is always 0.

### *EVFINDDESC*

This event message is thrown if `BleGATTcFindDesc()` returned a success. The message contains the following INTEGER parameters:

Connection Handle  
Handle of the Descriptor

If the specified instance of the service/characteristic/descriptor is not present in the remote GATT server table, then all parameters contain zero apart from the Connection Handle.

**'Handle of the Descriptor'** is the handle for the descriptor and is the value to store to keep track of important descriptors in a GATT server for later read/write operations – for example, CCCDs to enable notifications and/or indications.

### EVATTRREAD

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

Connection Handle  
Handle of the Attribute  
GATT status of the read operation

**'GATT status of the read operation'** is one of the following values, where 0 implies the read was successfully expedited and the data can be obtained by calling BlePubGattClientReadData().

```

0x0000 Success
0x0001 Unknown or not applicable status
0x0100 ATT Error: Invalid Error Code
0x0101 ATT Error: Invalid Attribute Handle
0x0102 ATT Error: Read not permitted
0x0103 ATT Error: Write not permitted
0x0104 ATT Error: Used in ATT as Invalid PDU
0x0105 ATT Error: Authenticated link required
0x0106 ATT Error: Used in ATT as Request Not Supported
0x0107 ATT Error: Offset specified was past the end of the attribute
0x0108 ATT Error: Used in ATT as Insufficient Authorisation
0x0109 ATT Error: Used in ATT as Prepare Queue Full
0x010A ATT Error: Used in ATT as Attribute not found
0x010B ATT Error: Attribute cannot be read or written using read/write blob requests
0x010C ATT Error: Encryption key size used is insufficient
0x010D ATT Error: Invalid value size
0x010E ATT Error: Very unlikely error
0x010F ATT Error: Encrypted link required
0x0110 ATT Error: Attribute type is not a supported grouping attribute
0x0111 ATT Error: Encrypted link required
0x0112 ATT Error: Reserved for Future Use range #1 begin
0x017F ATT Error: Reserved for Future Use range #1 end
0x0180 ATT Error: Application range begin
0x019F ATT Error: Application range end
0x01A0 ATT Error: Reserved for Future Use range #2 begin
0x01DF ATT Error: Reserved for Future Use range #2 end
0x01E0 ATT Error: Reserved for Future Use range #3 begin
0x01FC ATT Error: Reserved for Future Use range #3 end
0x01FD ATT Common Profile and Service Error: Client Characteristic Configuration Descriptor
(CCCD)improperly configured
0x01FE ATT Common Profile and Service Error:Procedure Already in Progress
0x01FF ATT Common Profile and Service Error: Out Of Range

```

### EVATTRWRITE

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

Connection Handle

Handle of the Attribute

GATT status of the write operation

'GATT status of the write operation' is one of the following values, where 0 implies the write was successfully expedited.

```

0x0000 Success
0x0001 Unknown or not applicable status
0x0100 ATT Error: Invalid Error Code
0x0101 ATT Error: Invalid Attribute Handle
0x0102 ATT Error: Read not permitted
0x0103 ATT Error: Write not permitted
0x0104 ATT Error: Used in ATT as Invalid PDU
0x0105 ATT Error: Authenticated link required
0x0106 ATT Error: Used in ATT as Request Not Supported
0x0107 ATT Error: Offset specified was past the end of the attribute
0x0108 ATT Error: Used in ATT as Insufficient Authorisation
0x0109 ATT Error: Used in ATT as Prepare Queue Full
0x010A ATT Error: Used in ATT as Attribute not found
0x010B ATT Error: Attribute cannot be read or written
        using read/write blob requests
0x010C ATT Error: Encryption key size used is insufficient
0x010D ATT Error: Invalid value size
0x010E ATT Error: Very unlikely error
0x010F ATT Error: Encrypted link required
0x0110 ATT Error: Attribute type is not a supported grouping attribute
0x0111 ATT Error: Encrypted link required
0x0112 ATT Error: Reserved for Future Use range #1 begin
0x017F ATT Error: Reserved for Future Use range #1 end
0x0180 ATT Error: Application range begin
0x019F ATT Error: Application range end
0x01A0 ATT Error: Reserved for Future Use range #2 begin
0x01DF ATT Error: Reserved for Future Use range #2 end
0x01E0 ATT Error: Reserved for Future Use range #3 begin
0x01FC ATT Error: Reserved for Future Use range #3 end
0x01FD ATT Common Profile and Service Error:
        Client Characteristic Configuration Descriptor (CCCD)
        improperly configured
0x01FE ATT Common Profile and Service Error:
        Procedure Already in Progress
0x01FF ATT Common Profile and Service Error:
        Out Of Range

```

### EVNOTIFYBUF

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

### EVATTRNOTIFY

This event is thrown when a notification or an indication arrives from a GATT server. The event contains no parameters. Please note that if one notification/indication arrives or many, like in the case of UART events, the same event mask bit is asserted. The *smart*BASIC application is informed that it must go and service the ring buffer using the function `BleGattcNotifyRead`.

### 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 BT900 use cases do not use it, the sacrifice of 300 bytes (nearly 15% of the available memory) 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 GATT client operations are not interrupted.

It is recommended that this function NOT be called when in a connection.

#### **BLEGATTOPEN (*nNotifyBufLen*, *nFlags*)**

<b>Returns</b>	INTEGER, a result code. The typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nNotifyBufLen</i></b>	<b>byVal <i>nNotifyBufLen</i> AS INTEGER</b> This is the size of the ring buffer used for incoming notifiable and indicatable characteristic data. Set to 0 to use the default size.
<b><i>nFlags</i></b>	<b>byVal <i>nFlags</i> AS INTEGER</b> <b>Bit 0</b> – 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. <b>Bit 1..31</b> – Reserved for future use and must be set to 0s.

#### **Example:**

```
//Example :: BleGattcOpen.sb (See in BT900CodeSnippets.zip)
DIM rc
//open the GATT client with default notify/indicate ring buffer size
rc = BleGattcOpen(0,0)
IF rc == 0 THEN
    PRINT "\nGATT Client is now open"
ENDIF
//open the client with default notify/indicate ring buffer size - again
rc = BleGattcOpen(128,1)
IF rc == 0 THEN
    PRINT "\nGATT Client is still open, because already open"
ENDIF
```

**Expected Output:**

```
GATT Client is now open
GATT Client is still open, because already open
```

## 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 ()**

<b>Returns</b>	
<b>Arguments</b>	None
<b>Interactive Command</b>	No

**Example:**

```
//Example :: BleGattcClose.sb (See in BT900CodeSnippets.zip)

DIM rc
//open the GATT client with default notify/indicate ring buffer size
rc = BleGattcOpen(0,0)
IF rc == 0 THEN
    PRINT "\nGATT Client is now open"
ENDIF
BleGattcClose()
PRINT "\nGATT Client is now closed"
BleGattcClose()
PRINT "\nGATT Client is closed - was safe to call when already closed"
```

**Expected Output:**

```
GATT Client is now open
GATT Client is now closed
GATT Client is closed - was safe to call when already closed
```

## BleDiscServiceFirst / BleDiscServiceNext

### FUNCTIONS

This pair of functions is used to scan the remote GATT server for all primary services with the help of the EVDISCPRIMSVC message event. When called, a handler for the event message must be registered as the discovered primary service information is passed back in that message.

A generic or UUID-based scan can be initiated. The former scans for all primary services and the latter scans for a primary service with a particular UUID, the handle of which must be supplied and is generated by using either BleHandleUuid16() or BleHandleUuid128().

While the scan is in progress and waiting for the next piece of data from a GATT server, the module enters low power state as the WAITEVENT statement is used as normal to wait for events and messages.

Depending on the size of the remote GATT server table and the connection interval, the scan of all primary may take many hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

### **BLEDISCSERVICEFIRST** (*connHandle,startAttrHandle,uuidHandle*)

A typical pseudo code for discovering primary services involves first calling BleDiscServiceFirst(), then waiting for the EVDISCPRIMSVC event message and depending on the information returned in that message calling BleDiscServiceNext(), which in turn will result in another EVDISCPRIMSVC event message and typically is as follows:

```
Register a handler for the EVDISCPRIMSVC event message

On EVDISCPRIMSVC event message
  If Start/End Handle == 0 then scan is complete
  Else Process information then
    call BleDiscServiceNext()
    if BleDiscServiceNext() not OK then scan complete

Call BleDiscServiceFirst()
If BleDiscServiceFirst() ok then Wait for EVDISCPRIMSVC
```

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>connHandle</b>	<b>byVal nConnHandle AS INTEGER</b> 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.
<b>startAttrHandle</b>	<b>byVal startAttrHandle AS INTEGER</b> 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
<b>uuidHandle</b>	<b>byVal uuidHandle AS INTEGER</b> Set this to 0 if you want to scan for any service, otherwise this value will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

### **BLEDISCSERVICENEXT** (*connHandle*)

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

<b>Returns</b>	<p><b>INTEGER, a result code.</b></p> <p>The typical value is 0x0000, indicating a successful operation and it means an EVDISCPRIMSVC event message is thrown by the <i>smart</i>BASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message is not thrown.</p>
<b>Arguments:</b>	
<b>connHandle</b>	<p><b>byVal nConnHandle AS INTEGER</b></p> <p>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</p>

**Example:**

```
//Example :: BleDiscServiceFirst.Next.sb (See in BT900CodeSnippets.zip)
//
//Remote server has 5 prim services with 16 bit uuid and 3 with 128 bit uuids
// 3 of the 16 bit uuid are the same value 0xDEAD and
// 2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGATTcTblDiscPrimSvc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
```

```
rc=BleAdvertStop()  
ENDSUB  
  
//=====  
// Ble event handler  
//=====  
FUNCTION HndlrBleMsg (BYVAL nMsgId, BYVAL nCtx)  
    DIM uu$  
    conHndl=nCtx  
    IF nMsgID==1 THEN  
        PRINT "\n\n- Disconnected"  
        EXITFUNC 0  
    ELSEIF nMsgID==0 THEN  
  
        PRINT "\n- Connected, so scan remote GATT Table for ALL services"  
        rc = BleDiscServiceFirst (conHndl,0,0)  
        IF rc==0 THEN  
            //HandlerPrimSvc() will exit with 0 when operation is complete  
            WAITEVENT  
  
            PRINT "\nScan for service with uuid = 0xDEAD"  
            uHndl = BleHandleUuid16 (0xDEAD)  
            rc = BleDiscServiceFirst (conHndl,0,uHndl)  
            IF rc==0 THEN  
                //HandlerPrimSvc() will exit with 0 when operation is complete  
                WAITEVENT  
  
                uu$ = "112233445566778899AABBCCDDEEFF00"  
                PRINT "\nScan for service with custom uuid ";uu$  
                uu$ = StrDehexize$(uu$)  
                uHndl = BleHandleUuid128 (uu$)  
                rc = BleDiscServiceFirst (conHndl,0,uHndl)  
                IF rc==0 THEN  
                    //HandlerPrimSvc() will exit with 0 when operation is complete  
                    WAITEVENT  
                ENDIF  
            ENDIF  
        ENDIF  
        CloseConnections()  
    ENDIF  
ENDIF
```



```
ENDIF
ENDFUNC 1

//=====
// EVDISCPRIMSVc event handler
//=====
FUNCTION HandlerPrimSvc(cHndl,svcUuid,sHndl,eHndl) AS INTEGER
    PRINT "\nEVDISCPRIMSVc : "
    PRINT " cHndl=";cHndl
    PRINT " svcUuid=";integer.h' svcUuid
    PRINT " sHndl=";sHndl
    PRINT " eHndl=";eHndl
    IF sHndl == 0 THEN
        PRINT "\nScan complete"

        EXITFUNC 0
    ELSE
        rc = BleDiscServiceNext(cHndl)
        IF rc != 0 THEN
            PRINT "\nScan abort"

            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1

//=====
// Main() equivalent
//=====
ONEVENT EVBLEMSG      CALL HndlrBleMsg
OnEvent EVDISCPRIMSVc call HandlerPrimSvc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
```

```
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

### Expected Output:

```
Advertising, and GATT Client is open

- Connected, so scan remote GATT Table for ALL services
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01FE01 sHndl=1 eHndl=3
EVDISCPRIMSVC : cHndl=2804 svcUuid=FC033344 sHndl=4 eHndl=6
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=7 eHndl=9
EVDISCPRIMSVC : cHndl=2804 svcUuid=FB04BEEF sHndl=10 eHndl=12
EVDISCPRIMSVC : cHndl=2804 svcUuid=FC033344 sHndl=13 eHndl=15
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=16 eHndl=18
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01FE03 sHndl=19 eHndl=21
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=22 eHndl=24
EVDISCPRIMSVC : cHndl=2804 svcUuid=00000000 sHndl=0 eHndl=0
Scan complete
Scan for service with uuid = 0xDEAD
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=7 eHndl=9
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=16 eHndl=18
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=22 eHndl=65535
Scan abort
Scan for service with custom uuid 112233445566778899AABBCCDDEEFF00
EVDISCPRIMSVC : cHndl=2804 svcUuid=FC033344 sHndl=4 eHndl=6
EVDISCPRIMSVC : cHndl=2804 svcUuid=FC033344 sHndl=13 eHndl=15
EVDISCPRIMSVC : cHndl=2804 svcUuid=00000000 sHndl=0 eHndl=0
Scan complete

- Disconnected
Exiting...
```

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

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

If a GATT table has a specific service and a specific characteristic, then it is more efficient to locate details of that characteristic by using the function `BleGATTcFindChar()`. This function is described later.

While the scan is in progress and waiting for the next piece of data from a GATT server, the module enters low power state as the `WAITEVENT` statement is used as normal to wait for events and messages.

Depending on the size of the remote GATT server table and the connection interval, the scan of all characteristics may take many hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

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

### **BLEDISCCHARFIRST** (*connHandle, charUuidHandle, startAttrHandle, endAttrHandle*)

A typical pseudo code for discovering characteristic involves first calling `BleDiscCharFirst()` with information obtained from a primary services scan, waiting for the `EVDISCCHAR` event message, and (depending on the information returned in that message) calling `BleDiscCharNext()`. This in turn results in another `EVDISCCHAR` event message and typically is as follows:

```
Register a handler for the EVDISCCHAR event message
```

```
On EVDISCCHAR event message
```

```
    If Char Value Handle == 0 then scan is complete
```

```
    Else Process information then
```

```
        call BleDiscCharNext()
```

```
        if BleDiscCharNext() not OK then scan complete
```

```
Call BleDiscCharFirst( --information from EVDISCPRIMSVC )
```

```
If BleDiscCharFirst() ok then Wait for EVDISCCHAR
```

<b>Returns</b>	<p>INTEGER, a result code.</p> <p>The typical value is 0x0000, indicating a successful operation and it means an <code>EVDISCCHAR</code> event message is thrown by the <i>smart</i>BASIC runtime engine containing the results. A non-zero return value implies an <code>EVDISCCHAR</code> message is not thrown.</p>
<b>Arguments:</b>	
<b>connHandle</b>	<p><b>byVal nConnHandle AS INTEGER</b></p> <p>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 <code>EVBLEMSG</code> event message with <code>msgId == 0</code> and <code>msgCtx</code> is the connection handle.</p>
<b>charUuidHandle</b>	<p><b>byVal charUuidHandle AS INTEGER</b></p> <p>Set this to 0 if you want to scan for any characteristic in the service, otherwise this value is generated either by <code>BleHandleUuid16()</code> or <code>BleHandleUuid128()</code> or <code>BleHandleUuidSibling()</code>.</p>
<b>startAttrHandle</b>	<p><b>byVal startAttrHandle AS INTEGER</b></p> <p>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.</p>
<b>endAttrHandle</b>	<p><b>byVal endAttrHandle AS INTEGER</b></p> <p>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.</p>

### BLEDISCCHARNEXT (*connHandle*)

Calling this assumes that BleDiscCharFirst() has been called at least once to set up the internal characteristics scanning state machine. It scans for the next characteristic.

<b>Returns</b>	INTEGER, a result code. The typical value is 0x0000, indicating a successful operation. It means an EVDISCCCHAR event message is thrown by the <i>smart</i> BASIC runtime engine containing the results. A non-zero return value implies an EVDISCCCHAR message is not thrown.
<b>Arguments:</b>	
<b><i>connHandle</i></b>	<b>byVal nConnHandle AS INTEGER</b> This is the connection handle as returned in the on-connect event for the connection on which the remote GATT server can be accessed. This is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.

#### Example:

```
//Example :: BleDiscCharFirst.Next.sb (See in BT900CodeSnippets.zip)
//
//Remote server has 1 prim service with 16 bit uuid and 8 characteristics where
// 5 uuids are 16 bit and 3 are 128 bit
// 3 of the 16 bit uuid are the same value 0xDEAD and
// 2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGATTcTblDiscChar.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sAttr,eAttr

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
```

```

ENDSUB

//=====
// Ble event handler
//=====
FUNCTION HndlrBleMsg (BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote GATT Table for first service"
        PRINT "\n- and a characteristic scan will be initiated in the event"
        rc = BleDiscServiceFirst (conHndl,0,0)
        IF rc==0 THEN
            //wait for start and end handles for first primary service
            WAITEVENT
            PRINT "\n\nScan for characteristic with uuid = 0xDEAD"
            uHndl = BleHandleUuid16 (0xDEAD)
            rc = BleDiscCharFirst (conHndl,uHndl,sAttr,eAttr)
            IF rc == 0 THEN
                //HandlerCharDisc() will exit with 0 when operation is complete
                WAITEVENT
                uu$ = "112233445566778899AABBCCDDEEFF00"
                PRINT "\n\nScan for service with custom uuid ";uu$
                uu$ = StrDehexize$(uu$)
                uHndl = BleHandleUuid128 (uu$)
                rc = BleDiscCharFirst (conHndl,uHndl,sAttr,eAttr)
                IF rc==0 THEN
                    //HandlerCharDisc() will exit with 0 when operation is complete
                    WAITEVENT
                ENDIF
            ENDIF
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

//=====
// EVDISCPRIMSVC event handler
//=====
FUNCTION HandlerPrimSvc (cHndl, svcUuid, sHndl, eHndl) AS INTEGER
    PRINT "\nEVDISCPRIMSVC :"
    PRINT " cHndl=";cHndl

```

```

PRINT " svcUuid=";integer.h' svcUuid
PRINT " sHndl=";sHndl
PRINT " eHndl=";eHndl
IF sHndl == 0 THEN
    PRINT "\nPrimary Service Scan complete"
    EXITFUNC 0
ELSE
    PRINT "\nGot first primary service so scan for ALL characteristics"
    sAttr = sHndl
    eAttr = eHndl
    rc = BleDiscCharFirst(conHndl,0,sAttr,eAttr)
    IF rc != 0 THEN
        PRINT "\nScan characteristics failed"
        EXITFUNC 0
    ENDIF
ENDIF
endfunc 1

'//=====
// EVDISCCCHAR event handler
'//=====
function HandlerCharDisc(cHndl,cUuid,cProp,hVal,isUuid) as integer
    print "\nEVDISCCCHAR :"
    print " cHndl=";cHndl
    print " chUuid=";integer.h' cUuid
    print " Props=";cProp
    print " valHndl=";hVal
    print " ISvcUuid=";isUuid
    IF hVal == 0 THEN
        PRINT "\nCharacteristic Scan complete"
        EXITFUNC 0
    ELSE
        rc = BleDiscCharNext(conHndl)
        IF rc != 0 THEN
            PRINT "\nCharacteristics scan abort"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1

//=====
// Main() equivalent
//=====
ONEVENT EVBLEMSG          CALL HndlrBleMsg
OnEvent EVDISCPRIMSVCS   call HandlerPrimSvc

```

```
OnEvent EVDISCCCHAR      call HandlerCharDisc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF
WAITEVENT
PRINT "\nExiting..."
```



**Expected Output:**

```
Advertising, and GATT Client is open

- Connected, so scan remote GATT Table for first service
- and a characteristic scan will be initiated in the event
EVDISCPRIMSVC : cHndl=3549 svcUuid=FE01FE02 sHndl=1 eHndl=17
Got first primary service so scan for ALL characteristics
EVDISCCHAR : cHndl=3549 chUuid=FE01FC21 Props=2 valHndl=3 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=5 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=7 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FB04BEEF Props=2 valHndl=9 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=11 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01FC23 Props=2 valHndl=13 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=15 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=17 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=00000000 Props=0 valHndl=0 ISvcUuid=0
Characteristic Scan complete

Scan for characteristic with uuid = 0xDEAD
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=7 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=15 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=17 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=00000000 Props=0 valHndl=0 ISvcUuid=0
Characteristic Scan complete

Scan for service with custom uuid 112233445566778899AABBCCDDEEFF00
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=5 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=11 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=00000000 Props=0 valHndl=0 ISvcUuid=0
Characteristic Scan complete

- Disconnected
Exiting...
```

## BleDiscDescFirst /BleDiscDescNext

### FUNCTIONS

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

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

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

While the scan is in progress and waiting for the next piece of data from a GATT server, the module enters low power state as the WAITEVENT statement is used as normal to wait for events and messages.

Depending on the size of the remote GATT server table and the connection interval, the scan of all descriptors may take many hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

### BLEDISCDESCFIRST (connHandle, descUuidHandle, charValHandle)

A typical pseudo code for discovering descriptors involves first calling BleDiscDescFirst() with information obtained from a characteristics scan and then waiting for the EVDISCDESC event message. Depending on the information returned in that message, calling BleDiscDescNext() results in another EVDISCDESC event message and typically is as follows:

```
Register a handler for the EVDISCDESC event message

On EVDISCDESC event message
    If Descriptor Handle == 0 then scan is complete
    Else Process information then
        call BleDiscDescNext()
        if BleDiscDescNext() not OK then scan complete

Call BleDiscDescFirst( --information from EVDISCCHAR )
If BleDiscDescFirst() ok then Wait for EVDISCDESC
```

#### Returns

INTEGER, a result code.

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

#### Arguments:

<b>connHandle</b>	<b>byVal nConnHandle AS INTEGER</b> 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.
<b>descUuidHandle</b>	<b>byVal descUuidHandle AS INTEGER</b> 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().

<b><i>charValHandle</i></b>	<b>byVal <i>charValHandle</i> AS INTEGER</b> 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.
-----------------------------	--

### BLEDISCDISCNEXT (*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.

<b>Returns</b>	INTEGER, a result code.  The typical value is 0x0000, indicating a successful operation and it means an EVDISCDISC event message is thrown by the <i>smart</i> BASIC runtime engine containing the results. A non-zero return value implies an EVDISCDISC message is not thrown.
<b>Arguments:</b>	
<b><i>connHandle</i></b>	<b>byVal <i>nConnHandle</i> AS INTEGER</b> 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 <code>msgId == 0</code> and <code>msgCtx</code> is the connection handle.

#### Example:

```
//Example :: BleDiscDescFirst.Next.sb (See in BT900CodeSnippets.zip)
//
//Remote server has 1 prim service with 16 bit uuid and 1 characteristics
// which contains 8 descriptors, that are ...
// 5 uuids are 16 bit and 3 are 128 bit
// 3 of the 16 bit uuid are the same value 0xDEAD and
// 2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGATTcTblDiscDesc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sAttr,eAttr,cValAttr

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
```

```
//open the GATT client with default notify/indicate ring buffer size
IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//=====
// Ble event handler
//=====

FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote GATT Table for first service"
        PRINT "\n- and a characteristic scan will be initiated in the event"
        rc = BleDiscServiceFirst(conHndl,0,0)
        IF rc==0 THEN
            //wait for start and end handles for first primary service
            WAITEVENT
            PRINT "\n\nScan for descriptors with uuid = 0xDEAD"
            uHndl = BleHandleUuid16(0xDEAD)
            rc = BleDiscDescFirst(conHndl,uHndl,cValAttr)
            IF rc == 0 THEN
                //HandlerDescDisc() will exit with 0 when operation is complete
                WAITEVENT
                uu$ = "112233445566778899AABBCCDDEEFF00"
                PRINT "\n\nScan for service with custom uuid ";uu$
                uu$ = StrDehexize$(uu$)
                uHndl = BleHandleUuid128(uu$)
                rc = BleDiscDescFirst(conHndl,uHndl,cValAttr)
                IF rc==0 THEN
```

```
        //HandlerDescDisc() will exit with 0 when operation is complete
        WAITEVENT
    ENDIF
ENDIF
ENDIF
    CloseConnections()
ENDIF
ENDFUNC 1

//=====
// EVDISCPRIMSV event handler
//=====
FUNCTION HandlerPrimSvc(cHndl,svcUuid,sHndl,eHndl) AS INTEGER
    PRINT "\nEVDISCPRIMSVC : "
    PRINT " cHndl=";cHndl
    PRINT " svcUuid=";integer.h' svcUuid
    PRINT " sHndl=";sHndl
    PRINT " eHndl=";eHndl
    IF sHndl == 0 THEN
        PRINT "\nPrimary Service Scan complete"
        EXITFUNC 0
    ELSE
        PRINT "\nGot first primary service so scan for ALL characteristics"
        sAttr = sHndl
        eAttr = eHndl
        rc = BleDiscCharFirst(conHndl,0,sAttr,eAttr)
        IF rc != 0 THEN
            PRINT "\nScan characteristics failed"
            EXITFUNC 0
        ENDIF
    ENDIF
ENDFUNC 1

'//=====
'// EVDISCCCHAR event handler
'//=====
function HandlerCharDisc(cHndl,cUuid,cProp,hVal,isUuid) as integer
    print "\nEVDISCCCHAR : "
    print " cHndl=";cHndl
```

```
print " chUuid=";integer.h' cUuid
print " Props=";cProp
print " valHndl=";hVal
print " ISvcUuid=";isUuid
IF hVal == 0 THEN
    PRINT "\nCharacteristic Scan complete"
    EXITFUNC 0
ELSE
    PRINT "\nGot first characteristic service at handle ";hVal
    PRINT "\nScan for ALL Descs"
    cValAttr = hVal
    rc = BleDiscDescFirst(conHndl,0,cValAttr)
    IF rc != 0 THEN
        PRINT "\nScan descriptors failed"
        EXITFUNC 0
    ENDIF
ENDIF
endfunc 1

'//=====
// EVDISCDESC event handler
'//=====
function HandlerDescDisc(cHndl,cUuid,hndl) as integer
    print "\nEVDISCDESC"
    print " cHndl=";cHndl
    print " dscUuid=";integer.h' cUuid
    print " dscHndl=";hndl
    IF hndl == 0 THEN
        PRINT "\nDescriptor Scan complete"
        EXITFUNC 0
    ELSE
        rc = BleDiscDescNext(cHndl)
        IF rc != 0 THEN
            PRINT "\nDescriptor scan abort"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1
```

```
//=====
// Main() equivalent
//=====

ONEVENT EVBLEMSG      CALL HndlrBleMsg
OnEvent EVDISCPRIMSVC  call HandlerPrimSvc
OnEvent EVDISCCCHAR    call HandlerCharDisc
OnEvent EVDISCDESC     call HandlerDescDisc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

**Expected Output:**

```
Advertising, and GATT Client is open

- Connected, so scan remote GATT Table for first service
- and a characteristic scan will be initiated in the event
EVDISCPRIMSVC : cHndl=3790 svcUuid=FE01FE02 sHndl=1 eHndl=11
Got first primary service so scan for ALL characteristics
EVDISCCHAR : cHndl=3790 chUuid=FE01FC21 Props=2 valHndl=3 ISvcUuid=0
Got first characteristic service at handle 3
Scan for ALL Descs
EVDISCDESC cHndl=3790 dscUuid=FE01FD21 dscHndl=4
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=5
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=6
EVDISCDESC cHndl=3790 dscUuid=FB04BEEF dscHndl=7
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=8
EVDISCDESC cHndl=3790 dscUuid=FE01FD23 dscHndl=9
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=10
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=11
EVDISCDESC cHndl=3790 dscUuid=00000000 dscHndl=0
Descriptor Scan complete

Scan for descriptors with uuid = 0xDEAD
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=6
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=10
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=11
EVDISCDESC cHndl=3790 dscUuid=00000000 dscHndl=0
Descriptor Scan complete

Scan for service with custom uuid 112233445566778899AABBCCDDEEFF00
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=5
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=8
EVDISCDESC cHndl=3790 dscUuid=00000000 dscHndl=0
Descriptor Scan complete

- Disconnected
Exiting...
```

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

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

### BLEGATTCFINDCHAR (connHandle, svcUuidHndl, svcIndex, charUuidHndl, charIndex)

A typical pseudo code for finding a characteristic involves calling BleGATTcFindChar() which in turn will result in the EVFINDCHAR event message and typically is as follows:-

```
Register a handler for the EVFINDCHAR event message
```

```
On EVFINDCHAR event message
  If Char Value Handle == 0 then
    Characteristic not found
  Else
    Characteristic has been found
```

```
Call BleGATTcFindChar()
If BleGATTcFindChar () ok then Wait for EVFINDCHAR
```

<b>Returns</b>	<p>INTEGER, a result code.</p> <p>The typical value is 0x0000, indicating a successful operation and it means an EVFINDCHAR event message is thrown by the <i>smart</i>BASIC runtime engine containing the results. A non-zero return value implies an EVFINDCHAR message is not thrown.</p>
<b>Arguments:</b>	
<b>connHandle</b>	<p><b>byVal nConnHandle AS INTEGER</b></p> <p>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.</p>
<b>svcUuidHndl</b>	<p><b>byVal svcUuidHndl AS INTEGER</b></p> <p>Set this to the service UUID handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().</p>
<b>svcIndex</b>	<p><b>byVal svcIndex AS INTEGER</b></p> <p>This is the instance of the service to look for with the UUID handle svcUuidHndl, where 0 is the first instance, 1 is the second, and so on.</p>

<b><i>charUuidHndl</i></b>	<b>byVal <i>charUuidHndl</i> AS INTEGER</b> Set this to the characteristic UUID handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
<b><i>charIndex</i></b>	<b>byVal <i>charIndex</i> AS INTEGER</b> This is the instance of the characteristic to look for with the UUID handle <i>charUuidHndl</i> , where 0 is the first instance, 1 is the second, and so on.

**Example:**

```
//Example :: BleGATTcFindChar.sb (See in BT900CodeSnippets.zip)
//
//Remote server has 5 prim services with 16 bit uuid and 3 with 128 bit uuids
// 3 of the 16 bit uuid are the same value 0xDEAD and
// 2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGATTcTblFindChar.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sIdx,cIdx

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB
```

```
//=====
// Ble event handler
//=====
FUNCTION HndlrBleMsg (BYVAL nMsgId, BYVAL nCtx)
    DIM uu$, uHndS, uHndC
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote GATT Table for an instance of char"
        uHndS = BleHandleUuid16(0xDEAD)
        uu$ = "112233445566778899AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndC = BleHandleUuid128(uu$)
        sIdx = 2
        cIdx = 1 //valHandle will be 32
        rc = BleGattcFindChar(conHndl, uHndS, sIdx, uHndC, cIdx)

        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF

        sIdx = 1
        cIdx = 3 //does not exist

        rc = BleGattcFindChar(conHndl, uHndS, sIdx, uHndC, cIdx)

        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF

        CloseConnections()
    ENDIF
ENDFUNC 1

'//=====
'//=====

function HandlerFindChar (cHndl, cProp, hVal, isUuid) as integer
    print "\nEVFINDCHAR "
```

```
print " cHndl=";cHndl
print " Props=";cProp
print " valHndl=";hVal
print " ISvcUuid=";isUuid
IF hVal == 0 THEN
    PRINT "\nDid NOT find the characteristic"
ELSE
    PRINT "\nFound the characteristic at handle ";hVal
    PRINT "\nSvc Idx=";sIdx;" Char Idx=";cIdx
ENDIF
endfunc 0

//=====
// Main() equivalent
//=====

ONEVENT EVBLEMSG      CALL HndlrBleMsg
OnEvent EVFINDCHAR    call HandlerFindChar

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

### Expected Output:

```
Advertising, and GATT Client is open

- Connected, so scan remote GATT Table for an instance of char
EVFINDCHAR cHndl=866 Props=2 valHndl=32 ISvcUuid=0
Found the characteristic at handle 32
Svc Idx=2 Char Idx=1
EVFINDCHAR cHndl=866 Props=0 valHndl=0 ISvcUuid=0
Did NOT find the characteristic

- Disconnected
Exiting...
```

## BleGattcFindDesc

### FUNCTION

This function facilitates an efficient way of locating the details of a descriptor if the UUID is known along with the UUID of the service and the UUID of the characteristic containing it. The results are delivered in a EVFINDDESC event message. If the GATT server table has multiple instances of the same service/characteristic/descriptor combination then this function works because, in addition to the UUID handles to be searched for, it accepts instance parameters which are indexed from 0. This means that the second instance of a descriptor in the fourth instance of a characteristic with the same UUID in the third instance of a service with the same UUID is located with index values 1, 3, and 2 respectively.

Given that the results are returned in an event message, a handler must be registered for the EVFINDDESC event.

Depending on the size of the remote GATT server table and the connection interval, the search of the characteristic may take many hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

---

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

---

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

A typical pseudo code for finding a 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
```

<b>Returns</b>	<p>INTEGER, a result code.</p> <p>The typical value is 0x0000, indicating a successful operation and it means an EVFINDDESC event message is thrown by the <i>smart</i>BASIC runtime engine containing the results. A non-zero return value implies an EVFINDDESC message is not thrown</p>
<b>Arguments:</b>	
<b>connHndl</b>	<p><b>byVal connHndl AS INTEGER</b></p> <p>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.</p>
<b>svcUuHndl</b>	<p><b>byVal svcUuHndl AS INTEGER</b></p> <p>Set this to the service UUID handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().</p>
<b>svclIdx</b>	<p><b>byVal svclIdx AS INTEGER</b></p> <p>This is the instance of the service to look for with the UUID handle svcUuidHndl, where 0 is the first instance, 1 is the second, and so on.</p>
<b>charUuHndl</b>	<p><b>byVal charUuHndl AS INTEGER</b></p> <p>Set this to the characteristic UUID handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().</p>
<b>charIdx</b>	<p><b>byVal charIdx AS INTEGER</b></p> <p>This is the instance of the characteristic to look for with the UUID handle charUuidHndl, where 0 is the first instance, 1 is the second, and so on.</p>
<b>descUuHndl</b>	<p><b>byVal descUuHndl AS INTEGER</b></p> <p>Set this to the descriptor uuid handle which is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().</p>
<b>descIdx</b>	<p><b>byVal descIdx AS INTEGER</b></p> <p>This is the instance of the descriptor to look for with the UUID handle charUuidHndl, where 0 is the first instance, 1 is the second, and so on.</p>

#### Example:

```
//Example :: BleGATTcFindDesc.sb (See in BT900CodeSnippets.zip)
//
//Remote server has 5 prim services with 16 bit uuid and 3 with 128 bit uuids
// 3 of the 16 bit uuid are the same value 0xDEAD and
// 2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGATTcTblFindDesc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sIdx,cIdx,dIdx
```

```
//=====
// Initialise and instantiate service, characteristic, start adverts
//=====
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

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

//=====
// Ble event handler
//=====
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$, uHndS, uHndC, uHndD
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote GATT Table for ALL services"
        uHndS = BleHandleUuid16(0xDEAD)
        uu$ = "112233445566778899AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndC = BleHandleUuid128(uu$)
        uu$ = "1122C0DE5566778899AABBCCDDEEFF00"
```

```
uu$ = StrDehexize$(uu$)
uHndD = BleHandleUuid128(uu$)
sIdx = 2
cIdx = 1
dIdx = 1 // handle will be 37
rc = BleGattcFindDesc(conHndl,uHndS,sIdx,uHndC,cIdx,uHndD,dIdx)
IF rc==0 THEN
    //BleDiscCharFirst() will exit with 0 when operation is complete
    WAITEVENT
ENDIF
sIdx = 1
cIdx = 3
dIdx = 4 //does not exist
rc = BleGattcFindDesc(conHndl,uHndS,sIdx,uHndC,cIdx,uHndD,dIdx)
IF rc==0 THEN
    //BleDiscCharFirst() will exit with 0 when operation is complete
    WAITEVENT
ENDIF
CloseConnections()
ENDIF
ENDFUNC 1

'//=====
'//=====
function HandlerFindDesc(cHndl,hndl) as integer
    print "\nEVFINDDDESC "
    print " cHndl=";cHndl
    print " dscHndl=";hndl
    IF hndl == 0 THEN
        PRINT "\nDid NOT find the descriptor"
    ELSE
        PRINT "\nFound the descriptor at handle ";hndl
        PRINT "\nSvc Idx=";sIdx;" Char Idx=";cIdx;" desc Idx=";dIdx
    ENDIF
endfunc 0

//=====
// Main() equivalent
```



```
//=====
ONEVENT EVBLEMSG      CALL HndlrBleMsg
OnEvent EVFINDDESC    call HandlerFindDesc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

#### Expected Output:

```
Advertising, and GATT Client is open

- Connected, so scan remote GATT Table for ALL services
EVFINDDESC cHndl=1106 dscHndl=37
Found the descriptor at handle 37
Svc Idx=2 Char Idx=1 desc Idx=1
EVFINDDESC cHndl=1106 dscHndl=0
Did NOT find the descriptor

- Disconnected
Exiting...
```

## BleGattcRead/BleGattcReadData

### FUNCTIONS

If the handle for an attribute is known, then these functions are used to read the content of that attribute from a specified offset in the array of octets in that attribute value.

Given that the success or failure of this read operation is returned in an event message, a handler **must** be registered for the EVATTRREAD event.

Depending on the connection interval, the read of the attribute may take many hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

BleGATTcRead is used to trigger the procedure and BleGattcReadData is used to read the data from the underlying cache when the EVATTRREAD event message is received with a success status.

### BLEGATTCREAD (connHndl, attrHndl, offset)

A typical pseudo code for reading the content of an attribute calling BleGattcRead() which in turn results in the EVATTRREAD event message and typically is as follows:

```
Register a handler for the EVATTRREAD event message

On EVATTRREAD event message
  If GATT_Status == 0 then
    BleGattcReadData() //to actually get the data
  Else
    Attribute could not be read

Call BleGATTcRead()
If BleGattcRead() ok then Wait for EVATTRREAD
```

<b>Returns</b>	<p>INTEGER, a result code.</p> <p>The typical value is 0x0000, indicating a successful operation and it means an EVATTRREAD event message is thrown by the <i>smart</i>BASIC runtime engine containing the results. A non-zero return value implies an EVATTRREAD message is not thrown.</p>
<b>Arguments:</b>	
<b>connHndl</b>	<p><b>byVal connHndl AS INTEGER</b></p> <p>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.</p>
<b>attrHndl</b>	<p><b>byVal attrHndl AS INTEGER</b></p> <p>Set to the handle of the attribute to read. It is a value in the range 1 to 65535.</p>
<b>offset</b>	<p><b>byVal offset AS INTEGER</b></p> <p>This is the offset from which the data in the attribute is to be read.</p>

## 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.

<b>Returns</b>	INTEGER, a result code. The typical value is 0x0000, indicating a successful read.
<b>Arguments:</b>	
<b><i>connHndl</i></b>	<b>byVal <i>connHndl</i> AS INTEGER</b> 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 <i>msgId</i> == 0 and <i>msgCtx</i> is the connection handle.
<b><i>attrHndl</i></b>	<b>byRef <i>attrHndl</i> AS INTEGER</b> The handle for the attribute that was read is returned in this variable. It is the same as the one supplied in <i>BleGATTcRead</i> , but supplied here so that the code can be stateless.
<b><i>offset</i></b>	<b>byRef <i>offset</i> AS INTEGER</b> The offset into the attribute data that was read is returned in this variable. It is the same as the one supplied in <i>BleGATTcRead</i> , but supplied here so that the code can be stateless.
<b><i>attrData\$</i></b>	<b>byRef <i>attrData\$</i> AS STRING</b> The attribute data which was read is supplied in this parameter.

### Example:

```
//Example :: BleGATTcRead.sb (See in BT900CodeSnippets.zip)
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGattcTblRead.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,nOff,atHndl

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc
```

```
//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//=====
// Ble event handler
//=====

FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so read attribute handle 3"
        atHndl = 3
        nOff = 0
        rc=BleGattcRead(conHndl,atHndl,nOff)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\nread attribute handle 300 which does not exist"
        atHndl = 300
        nOff = 0
        rc=BleGattcRead(conHndl,atHndl,nOff)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//=====
'//=====

function HandlerAttrRead(cHndl,aHndl,nSts) as integer
```

```
dim nOfst,nAhndl,at$
print "\nEVATTRREAD "
print " cHndl=";cHndl
print " attrHndl=";aHndl
print " status=";integer.h' nSts
if nSts == 0 then
    print "\nAttribute read OK"
    rc = BleGattcReadData(cHndl,nAhndl,nOfst,at$)
    print "\nData  = ";StrHexize$(at$)
    print " Offset= ";nOfst
    print " Len=";strlen(at$)
    print "\nhandle = ";nAhndl
else
    print "\nFailed to read attribute"
endif
endfunc 0
//=====
// Main() equivalent
//=====
ONEVENT EVBLEMSG      CALL HndlrBleMsg
OnEvent EVATTRREAD    call HandlerAttrRead

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

### Expected Output:

```
Advertising, and GATT Client is open

- Connected, so read 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...
```

## BleGattcWrite

### FUNCTION

If the handle for an attribute is known then this function is used to write into an attribute starting at offset 0. The acknowledgement is returned via a EVATTRWRITE event message.

Given that the success or failure of this write operation is returned in an event message, a handler **must** be registered for the EVATTRWRITE event.

Depending on the connection interval, the write to the attribute may take many hundreds of milliseconds. While this is in progress, it is safe to do other non GATT related operations such as servicing sensors and displays or any of the onboard peripherals.

### 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:**

<b>connHndl</b>	<b>byVal connHndl AS INTEGER</b> 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.
<b>attrHndl</b>	<b>byVal attrHndl AS INTEGER</b> The handle for the attribute that is to be written to.
<b>attrData\$</b>	<b>byRef attrData\$ AS STRING</b> The attribute data to write.

**Example:**

```
//Example :: BleGATTcWrite.sb (See in BT900CodeSnippets.zip)
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGATTcTblWrite.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,atHndl

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
```

```

ENDSUB

//=====
// Ble event handler
//=====
FUNCTION HndlrBleMsg (BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so write to 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

'//=====
'//=====
function HandlerAttrWrite(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRWRITE "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute write OK"
    
```



```
else
    print "\nFailed to write attribute"
endif
endfunc 0

//=====
// Main() equivalent
//=====

ONEVENT EVBLEMSG      CALL HndlrBleMsg
OnEvent EVATTRWRITE   call HandlerAttrWrite

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

### Expected Output:

```
Advertising, and GATT Client is open

- Connected, so read 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...
```

## BleGattcWriteCmd

### FUNCTION

If the handle for an attribute is known, then this function is used to write into an attribute at offset 0 when no acknowledgment response is expected. The signal that the command has actually been transmitted and that the remote link layer has acknowledged is by the EVNOTIFYBUF event.

**Note:** The acknowledgement received for the BleGattcWrite() command is from the higher level GATT layer. Do not confuse this with the link layer ACK .

*All packets are acknowledged at link layer level. If a packet fails to get through, then that condition manifests as a connection drop due to the link supervision timeout.*

Given that the transmission and link layer ACK of this write operation is indicated in an event message, a handler **must** be registered for the EVNOTIBUF event.

Depending on the connection interval, the write to the attribute may take many hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

### 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
```

<b>Returns</b>	INTEGER, a result code. The typical value is 0x0000, indicating a successful read.
<b>Arguments:</b>	
<b>connHndl</b>	<b>byVal connHndl AS INTEGER</b> 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.
<b>attrHndl</b>	<b>byVal attrHndl AS INTEGER</b> The handle for the attribute that is to be written to.
<b>attrData\$</b>	<b>byRef attrData\$ AS STRING</b> The attribute data to write.

### Example:

```
//Example :: BleGATTcWriteCmd.sb (See in BT900CodeSnippets.zip)
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGATTcTblWriteCmd.sub invoked in _OpenMcp.scr
```

```
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,atHndl

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====

FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

//=====
// Close connections so that we can run another app without problems
//=====

SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//=====
// Ble event handler
//=====

FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so write to attribute handle 3"
        atHndl = 3
        at$="\01\02\03\04"
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
```

```

    IF rc==0 THEN
        WAITEVENT
    ENDIF
    PRINT "\n- write again to attribute handle 3"
    atHndl = 3
    at$="\05\06\07\08"

    rc=BleGattcWriteCmd(conHndl,atHndl,at$)

    IF rc==0 THEN
        WAITEVENT
    ENDIF
    PRINT "\n- write again to attribute handle 3"
    atHndl = 3
    at$="\09\0A\0B\0C"

    rc=BleGattcWriteCmd(conHndl,atHndl,at$)

    IF rc==0 THEN
        WAITEVENT
    ENDIF
    PRINT "\nwrite to attribute handle 300 which does not exist"
    atHndl = 300

    rc=BleGattcWriteCmd(conHndl,atHndl,at$)

    IF rc==0 THEN
        PRINT "\nEven when the attribute does not exist an event will occur"
        WAITEVENT
    ENDIF
    CloseConnections()
ENDIF
ENDFUNC 1

'//=====
'//=====

function HandlerNotifyBuf() as integer
    print "\nEVNOTIFYBUF Event"
endfunc 0 '//need to progress the WAITEVENT

//=====
// Main() equivalent
//=====

ONEVENT EVBLEMSG          CALL HndlrBleMsg
OnEvent EVNOTIFYBUF       call HandlerNotifyBuf

```

```
IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

#### Expected Output:

```
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...
```

## BleGattcNotifyRead

### 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 *smart*BASIC runtime engine.

Data arriving via a notification does not require GATT acknowledgements, however indications require them. This GATT client manager saves data arriving via a notification in the same ring buffer for later extraction using the command `BleGattcNotifyRead()`; for indications, an automatic GATT acknowledgement is sent when the data is saved in the ring buffer. This acknowledgment happens even if the data is discarded because the ring buffer is full. If the data must not be acknowledged when it is discarded on a full buffer, set the flags parameter in the `BleGattcOpen()` function where the GATT client manager is opened.

In the case when an ACK is NOT sent on data discard, the GATT server is throttled and no further data is notified or indicated by it until BleGattNotifyRead() is called to extract data from the ring buffer to create space and it triggers a delayed acknowledgement.

When the GATT client manager is opened using BleGattcOpen(), it is possible to specify the size of the ring buffer. If a value of 0 is supplied, then a default size is created. SYSINFO(2019) in a 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, a EVATTRNOTIFY event is thrown to the smartBASIC runtime engine. This is an event, in the same way an incoming UART receive character generates an event; that is, no data payload is attached to the event.

#### **BLEGATTCTNOTIFYREAD (connHndl, attrHndl, attrData\$, discardCount)**

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

```
Register a handler for the EVATTRNOTIFY event message

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

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

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

#### **Example:**

```
//Example :: BleGATTcNotifyRead.sb (See in BT900CodeSnippets.zip)
//
// Server created using BleGattcTblNotifyRead.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000
//
// Charactersitic at handle 15 has notify (16==cccd)
// Charactersitic at handle 18 has indicate (19==cccd)
```

```
DIM rc,at$,conHndl,uHndl,atHndl

//=====
// Initialise and instantiate service, characteristic, start adverts
//=====
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the GATT client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

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

//=====
// Ble event handler
//=====
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so enable notification for char with cccd at 16"
        atHndl = 16
        at$="\01\00"
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
    ENDIF
ENDIF
```

```

PRINT "\n- enable indication for char with cccd at 19"
atHndl = 19
at$="\02\00"
rc=BleGattcWrite(conHndl,atHndl,at$)
IF rc==0 THEN
    WAITEVENT
ENDIF
ENDIF
ENDFUNC 1

'//=====
'//=====

function HandlerAttrWrite(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRWRITE "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute write OK"
    else
        print "\nFailed to write attribute"
    endif
endfunc 0

'//=====
'//=====

function HandlerAttrNotify() as integer
    dim chndl,aHndl,att$,dscd
    print "\nEVATTRNOTIFY Event"
    rc=BleGattcNotifyRead(cHndl,aHndl,att$,dscd)
    print "\n BleGattcNotifyRead()"
    if rc==0 then
        print " cHndl=";cHndl
        print " attrHndl=";aHndl
        print " data=";StrHexize$(att$)
        print " discarded=";dscd
    else
        print " failed with ";integer.h' rc
    
```



```
endif
endfunc 1

//=====
// Main() equivalent
//=====

ONEVENT EVBLEMSG      CALL HndlrBleMsg
OnEvent EVATTRWRITE   call HandlerAttrWrite
OnEvent EVATTRNOTIFY  call HandlerAttrNotify

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and GATT Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

#### Expected Output:

```
Advertising, and GATT Client is open

- Connected, so enable notification for char with cccd at 16
EVATTRWRITE cHndl=877 attrHndl=16 status=00000000
Attribute write OK
- enable indication for char with cccd at 19
EVATTRWRITE cHndl=877 attrHndl=19 status=00000000
Attribute write OK
EVATTRNOTIFY Event
  BleGATTcNotifyRead() cHndl=877 attrHndl=15 data=BAADC0DE discarded=0
EVATTRNOTIFY Event
  BleGATTcNotifyRead() cHndl=877 attrHndl=18 data=DEADBEEF discarded=0
EVATTRNOTIFY Event
  BleGATTcNotifyRead() cHndl=877 attrHndl=15 data=BAADC0DE discarded=0
EVATTRNOTIFY Event
  BleGATTcNotifyRead() cHndl=877 attrHndl=18 data=DEADBEEF discarded=0
```

## Attribute Encoding Functions

Data for characteristics are stored in value attributes, arrays of bytes. Multibyte Characteristic Descriptors content is stored similarly. Those bytes are manipulated in *smart*BASIC applications using STRING variables.

The Bluetooth specification stipulates that multibyte data entities are stored in little endian format and so all data manipulation is done similarly. Little endian means that a multibyte data entity is stored so that lowest significant byte is positioned at the lowest memory address and likewise, when transported, the lowest byte is on the wire first.

This section describes all the encoding functions which allow those strings to be written in smaller bitwise subfields in a more efficient manner compared to the generic STRXXXX functions that are made available in *smart*BASIC.

---

**Note:** CCCD and SCCD descriptors are special cases; they have two bytes which are treated as 16-bit integers. This is reflected in *smart*BASIC applications so that INTEGER variables are used to manipulate those values instead of STRINGS.

---

### BleEncode8

#### FUNCTION

This function overwrites a single byte in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

#### BLEENCODE8 (attr\$,nData, nIndex)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This argument is the string that is written to an attribute.
<b>nData</b>	<b>byVal nData AS INTEGER</b> The least significant byte of this integer is saved. The rest is ignored.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero-based index into the string attr\$ where the new data fragment is written to. If the string attr\$ is not long enough to fit the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

#### Example:

```
//Example :: BleEncode8.sb (See in BT900CodeSnippets.zip)

DIM rc
DIM attr$

attr$="Laird"
```

```
PRINT "\nattr$=";attr$

//Remember: - 4 bytes are used to store an integer on the BT900

//write 'C' to index 2 -- '111' will be ignored
rc=BleEncode8(attr$,0x11143,2)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)
//write 'B' to index 1
rc=BleEncode8(attr$,0x42,1)
//write 'D' to index 3
rc=BleEncode8(attr$,0x44,3)
//write 'y' to index 7 -- attr$ will be extended
rc=BleEncode8(attr$,0x67, 7)

PRINT "\nattr$ now = ";attr$
```

#### Expected Output:

```
attr$=Laird
attr$ now = ABCDd\00\00g
```

## 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)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This argument is the string that is written to an attribute.
<b>nData</b>	<b>byVal nData AS INTEGER</b> The two least significant bytes of this integer is saved. The rest is ignored.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ where the new fragment of data is written. If the string attr\$ is not long enough to accommodate the index plus the length of the fragment, it

is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

#### Example:

```
//Example :: BleEncode16.sb (See in BT900CodeSnippets.zip)

DIM rc, attr$
attr$="Laird"
PRINT "\nattr$=";attr$

//write 'CD' to index 2
rc=BleEncode16(attr$,0x4443,2)
//write 'AB' to index 0 - '2222' will be ignored
rc=BleEncode16(attr$,0x22224241,0)
//write 'EF' to index 3
rc=BleEncode16(attr$,0x4645,4)

PRINT "\nattr$ now = ";attr$
```

#### Expected Output:

```
attr$=Laird
attr$ now = ABCDEF
```

## 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)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This argument is the string that is written to an attribute.
<b>nData</b>	<b>byVal nData AS INTEGER</b> The three least significant bytes of this integer is saved. The rest is ignored.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ where the new fragment of data is written. If the string attr\$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If

the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

#### Example:

```
//Example :: BleEncode24.sb (See in BT900CodeSnippets.zip)

DIM rc
DIM attr$ : attr$="Laird"

//write 'BCD' to index 1
rc=BleEncode24(attr$,0x444342,1)

//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)

//write 'EF' to index 4
rc=BleEncode16(attr$,0x4645,4)

PRINT "attr$=";attr$
```

#### Expected Output:

```
attr$=ABCDEF
```

## 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)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This argument is the string that is written to an attribute.
<b>nData</b>	<b>byVal nData AS INTEGER</b> The four bytes of this integer is saved. The rest is ignored.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ where the new fragment of data is written. If the string attr\$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

**Example:**

```
//Example :: BleEncode32.sb (See in BT900CodeSnippets.zip)

DIM rc
DIM attr$ : attr$="Laird"

//write 'BCDE' to index 1
rc=BleEncode32(attr$,0x45444342,1)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)

PRINT "attr$=";attr$
```

**Expected Output:**

```
attr$=ABCDE
```

## 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\$, nMantissa, nExponent, nIndex)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.										
<b>Arguments:</b>											
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This argument is the string that is written to an attribute.										
<b>nMantissa</b>	<b>byVal nMantissa AS INTEGER</b> 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. <b>Note:</b> The range is not +/- 2048 because after encoding the following 2 byte values have special meaning: <table border="1"> <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> </table>	0x007FFFFF	NaN (Not a Number)	0x00800000	NRes (Not at this resolution)	0x007FFFFE	+ INFINITY	0x00800002	- INFINITY	0x00800001	Reserved for future use
0x007FFFFF	NaN (Not a Number)										
0x00800000	NRes (Not at this resolution)										
0x007FFFFE	+ INFINITY										
0x00800002	- INFINITY										
0x00800001	Reserved for future use										
<b>nExponent</b>	<b>byVal nExponent AS INTEGER</b> This value must be in the range -128 to 127 or the function fails.										
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b>										

This is the zero based index into the string attr\$ where the new fragment of data is written. If the string attr\$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

#### Example:

```
//Example :: BleEncodeFloat.sb (See in BT900CodeSnippets.zip)

DIM rc
DIM attr$ : attr$=""

//write 1234567 x 10^-54 as FLOAT to index 2
PRINT BleEncodeFLOAT(attr$,123456,-54,0)

//write 1234567 x 10^1000 as FLOAT to index 2 and it will fail
//because the exponent is too large, it has to be < 127
IF BleEncodeFLOAT(attr$,1234567,1000,2) !=0 THEN
  PRINT "\nFailed to encode to FLOAT"
ENDIF

//write 10000000 x 10^0 as FLOAT to index 2 and it will fail
//because the mantissa is too large, it has to be < 8388600
IF BleEncodeFLOAT(attr$,10000000,0,2) !=0 THEN
  PRINT "\nFailed to encode to FLOAT"
ENDIF
```

#### Expected Output:

```
0
Failed to encode to FLOAT
Failed to encode to FLOAT
```

## 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.

## BLENCODESFLOATEX(attr\$,nData, nIndex)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This argument is the string that is written to an attribute
<b>nData</b>	<b>byVal nData AS INTEGER</b> 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.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero-based index into the string attr\$ where the new fragment of data is written. If the string attr\$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

### Example:

```
//Example :: BleEncodeSFloatEx.sb (See in BT900CodeSnippets.zip)

DIM rc, mantissa, exp
DIM attr$ : attr$=""

//write 2,147,483,647 as SFLOAT to index 0
rc=BleEncodeSFloatEX(attr$,2147483647,0)
rc=BleDecodeSFloat(attr$,mantissa,exp,0)
PRINT "\nThe number stored is ";mantissa;" x 10^";exp
```

### Expected Output:

```
The number stored is 214 x 10^7
```

## BleEncodeSFLOAT

### 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.

## BLENCODESFLOAT(attr\$, nMatissa, nExponent, nIndex)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This argument is the string that is written to an attribute.



<b><i>nMantissa</i></b>	<p><b>byVal <i>nMantissa</i> AS INTEGER</b> 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.</p> <p><b>Note:</b> The range is not +/- 2048 because after encoding, the following 2-byte values have special meaning:</p> <table> <tr> <td>0x007FF</td><td>NaN (Not a Number)</td></tr> <tr> <td>0x00800</td><td>NRes (Not at this resolution)</td></tr> <tr> <td>0x007FE</td><td>+ INFINITY</td></tr> <tr> <td>0x00802</td><td>- INFINITY</td></tr> <tr> <td>0x00801</td><td>Reserved for future use</td></tr> </table>	0x007FF	NaN (Not a Number)	0x00800	NRes (Not at this resolution)	0x007FE	+ INFINITY	0x00802	- INFINITY	0x00801	Reserved for future use
0x007FF	NaN (Not a Number)										
0x00800	NRes (Not at this resolution)										
0x007FE	+ INFINITY										
0x00802	- INFINITY										
0x00801	Reserved for future use										
<b><i>nExponent</i></b>	<p><b>byVal <i>nExponent</i> AS INTEGER</b> This value must be in the range -8 to 7 or the function fails.</p>										
<b><i>nIndex</i></b>	<p><b>byVal <i>nIndex</i> AS INTEGER</b> 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.</p>										

**Example:**

```
//Example :: BleEncodeSFloat.sb (See in BT900CodeSnippets.zip)

DIM rc
DIM attr$ : attr$=""

SUB Encode(BYVAL mantissa, BYVAL exp)
  IF BleEncodeSFloat(attr$,mantissa,exp,2) !=0 THEN
    PRINT "\nFailed to encode to SFLOAT"
  ELSE
    PRINT "\nSuccess"
  ENDIF
ENDSUB

Encode(1234,-4)    //1234 x 10^-4
Encode(1234,10)    //1234 x 10^10 will fail because exponent too large
Encode(10000,0)    //10000 x 10^0 will fail because mantissa too large
```

**Expected Output:**

```
Success
Failed to encode to SFLOAT
Failed to encode to SFLOAT
```

## BleEncodeTIMESTAMP

### FUNCTION

This function overwrites a 7-byte string into the string at a specified offset. If the string is not long enough, it is extended with the new extended block uninitialized and then the byte specified is overwritten.

The 7-byte string consists of a byte each for century, year, month, day, hour, minute and second. If (year \* month) is zero, it is taken as “not noted” year and all the other fields are set zero (not noted).

For example, 5 May 2013 10:31:24 is represented as \14\0D\05\05\0A\1F\18.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

**Note:** When the attr\$ string variable is updated, the two byte year field is converted into a 16-bit integer. Hence \14\0D gets converted to \DD\07

### BLEENCODETIMESTAMP (attr\$, timestamp\$, nIndex)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This argument is the string that is written to an attribute.
<b>timestamp\$</b>	<b>byRef timestamp\$ AS STRING</b> 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.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ where the new fragment of data is written. If the string attr\$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

### Example:

```
//Example :: BleEncodeTimestamp.sb (See in BT900CodeSnippets.zip)

DIM rc, ts$
DIM attr$ : attr$=""

//write the timestamp <5 May 2013 10:31:24>
ts$="\14\0D\05\05\0A\1F\18"

PRINT BleEncodeTimestamp(attr$,ts$,0)
```

### Expected Output:

0

## 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*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>attr\$</i></b>	<b>byRef <i>attr\$</i> AS STRING</b> This argument is the string is written to an attribute
<b><i>nIndex1</i></b>	<b>byVal <i>nIndex1</i> AS INTEGER</b> This is the zero based index into the string <i>attr\$</i> where the new fragment of data is written. If the string <i>attr\$</i> 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 <code>SYSINFO(2013)</code> ), this function fails.
<b><i>str\$</i></b>	<b>byRef <i>str\$</i> AS STRING</b> This contains the source data which is qualified by the <i>nIndex2</i> and <i>nLen</i> arguments that follow.
<b><i>nIndex2</i></b>	<b>byVal <i>nIndex2</i> AS INTEGER</b> This is the zero based index into the string <i>str\$</i> from which data is copied. No data is copied if this is negative or greater than the string.
<b><i>nLen</i></b>	<b>byVal <i>nLen</i> AS INTEGER</b> This species the number of bytes from offset <i>nIndex2</i> to be copied into the destination string. It is clipped to the number of bytes left to copy after the index.

### Example:

```
//Example :: BleEncodeString.sb (See in BT900CodeSnippets.zip)
DIM rc, attr$, ts$ : ts$="Hello World"
//write "Wor" from "Hello World" to the attribute at index 2
rc=BleEncodeString(attr$,2,ts$,6,3)
PRINT attr$
```

### Expected Output:

```
\00\00Wor
```

## 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*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>attr\$</i></b>	<b>byRef <i>attr\$</i> AS STRING</b> This is the string written to an attribute. It is treated as a bit array.
<b><i>nDstIdx</i></b>	<b>byVal <i>nDstIdx</i> AS INTEGER</b> This is the zero based bit index into the string <i>attr\$</i> , treated as a bit array, where the new fragment of data bits is written. If the string <i>attr\$</i> 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 <code>SYSINFO(2013)</code> ), this function fails.
<b><i>srcBitArr</i></b>	<b>byVal <i>srcBitArr</i> AS INTEGER</b> This contains the source data bits which is qualified by the <i>nSrcIdx</i> and <i>nBitLen</i> arguments that follow.
<b><i>nSrcIdx</i></b>	<b>byVal <i>nSrcIdx</i> AS INTEGER</b> This is the zero-based bit index into the bit array contained in <i>srcBitArr</i> from where the data bits is copied. No data is copied if this index is negative or greater than 32.
<b><i>nBitLen</i></b>	<b>byVal <i>nBitLen</i> AS INTEGER</b> This species the number of bits from offset <i>nSrcIdx</i> to be copied into the destination bit array represented by the string <i>attr\$</i> . It is clipped to the number of bits left to copy after the index <i>nSrcIdx</i> .

#### Example:

```
//Example :: BleEncodeBits.sb (See in BT900CodeSnippets.zip)
DIM attr$, rc, bA: bA=b'1110100001111
rc=BleEncodeBits(attr$,20,bA,7,5) : PRINT attr$ //copy 5 bits from index 7 to attr$
```

#### Expected Output:

```
\00\00\A0\01
```

## Attribute Decoding Functions

Data in a characteristic is stored in a value attribute, a byte array. Multibyte characteristic descriptors content is stored similarly. Those bytes are manipulated in *smart*BASIC applications using STRING variables.

Attribute data is stored in little endian format.

This section describes decoding functions that allow attribute strings to be read from smaller bitwise subfields more efficiently than the generic STRXXXX functions that are made available in *smart*BASIC.

**Note:** CCCD and SCCD descriptors are special cases as they are defined as having two bytes which are treated as 16-bit integers mapped to INTEGER variables in *smart*BASIC.

## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>nData</b>	<b>byRef nData AS INTEGER</b> This references an integer to be updated with the 8-bit data from attr\$, after sign extension.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which the data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

#### Example:

```
//Example :: BleDecodeS8.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

//create random service just for this example
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

//create char and commit as part of service committed above
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read signed byte from index 2
rc=BleDecodeS8(attr$,v1,2)

PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

```
//read signed byte from index 6 - two's complement of -122
rc=BleDecodeS8(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

#### Expected Output:

```
data in Hex = 0x00000002
data in Decimal = 2

data in Hex = 0xFFFFF86
data in Decimal = -122
```

## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>nData</b>	<b>byRef nData AS INTEGER</b> This references an integer to be updated with the 8-bit data from attr\$, without sign extension.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

#### Example:

```
//Example :: BleDecodeU8.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
```

```
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read unsigned byte from index 2
rc=BleDecodeU8(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read unsigned byte from index 6
rc=BleDecodeU8(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

#### Expected Output:

```
data in Hex = 0x00000002
data in Decimal = 2

data in Hex = 0x00000086
data in Decimal = 134
```

## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>nData</b>	<b>byRef nData AS INTEGER</b> This references an integer to be updated with the 2-byte data from attr\$, after sign extension.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

**Example:**

```
//Example :: BleDecodeS16.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 2 signed bytes from index 2
rc=BleDecodeS16(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read 2 signed bytes from index 6
rc=BleDecodeS16(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

**Expected Output:**

```
data in Hex = 0x00000302
data in Decimal = 770

data in Hex = 0xFFFF8786
data in Decimal = -30842
```



## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>nData</b>	<b>byRef nData AS INTEGER</b> This references an integer to be updated with the 2-byte data from attr\$, without sign extension.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

#### Example:

```
//Example :: BleDecodeU16.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 2 unsigned bytes from index 2
rc=BleDecodeU16(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read 2 unsigned bytes from index 6
rc=BleDecodeU16(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
```

```
PRINT "\ndata in Decimal = "; v1;"\n"
```

#### Expected Output:

```
data in Hex = 0x00000302
data in Decimal = 770

data in Hex = 0x00008786
data in Decimal = 34694
```

## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>nData</b>	<b>byRef nData AS INTEGER</b> This references an integer to be updated with the 3-byte data from attr\$, with sign extension.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

#### Example:

```
//Example :: BleDecodeS24.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)
```

```
rc=BleCharValueRead(chrHandle,attr$)

//read 3 signed bytes from index 2
rc=BleDecodeS24(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read 3 signed bytes from index 6
rc=BleDecodeS24(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

### Expected Output:

```
data in Hex = 0x00040302
data in Decimal = 262914

data in Hex = 0xFF888786
data in Decimal = -7829626
```

## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>nData</b>	<b>byRef nData AS INTEGER</b> This references an integer to be updated with the 3-byte data from attr\$, without sign extension.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

**Example:**

```
//Example :: BleDecodeU24.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 3 unsigned bytes from index 2
rc=BleDecodeU24(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read 3 unsigned bytes from index 6
rc=BleDecodeU24(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

**Expected Output:**

```
data in Hex = 0x00040302
data in Decimal = 262914

data in Hex = 0x00888786
data in Decimal = 8947590
```

## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>nData</b>	<b>byRef nData AS INTEGER</b> This references an integer to be updated with the 3-byte data from attr\$, after sign extension.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

#### Example:

```
//Example :: BleDecode32.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 4 signed bytes from index 2
rc=BleDecode32(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read 4 signed bytes from index 6
rc=BleDecode32(attr$,v1,6)
```

```
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

### Expected Output:

```
data in Hex = 0x85040302
data in Decimal = -2063334654

data in Hex = 0x89888786
data in Decimal = -1987541114
```

## 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*)

<b>Returns</b>	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the <i>nIndex</i> parameter is positioned towards the end of the string.										
<b>Arguments:</b>											
<b><i>attr\$</i></b>	<b>byRef <i>attr\$</i> AS STRING</b> This references the attribute string from which the function reads.										
<b><i>nMantissa</i></b>	<b>byRef <i>nMantissa</i> AS INTEGER</b> This is updated with the 24 bit mantissa from the 4-byte object. If <i>nExponent</i> is 0, you must check for the following special values: <table border="1"> <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> </table>	0x007FFFFF	NaN (Not a Number)	0x00800000	NRes (Not at this resolution)	0x007FFFFE	+ INFINITY	0x00800002	- INFINITY	0x00800001	Reserved for future use
0x007FFFFF	NaN (Not a Number)										
0x00800000	NRes (Not at this resolution)										
0x007FFFFE	+ INFINITY										
0x00800002	- INFINITY										
0x00800001	Reserved for future use										
<b><i>nExponent</i></b>	<b>byRef <i>nExponent</i> AS INTEGER</b> This is updated with the 8-bit mantissa. If it is zero, check <i>nMantissa</i> for special cases as stated above.										
<b><i>nIndex</i></b>	<b>byVal <i>nIndex</i> AS INTEGER</b> This is the zero based index into the string <i>attr\$</i> from which data is read. If the string <i>attr\$</i> is not long enough to accommodate the index plus the number of bytes to read, this function fails.										

### Example:

```
//Example :: BleDecodeFloat.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, mantissa, exp
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
```

```
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 4 bytes FLOAT from index 2 in the string
rc=BleDecodeFloat(attr$,mantissa,exp,2)
PRINT "\nThe number read is ";mantissa;" x 10^";exp

//read 4 bytes FLOAT from index 6 in the string
rc=BleDecodeFloat(attr$,mantissa,exp,6)
PRINT "\nThe number read is ";mantissa;"x 10^";exp
```

#### Expected Output:

```
The number read is 262914*10^-123
The number read is -7829626*10^-119
```

## BleDecodeSFloat

### FUNCTION

This function reads two bytes in a string at a specified offset into a couple of 32-bit integer variables. The decoding results in two variables, the 12-bit signed mantissa and the 4-bit signed exponent. If the offset points beyond the end of the string then this function fails.

#### BLEDECODESFloat (attr\$, nMantissa, nExponent, nIndex)

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\$	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.		
nMantissa	<b>byRef nMantissa AS INTEGER</b> 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:		
	0x007FFFFF	NaN (Not a Number)	
	0x00800000	NRes (Not at this resolution)	
	0x007FFFFE	+ INFINITY	

	0x00800002	- INFINITY
	0x00800001	Reserved for future use
<b><i>nExponent</i></b>	<b>byRef nExponent AS INTEGER</b> This is updated with the 4-bit mantissa. If it is zero, check the nMantissa for special cases as stated above.	
<b><i>nIndex</i></b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.	

**Example:**

```
//Example :: BleDecodeSFloat.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, mantissa, exp
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 2 bytes FLOAT from index 2 in the string
rc=BleDecodeSFloat(attr$,mantissa,exp,2)
PRINT "\nThe number read is ";mantissa;" x 10^";exp

//read 2 bytes FLOAT from index 6 in the string
rc=BleDecodeSFloat(attr$,mantissa,exp,6)
PRINT "\nThe number read is ";mantissa;"x 10^";exp
```

**Expected Output:**

```
The number read is 770 x 10^0
The number read is 1926x 10^-8
```



## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>timestamp\$</b>	<b>byRef timestamp\$ AS STRING</b> 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
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into the string attr\$ from which data is read. If the string attr\$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

### Example:

```
//Example :: BleDecodeTimestamp.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, ts$
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//5th May 2013, 10:31:24
DIM attr$ : attr$="\00\01\02\DD\07\05\05\0A\1F\18"
DIM uuid : uuid = 0x1853

rc=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
```

## 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)

<b>Returns</b>	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.
<b>Arguments:</b>	
<b>attr\$</b>	<b>byRef attr\$ AS STRING</b> This references the attribute string from which the function reads.
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is the zero based index into string attr\$ from which data is read.
<b>dst\$</b>	<b>byRef dst\$ AS STRING</b> This argument is a reference to a string that is updated with up to nMaxBytes of data from the index specified. A shorter string is returned if there are not enough bytes beyond the index.
<b>nMaxBytes</b>	<b>byVal nMaxBytes AS INTEGER</b> This specifies the maximum number of bytes to read from attr\$.

### Example:

```
//Example :: BleDecodeString.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, ts$,decStr$
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
/"ABCDEFGHJIJ"
DIM attr$ : attr$="41\42\43\44\45\46\47\48\49\4A"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)
```

```
//read max 4 bytes from index 3 in the string
rc=BleDecodeSTRING(attr$,3,decStr$,4)
PRINT "\nd$=";decStr$

//read max 20 bytes from index 3 in the string - will be truncated
rc=BleDecodeSTRING(attr$,3,decStr$,20)
PRINT "\nd$=";decStr$

//read max 4 bytes from index 14 in the string - nothing at index 14
rc=BleDecodeSTRING(attr$,14,decStr$,4)
PRINT "\nd$=";decStr$
```

#### Expected Output:

```
d$=CDEF
d$=CDEFGHIJ
d$=
```

## 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*)

<b>Returns</b>	INTEGER, the number of bits extracted from the attribute string. Can be less than the size expected if the <i>nSrcIdx</i> parameter is positioned towards the end of the source string or if <i>nDstIdx</i> will not allow more to be copied.
<b>Arguments:</b>	
<b><i>attr\$</i></b>	<b>byRef <i>attr\$</i> AS STRING</b> 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.
<b><i>nSrcIdx</i></b>	<b>byVal <i>nSrcIdx</i> AS INTEGER</b> This is the zero based bit index into the string <i>attr\$</i> from which data is read. For example, the third bit in the second byte is index number 10.
<b><i>dstBitArr</i></b>	<b>byRef <i>dstBitArr</i> AS INTEGER</b> This argument references an integer treated as an array of 32 bits into which data is copied. Only the written bits are modified.
<b><i>nDstIdx</i></b>	<b>byVal <i>nDstIdx</i> AS INTEGER</b> This is the zero based bit index into the bit array <i>dstBitArr</i> to where the data is written.

***nMaxBits***

**byVal *nMaxBits* AS INTEGER**

This argument specifies the maximum number of bits to read from attr\$. Due to the destination being an integer variable, it cannot be greater than 32. Negative values are treated as zero.

**Example:**

```
//Example :: BleDecodeBits.sb (See in BT900CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, ts$,decStr$
DIM ba : ba=0
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//"ABCDEFGHJIJ"
DIM attr$ : attr$="41\42\43\44\45\46\47\48\49\4A"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read max 14 bits from index 20 in the string to index 10
rc=BleDecodeBITS(attr$,20,ba,10,14)
PRINT "\nbit array = ", INTEGER.B' ba

//read max 14 bits from index 20 in the string to index 10
ba=0x12345678
PRINT "\n\nbit array = ",INTEGER.B' ba

rc=BleDecodeBITS(attr$,14000,ba,0,14)
PRINT "\nbit array now = ", INTEGER.B' ba
//ba will not have been modified because index 14000
//doesn't exist in attr$
```

## Expected Output:

```
bit array =      00000000000010000110100000000000

bit array =      00010010001101000101011001111000
bit array now = 00010010001101000101011001111000
```

## Pairing, Bonding, and Security Manager Functions

### Pairing and 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.

#### BLEBONDINGSTATS (nRolling, nPersistent)

<b>Returns</b>	The total capacity of the database
<b>Arguments:</b>	
<b>nRolling</b>	<b>byREF nRolling AS INTEGER</b> On return, this integer contains the total number of bonds in the rolling database.
<b>nPersistent</b>	<b>byREF nPersistent AS INTEGER</b> On return, this integer contains the total number of bonds in the persistent database.

#### Example:

```
dim rc, nRoll, nPers
print "\nBonding 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.

### *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\$)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>bdAddr\$</b>	<b>byREF bdAddr\$ AS STRING</b> Bluetooth address in big endian. Must be exactly seven bytes long.

#### Example:

```
dim rc, i, j, k, adr$, inf

'//Loop through the bonding manager. Make all entries persistent
for i=0 to BleBondingStats(j,k)
    rc=BleBondMgrGetInfo(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.

## 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)

<b>Returns</b>	INTEGER: Is 0 if not trusted, otherwise it is the length of the long term key (LTK)												
<b>Arguments</b>													
<b>addr\$</b>	<b>byRef addr\$ AS STRING</b> This is the address of the device for which the bonding information is to be checked.												
<b>fAsCentral</b>	Set to 0 if the device is to be trusted as a peripheral and non-zero if to be trusted as central.												
<b>keyInfo</b>	This is a bit mask with bit meanings as follows: This specifies the write rights and shall have one of the following values: <table border="1"> <tr> <td>Bit 0</td><td>Set if MITM is authenticated</td></tr> <tr> <td>Bit 1</td><td>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>Bit 2</td><td>Set if an IRK (identity resolving key) exists</td></tr> <tr> <td>Bit 3</td><td>Set if a CSRK (connection signing resolving key) exists</td></tr> <tr> <td>Bit 4</td><td>Set if LTK as slave exists</td></tr> <tr> <td>Bit 5</td><td>Set if LTK as master exists</td></tr> </table>	Bit 0	Set if MITM is authenticated	Bit 1	Set if it is a rolling bond and can be automatically deleted if the database is full and a new bonding occurs	Bit 2	Set if an IRK (identity resolving key) exists	Bit 3	Set if a CSRK (connection signing resolving key) exists	Bit 4	Set if LTK as slave exists	Bit 5	Set if LTK as master exists
Bit 0	Set if MITM is authenticated												
Bit 1	Set if it is a rolling bond and can be automatically deleted if the database is full and a new bonding occurs												
Bit 2	Set if an IRK (identity resolving key) exists												
Bit 3	Set if a CSRK (connection signing resolving key) exists												
Bit 4	Set if LTK as slave exists												
Bit 5	Set if LTK as master exists												
<b>rollingAge</b>	If the value is <= 0, this is not a rolling device. 1 implies it is the newest bond, 2 implies it is the second newest bond, and so on.												
<b>rollingCount</b>	On exit this will contain the total number of rolling bonds. This provides some context with regards to how old this device is compared to other bonds in the rolling group.												

#### Example:

```
//Example
DIM rc, addr$
addr$="000016A4123456"
rc = BleBondingPersistKey(addr$)
```

## BleBondingEraseKey

### FUNCTION

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

#### BLEBONDINGERASEKEY (bdAddr\$)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>bdAddr\$</b>	<b>byREF bdAddr\$ AS STRING</b> Bluetooth address in big endian. Must be exactly seven bytes long.

#### Example:

```
dim rc, i, adr$, inf

//delete link key at index 0
```

```
rc=BleBondMngrGetInfo(0,adr$,inf)    //get the BT address
rc=BleBondingEraseKey(adr$)
if rc==0 then
    print "\nLink key for device ";StrHexize$(adr$);" erased"
else
    print "\nError erasing link key ";integer.h'rc
endif
```

**Expected Output:**

```
Link key for device 01FA84D748D903 erased
```

BLEBONDINGERASEKEY is a built-in function.

*BleBondingEraseAll*

**FUNCTION**

This function is used to erase all bondings in the database.

**BLEBONDINGERASEALL ()**

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

**Example:**

```
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.

### BLEBONDMNGRGETINFO (nIndex, addr\$, nExtraInfo)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nIndex</b>	<b>byVal nIndex AS INTEGER</b> This is an index in the range 0 to 1, less than the value returned by SYSINFO(2012).
<b>addr\$</b>	<b>byRef addr\$ AS STRING</b> 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.
<b>nExtraInfo</b>	<b>byRef nExtraInfo AS INTEGER</b> On exit, if nIndex points to a valid entry in the database, this variable contains a composite integer value where the lower 16 bits are for internal use and should be treated as opaque data. Bit 17 is set if the IRK (Identity Resolving Key) exists for the trusted device and bit 18 is set if the CSRK (Connection Signing Resolving Key) exists for the trusted device.

### Example:

```
//Example :: BleBondMngrGetInfo.sb (See in BT900CodeSnippets.zip)
#define BLE_INV_INDEX      24619
DIM rc, addr$, exInfo

rc = BleBondMngrGetInfo(0,addr$,exInfo) //Extract info of device at index 1

IF rc==0 THEN
    PRINT "\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

## 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:

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

To submit a passkey, use the function [BLESECMNGRPASKEY](#).

### *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.”

#### [BLESECMNGRJUSTWORKSCONF\(nJustWorksConf\)](#)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<i>nJustWorksConf</i>	<b>byVal nJustWorksConf AS INTEGER.</b> If set to 0, pairing <i>just works</i> 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 <a href="#">BleAcceptPairing()</a> .

See example for [BlePair\(\)](#).

### *BleSecMngrOobPref*

#### FUNCTION

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

#### [BLESECMNGROOBPREF\(nOobPreferred\)](#)

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

<b><i>nJustWorksConf</i></b>	<b>byVal nJustWorksConf AS INTEGER.</b> If set to 0, there will be no OOB data available. If set to 1, OOB data is available. If set to 2, prompt for OOB data availability.
------------------------------	---

**Example:**

```
//Example :: BleSecMngrOobPref.sb (See in BT900CodeSnippets.zip)

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.
```

### *BleSecMngrOobAvailable*

**FUNCTION**

This function is used indicate that OOB data is available for the requested connection.

#### **BLESECMNGROOBAVAILABLE(connHandle, nOobAvail)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>connHandle</i></b>	<b>byVal connHandle AS INTEGER.</b> The connection handle as received via the EVBLEMSG event with msgId set to 0.
<b><i>nOobAvail</i></b>	<b>byVal nOobAvail AS INTEGER.</b> If set to 0, we do not have OOB data available. If set to 1, OOB data is available.

### *BleAcceptPairing*

**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 error (invalid state) is returned.

#### **BLEACCEPTPAIRING(nConnHandle, nAccept)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nConnHandle</i></b>	<b>byVal nConnHandle AS INTEGER.</b> The handle of the connection for which you are accepting or rejecting a pairing request.

<b><i>nAccept</i></b>	<b>byVal nAccept AS INTEGER.</b> Set to 0 to reject the pairing request, set to 1 to accept the pairing request.
-----------------------	---

See example for `BlePair()`.

### *BleSecMngrPasskey*

#### FUNCTION

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

#### *BLESECMNGRPASSKEY(connHandle, nPassKey)*

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>connHandle</i></b>	<b>byVal connHandle AS INTEGER.</b> The connection handle as received via the EVBLEMSG event with msgId set to 0.
<b><i>nPassKey</i></b>	<b>byVal nPassKey AS INTEGER.</b> The passkey to submit to the stack. Submit a value outside the range 0 to 999999 to reject the pairing.

#### Example:

```
//Example :: BleSecMngrPasskey.sb (See in BT900CodeSnippets.zip)

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

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

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

ONEVENT EVBLEMSG CALL HandlerBleMsg

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

WAITEVENT
```

### Expected Output:

```
Adverts Started

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

## *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](#).

### *BLESECMNGRPASSKEY(connHandle, nPassKey)*

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

### Example:

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

#define OOB_KEY 2

FUNCTION HandlerBleMsg (BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
    SELECT nMsgId
        CASE 0
            connHandle = nCtx
            PRINT "\nBle Connection ",nCtx
        CASE 1
            PRINT "\nDisconnected ";nCtx;"\n"
            EXITFUNC 0
        CASE 10
            PRINT "\n--- New bond"
```

```
CASE 11
    PRINT "\n +++ Auth Key Request, type=",nCtx
    if nCtx == OOB_KEY then
        rc=BleSecMngrOobKey(connHandle,oob$)
        print "\nOOB Key ";StrHexize$(oob$);" was used"
    endif

CASE ELSE
    PRINT "\nUnknown Ble Msg"
ENDSELECT
ENDFUNC 1

ONEVENT EVBLEMSG CALL HandlerBleMsg

IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "\nAdverts Started\n"
    PRINT "\nMake a connection to the BT900"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

### Expected Output:

```
Adverts Started

Make a connection to the BT900
Ble Connection,      1655
  +++ Auth Key Request, type=2
OOB Key 11223344556677889911AACCBBDDEEFF was used
--- New bond
Disconnected 1655
Passkey: 904096
--- New bond
--- Disconnected 2782
```

## BleSecMngrKeySizes

### FUNCTION

This function sets minimum and maximum long term encryption key size requirements for subsequent pairings. If this function is not called, default values are 7 and 16 respectively. To ship your end product to a country with an export restriction, reduce nMaxKeySize to an appropriate value and ensure it is not modifiable.

#### BLESECMNGRKEYSIZES(nMinKeysize, nMaxKeysize)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nMinKeysiz</b>	<b>byVal nMinKeysiz AS INTEGER.</b> The minimum key size. The range of this value is from 7 to 16.
<b>nMaxKeysize</b>	<b>byVal nMaxKeysize AS INTEGER.</b> The maximum key size. The range of this value is from nMinKeysize to 16.

#### Example:

```
//Example :: BleSecMngrKeySizes.sb (See in BT900CodeSnippets.zip)
PRINT BleSecMngrKeySizes(8,15)
```

#### Expected Output:

0

## BleSecMngrIloCap

### 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:

[https://www.Bluetooth.org/docman/handlers/DownloadDoc.ashx?doc\\_id=86174](https://www.Bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=86174)

[https://www.Bluetooth.org/docman/handlers/DownloadDoc.ashx?doc\\_id=86173](https://www.Bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=86173)

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

#### BLESECMNGRIOCAP (nIloCap)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nIloCap</b>	<b>byVal nIloCap AS INTEGER.</b> The user I/O capability for all subsequent pairings.
	0 None; also known as <i>Just Works</i> (unauthenticated pairing)
	1 Display with Yes/No input capability (authenticated pairing)
	2 Keyboard Only (authenticated pairing)



	3	Display Only (authenticated pairing – if other end has input cap)
	4	Keyboard and Display (authenticated pairing)

**Example:**

```
//Example :: BleSecMngrIoCap.sb (See in BT900CodeSnippets.zip)
PRINT BleSecMngrIoCap(1)
```

**Expected Output:**

```
0
```

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

### *BleSecMngrBondReq*

**FUNCTION**

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

#### **BLESECMNGRBONDREQ (nBondReq)**

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nBondReq</b>	<b>byVal nBondReq AS INTEGER.</b> 0 – Disable 1 – Enable

**Example:**

```
//Example :: BleSecMngrBondReq.sb (See in BT900CodeSnippets.zip)
IF BleSecMngrBondReq(0)==0 THEN
    PRINT "\nBonding disabled"
ENDIF
```

**Expected Output:**

```
Bonding disabled
```

### *BlePair*

**FUNCTION**

This routine is used to induce the module to pair with the peer and to specify whether to bond with the peer by storing pairing information in the bonding manager. This function is likely to be used if a write attempt to an attribute fails with a status code such as 0x105. See [EvAttrWrite](#) and [EvAttrRead](#).

## BLEPAIR (hConn, nSave)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.	
Arguments:		
hConn	<b>byRef hConn AS INTEGER.</b> This is the connection handle provided in the EVBLEMSG(0) message which informs the stack that a connection had been established.	
nSave	<b>byVal nSave AS INTEGER</b> This flag sets whether or not to bond.	
	Value	Description
	0	Do not store pairing information (don't bond)
	1	Store pairing information (bond)

### Example:

```

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 BL600 running the health thermometer sensor sample app
//which requires bonding.
//It connects, tries to read from the temperature characteristic and then initiates a bonding
//procedure when it fails.

#define GATT_SERVER_ADDRESS      "\01\F6\36\27\A6\0B\EA"
#define AUTHENTICATION_REQUIRED  0x0105

#define SERVICE_UUID             0x1809
#define CHAR_UUID                 0x2a1c
#define DESC_UUID                 0x2902

'//-----
'// For debugging
'// --- rc = result code
'// --- ln = line number
'//-----

Sub AssertRC(rc,ln)
    if rc!=0 then
        print "\nFail :";integer.h' rc;" at tag ";ln
    endif
EndSub

'//-----
'// This handler is called when there is a significant BLE event
'//-----

```

```
function HndlrBleMsg (byval nMsgId as integer, byval nCtx as integer)
    select nMsgId
        case 0
            hC = nCtx
            print "\nConnected, Finding Temp Measurement Char"
            rc=BleGattcFindDesc(nCtx, BleHandleUuid16(SERVICE_UUID), 0, BleHandleUuid16(CHAR_UUID),
0, BleHandleUuid16(DESC_UUID), 0)
            AssertRC(rc,35)
        case 1
            print "\n\n --- Disconnected"
        case 10
            print "\nNew bond created"
            print "\n\nAttempting to enable indications again"
            rc=BleGattcWrite(hC, hDesc, s$)
            AssertRC(rc,58)
        case 11
            print "\nPair request: Accepting"
            rc=BleAcceptPairing(hC,1)
            AssertRC(rc,52)
            print "\nPairing in progress"
        case 17
            print "\nNew pairing/bond has replaced old key"
        case 18
            print "\nConnection now encrypted"
        case else
    endselect
endfunc 1

'//-----
'// Called after BleGattcFindDesc returns success
'//-----

function HndlrFindDesc(hConn, hD)
    if hD==0 then
        print "\nCCCD not found"
        exitfunc 0
    endif

    hDesc = hD
    print "\nTemp Measurement Char CCCD Found. Attempting to enable indications"
```

```
rc=BleGattcWrite(hConn, hDesc, s$)
AssertRC(rc,58)
endfunc 1

'//-----
'// Called after BleGattcRead returns success
'//-----

function HndlrAttrWriteExit(hConn, hAttr, nSts)
endfunc 0

'//-----
'// Called after BleGattcRead returns success
'//-----

function HndlrAttrWrite(hConn, hAttr, nSts)
    if nSts == 0 then
        print "\nIndications enabled"
        print "\nDisabling indications"
        s$ = "\00\00"
        rc=BleGattcWrite(hC, hDesc, s$)
        onevent evattrwrite call HndlrAttrWriteExit
        exitfunc 1

    elseif nSts == AUTHENTICATION_REQUIRED then
        print "\n\nAuthentication required."
        '//bond with the peer

        rc=BlePair(hConn, 1)
        AssertRC(rc,75)
        print " Bonding..."
    endif
endfunc 1

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

rc=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, 30000000)
AssertRC(rc,91)

//-----
// Enable synchronous event handlers
//-----

onevent evblemsg call HndlrBleMsg
onevent evfinddesc call HndlrFindDesc
onevent evattrwrite call HndlrAttrWrite

waitevent

print "\nExiting..."
```

#### Expected Output:

```
Connected, Finding Temp Measurement Char
Temp Measurement Char CCCD Found. Attempting to enable indications

Authentication required. Bonding...
Pair request: Accepting
Pairing in progress
Connection now encrypted
New bond created

Attempting to enable indications again
Indications enabled
Disabling indications
Exiting...
```

### *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).

#### **BLEENCRYPTCONNECTION(nConnHandle, nLtkMinSize, nMitmRequired)**

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

<b><i>nConnHandle</i></b>	<b>byVal nConnHandle AS INTEGER.</b> The handle of the connection which is obtained from an EVBLEMSG message with ID 0 indicating that a connection had been established.
<b><i>nLtkMinSize</i></b>	<b>byVal nLtkMinSize AS INTEGER.</b> The minimum long term key size which must be in the range 7-16.
<b><i>nMitmRequired</i></b>	<b>byVal nMitmRequired AS INTEGER.</b> Set to 1 if MITM protection is required, 0 if not required.

**Example:**

```

dim rc, pr$, hC, hDesc
#define GATT_SERVER_ADDRESS      "\01\F6\36\27\A6\0B\EA"

//This example app was tested with a BL600 running the health thermometer sensor sample app
//which the module had previously bonded with.

'//-----
'// For debugging
'// --- rc = result code
'// --- ln = line number
'//-----

Sub AssertRC(rc,ln)
    if rc!=0 then
        print "\nFail :";integer.h' rc;" at tag ";ln
    endif
EndSub

'//-----
'// This handler is called when there is a significant BLE event
'//-----

function HndlrBleMsg(byval nMsgId as integer, byval nCtx as integer)
    select nMsgId
    case 0
        hC = nCtx
        print "\nConnected"
        rc=BleEncryptConnection(hC, 16, 0)
        if rc==0 then
            print "\nEncrypting connection"
        else
            AssertRC(rc,28)
        end if
    end select
end function

```

```
endif
case 1
    print "\n\n --- Disconnected"
    exitfunc 0
case 10
    print "\nNew bond created"

case 11
    print "\nPair request: Accepting"
    rc=BleAcceptPairing(hC,1)
    AssertRC(rc,52)
    print "\nPairing in progress"
case 17
    print "\nNew pairing/bond has replaced old key"
case 18
    print "\nConnection now encrypted"
    rc=BleDisconnect(hC)
case else
endselect
endfunc 1

rc=BleSecMngrIoCap(0)          //set io capability to just works
rc=BleSecMngrJustWorksConf(0) //module will not wait for confirmation (EVBLEMSG 11) before
just works pairing

pr$ = GATT_SERVER_ADDRESS
rc=BleConnect(pr$, 10000, 25, 100, 30000000)
AssertRC(rc,91)

onevent evblemsg call HndlrBleMsg

waitevent

print "\nExiting..."
```

### Expected Output:

```
Connected
Encrypting connection
Connection now encrypted

--- Disconnected
Exiting...
```

## HID REPORT PARSING

The BT900 module contains a number of smart basic functions that assist in the creation and parsing of HID reports.

Reports can be created to a specified size to which variables can then be assigned, specifying lengths and offsets to build up the report. The report can then be passed to the BTC HID functions to be transmitted over a HID connection.

The HID report parsing functions can also be used to extract variables from a report that has been received over a HID connection or extract the whole report as a string (for example, for sending over the UART).

### HIDReportInit

#### FUNCTION

This function creates a HID report of the specified size and initialises the contents to zero.

#### HIDReportInit (numBitsn, nHandle)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>numBits</b>	<b>byVAL numBits as INTEGER</b> Length of the report to create in bits.
<b>nHandle</b>	<b>byREF nHandle as INTEGER</b> Returns a handle to the newly created HID report.

#### Example:

```
dim rc, nHandle

// Initialise a report with a length of 40 bits (5 bytes)
rc = HIDReportInit(40, nHandle)
if rc==0 then
    print "\nHID Report Created. Handle: "; nHandle
endif
```

#### Expected Output when report is successfully initialised:

```
HID Report Created. Handle: 130060
```



## HIDReportAppendInt

### FUNCTION

This function inserts an integer into the HID report object at offset bitIndex and of length bitLen.

#### HIDReportAppendInt (nHandle, bitIndex, bitLen, nVal)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVAL nHandle as INTEGER</b> Handle of the report to be written to.
<b>bitIndex</b>	<b>byVAL bitIndex as INTEGER</b> Location in the report to write the integer to, defined in bits.
<b>bitLen</b>	<b>byVAL bitLen as INTEGER</b> Length of the integer to be written in bits.
<b>nVal</b>	<b>byVAL nVal as INTEGER</b> Value to write to the report. Should not exceed the length defined in bitLen.

#### Example:

```
dim rc, nHandle

// Initialise a report with a length of 40 bits (5 bytes)
rc = HIDReportInit(40, nHandle)
if rc==0 then
    print "\nHID Report Created. Handle: "; nHandle
endif

// Write a 16 bit integer at an offset of 8 bits in the report
rc = HIDReportAppendInt(nHandle, 8, 16, 65535)
if rc==0 then
    print "\nSuccessfully appended integer"
endif
```

#### Expected Output when the integer is successfully appended to the report:

```
HID Report Created. Handle: 130060
Successfully appended integer
```

## HIDReportAppendStr

### FUNCTION

This function inserts a string into the HID report object at offset bitIndex and of length bitLen.

#### HIDReportAppendString (nHandle, bitIndex, bitLen, sVal)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVAL nHandle as INTEGER</b> Handle of the report to be written to.
<b>bitIndex</b>	<b>byVAL bitIndex as INTEGER</b> Location in the report to write the string to, defined in bits.
<b>bitLen</b>	<b>byVAL bitLen as INTEGER</b> Length of the string to be written in bits.
<b>sVal</b>	<b>byVAL sVal as STRING</b> String to write to the report. Should not exceed the length defined in bitLen.

#### Example:

```
dim rc, nHandle
// String to write into the report
dim str$ : str$ = "abc"

// Initialise a report with a length of 40 bits (5 bytes)
rc = HIDReportInit(40, nHandle)
if rc==0 then
    print "\nHID Report Created. Handle: "; nHandle
endif

// Write a 24 bit string at an offset of 8 bits in the report
rc = HIDReportAppendStr(nHandle, 8, 24, str$)
if rc==0 then
    print "\nSuccessfully appended string"
endif
```

#### Expected Output when the string is successfully appended to the report:

```
HID Report Created. Handle: 130060
Successfully appended string
```

## HIDReportImport

### FUNCTION

This function imports a string into a HID report object to overwrite the entire report.

#### HIDReportImport (nHandle, bitLen, sReport\$)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVAL nHandle as INTEGER</b> Handle of the report to be written to.
<b>bitLen</b>	<b>byVAL bitLen as INTEGER</b> Length of the string to be written in bits.
<b>sReport</b>	<b>byVAL sReport as STRING</b> String to write to the report. Should not exceed the length defined in bitLen.

#### Example:

```
dim rc, nHandle
// String to write into the report
dim str$ : str$ = "abc"

// Initialise a report with a length of 40 bits (5 bytes)
rc = HIDReportInit(40, nHandle)
if rc==0 then
    print "\nHID Report Created. Handle: "; nHandle
endif

// Write a 24 bit string at an offset of 8 bits in the report
rc = HIDReportAppendStr(nHandle, 8, 24, str$)
if rc==0 then
    print "\nSuccessfully appended string"
endif
```

#### Expected Output when the string is successfully imported into the report:

```
HID Report Created. Handle: 130060
Successfully imported string
```

## HIDReportExport

### FUNCTION

This function exports a HID report from a HID report object to a string.

#### HIDReportExport (nHandle, bitLen, sReport\$)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVAL nHandle as INTEGER</b> Handle of the report to be written to.
<b>bitLen</b>	<b>byREF bitLen as INTEGER</b> Returns the length of the exported string in bits.
<b>sReport</b>	<b>byREF sReport as STRING</b> Returns the contents of the HID report as a string.

#### Example:

```

dim rc, nHandle

// String to write into the report
dim strImport$ : strImport$ = "abcde"
dim nLen
dim strExport$

// Initialise a report with a length of 40 bits (5 bytes)
rc = HIDReportInit(40, nHandle)
if rc==0 then
    print "\nHID Report Created. Handle: "; nHandle
endif

// Write a 40 bit string to the report. This overwrites the whole report
rc = HIDReportImport(nHandle, 40, strImport$)

// Export the HID report to a string
rc = HIDReportExport(nHandle, nLen, strExport$)
if rc==0 then
    print "\nSuccessfully exported report. "
    print "Length: "; nLen; " Report: "; strExport$
endif

```

#### Expected Output when the report is successfully exported to a string:

```

HID Report Created. Handle: 130060
Successfully exported reported. Length: 40 Report: abcde

```

## HIDReportExtractInt

### FUNCTION

This function extracts an integer from a HID report object at an offset bitIndex of length bitLen.

#### HIDReportExtractInt (nHandle, bitIndex, bitLen, nVal)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVAL nHandle as INTEGER</b> Handle of the report to be written to.
<b>bitIndex</b>	<b>byVAL bitIndex as INTEGER</b> Location in the report to read the integer from, defined in bits.
<b>bitLen</b>	<b>byVAL bitLen as INTEGER</b> Length of the integer to extract in bits.
<b>nVal</b>	<b>byREF nVal as INTEGER</b> Returns the contents of the HID report as a string.

#### Example:

```
dim rc, nHandle
// String to write into the report
dim strImport$ : strImport$ = "abcde"
dim nExtract

// Initialise a report with a length of 40 bits (5 bytes)
rc = HIDReportInit(40, nHandle)
if rc==0 then
    print "\nHID Report Created. Handle: "; nHandle
endif

// Write a 40 bit string to the report. This overwrites the whole report
rc = HIDReportImport(nHandle, 40, strImport$)

// Extract the integer from the string at offset 16 and length 8
rc = HIDReportExtractInt(nHandle, 16, 8, nExtract)
if rc==0 then
    print "\nSuccessfully extracted integer: ";nExtract
endif
```

#### Expected Output when an integer is successfully extracted from the report:

```
HID Report Created. Handle: 130060
Successfully extracted integer: 142
```

## HIDReportExtractStr

### FUNCTION

This function extracts a string from a HID report object at an offset `bitIndex` of length `bitLen`.

#### HIDReportExtractStr (`nHandle`, `bitIndex`, `bitLen`, `sVal`)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nHandle</i></b>	<b>byVAL nHandle as INTEGER</b> Handle of the report to be written to.
<b><i>bitIndex</i></b>	<b>byVAL bitIndex as INTEGER</b> Location in the report to read the integer from, defined in bits.
<b><i>bitLen</i></b>	<b>byVAL bitLen as INTEGER</b> Length of the integer to extract in bits.
<b><i>sVal</i></b>	<b>byREF sVal as STRING</b> Returns the contents of the HID report as a string.

#### Example:

```
dim rc, nHandle
// String to write into the report
dim strImport$ : strImport$ = "abcde"

dim strExtract$

// Initialise a report with a length of 40 bits (5 bytes)
rc = HIDReportInit(40, nHandle)
if rc==0 then
    print "\nHID Report Created. Handle: "; nHandle
endif

// Write a 40 bit string to the report. This overwrites the whole report
rc = HIDReportImport(nHandle, 40, strImport$)

// Extract the integer from the string at offset 8 and length 16
rc = HIDReportExtractStr(nHandle, 8, 16, strExtract$)
if rc==0 then
    print "\nSuccessfully extracted string: ";strExtract$
endif
```

### Expected Output when a string is successfully extracted from the report:

```
HID Report Created. Handle: 130060
Successfully extracted string: bc
```

## HIDReportDestroy

### FUNCTION

This function destroys a HID report object.

### HIDReportDestroy (nHandle)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVAL nHandle as INTEGER</b> Handle of the report to be written to.

### Example:

```
dim rc, nHandle

// Initialise a report with a length of 40 bits (5 bytes)
rc = HIDReportInit(40, nHandle)
if rc==0 then
    print "\nHID Report Created. Handle: "; nHandle
endif

// Destroy the report
rc = HIDReportDestroy(nHandle)
if rc==0 then
    print "\nSuccessfully destroyed report"
endif
```

### Expected Output when report is successfully destroyed:

```
HID Report Created. Handle: 130060
Successfully destroyed report
```

## RTC ALARM

The BT900 module contains an inbuilt RTC clocked from an internal 32768Hz clock. This feature has two main functions:

- To provide a time stamp to define when data was gathered
- To set a time or duration to wake up from Deep Sleep

All the functions where the time is set or read use the 24 hour clock. Leap years are defined as those years divisible by 4; year 00 is a leap year.

The time and date can be set through the relevant smartBasic commands described below and also by the `at+rtc` command. The module will keep time from this point until it is powered down. Then the time and date will reset to 00:00:00 01/01/00, i.e. midnight on the 1<sup>st</sup> January year 00.

The alarm can only be used to wake up the module from Deep Sleep. If the module is not in Deep Sleep, no events will be returned if an alarm triggers.

Once the alarm has been set by any of the commands below the module can either be put automatically into Deep Sleep mode using the `nSleep` parameter. If you should choose to manually put the module into Deep Sleep mode you can use the `SystemStateSet(0)` command. Note that you should ensure everything has been completed before putting the module into Deep Sleep mode. The call to put the module into Deep Sleep, either through one of the set alarm commands or `SystemStateSet(0)`, should be the last statement the smartBasic application should make.

On reawakening the firmware performs a full system reset and the Program counter will be loaded with the reset vector. The module will go through the full initialisation procedure. Should there be an `$autorun$` smartBasic application present, this will restart from the first command. In this way it is possible to continually perform some functionality and then go to sleep in a never ending loop, that can only be broken by disabling the autorun pin. Partial examples of this are given below.

---

**Note:** It is possible to set an RTC Alarm without setting the current time. Once it is powered on, the module starts keeping track of the time from 00:00:00 01/01/00. It is possible to get the current time and then calculate the time you want the alarm to trigger from there. Or you could use the `RTCSETALARMMDURATION()` command which causes the alarm to trigger after a set number of hours and minutes.

---

There are also alarm options to trigger every hour, minute or day. It is important to note for the hour and minute alarms that the first alarm may trigger before you expect. They do not trigger AFTER a minute or hour, but when the minute or hour counters tick over. So if you set the minute alarm at 11:55:55 then it will trigger at 11:56:00 (after only 5 seconds).

If this application is being run in autorun mode, all subsequent triggers would occur when expected.

The RTC function can also be used to provide a time stamp using the `RTCGETTIME()` smartBasic command. However, for this to be meaningful the time should be set to the current date and time.

Note that the RTC will reset back to 00:00:00 01/01/00 if the module is powered down, or is reset using the Reset line, either during normal working conditions or waking up from Deep Sleep.



## RTCSetTime

### FUNCTION

This function sets the date and time of the RTC.

#### RTCSETTIME (nYear,nMonth,nDay,nHour,nMin,nSec)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nYear</b>	<b>byVal nYear AS INTEGER</b> The current year.
<b>nMonth</b>	<b>byRef nMonth AS INTEGER</b> The current month.
<b>nDay</b>	<b>byVal nDay AS INTEGER</b> The current day of the month.
<b>nHour</b>	<b>byVal nHour AS INTEGER</b> The current hour.
<b>nMin</b>	<b>byVal nMin AS INTEGER</b> The current minute.
<b>nSec</b>	<b>byVal nSec AS INTEGER</b> The current second.

**Note:** The following test application was run with the format set to 4.

```
// ***** RTCSet.sb*****
dim rc
dim strTime$

function handleTimer0()
    rc = RTCGetTime$(strTime$)
    print "\n";strTime$
    strTime$ = ""
endfunc 1

function handleTimer1()
    print "\nTimer 1 has expired"
endfunc 0

onevent EvTmr0 call handleTimer0
onevent EvTmr1 call handleTimer1

rc = RTCSetTime(15,1,8,23,55,0)
print "\nSetting Time ";integer.h' rc
TimerStart(0,2000,1)
```

```
TimerStart(1,10000,0)

WAITEVENT

print "\nfinished"
```

#### Expected Output:

```
Setting Time 00000000
23:55:02 08/01/15
23:55:04 08/01/15
23:55:06 08/01/15
23:55:08 08/01/15
23:55:10 08/01/15
Timer 1 has expired
finished
00
```

## RTCGetTime\$

### FUNCTION

This function returns a string indicating the date and/or time. The format of the string is as defined in the RTCSETFORMAT() command.

#### RTCGETTIME\$(strTime\$)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>strTime\$</b>	<b>byREF strTime\$ AS STRING</b> Returns the date and/or time, formatted as defined in the RTCSETFORMAT() command.

Refer to other functions in this section for examples of this function.

## RTCGetTime

### FUNCTION

This function returns the time and date as a series of 6 integers, denoting the year, month, day, hour, minute and second.

#### RTCGETTIME (nYear,nMonth,nDay,nHour,nMin,nSec)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nYear</b>	<b>byREF nYear AS INTEGER</b> Returns the current year.
<b>nMonth</b>	<b>byREF nMonth AS INTEGER</b> Returns the current month.

<b><i>nDay</i></b>	<b>byREF <i>nDay</i> AS INTEGER</b> Returns the current day of the month.
<b><i>nHour</i></b>	<b>byREF <i>nHour</i> AS INTEGER</b> Returns the current hour.
<b><i>nMin</i></b>	<b>byREF <i>nMin</i> AS INTEGER</b> Returns the current minute.
<b><i>nSec</i></b>	<b>byREF <i>nSec</i> AS INTEGER</b> Returns the current second.

Refer to other functions in this section for examples of this function being used.

## RTCSetAlarm

### FUNCTION

This function sets the date and time when the alarm will be triggered.

#### RTCSETALARM(*nHandle*,*nYear*,*nMonth*,*nDay*,*nHour*,*nMin*,*nSleep*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation. If the function attempts to set the alarm to a time and date earlier than the present date then error 0x5246 will be returned.
<b>Arguments:</b>	
<b><i>nHandle</i></b>	byVal <i>nHandle</i> AS INTEGER ID of the alarm. This parameter is not used in the BT900 and should be set to 0.
<b><i>nYear</i></b>	byVal <i>nYear</i> AS INTEGER The year the alarm is to trigger.
<b><i>nMonth</i></b>	byRef <i>nMonth</i> AS INTEGER The month the alarm is to trigger.
<b><i>nDay</i></b>	byVal <i>nDay</i> AS INTEGER The day of the month the alarm is to trigger.
<b><i>nHour</i></b>	byVal <i>nHour</i> AS INTEGER The hour of day the alarm is to trigger.
<b><i>nMin</i></b>	byVal <i>nMin</i> AS INTEGER The minute the alarm is to trigger
<b><i>nSleep</i></b>	byVal <i>nSleep</i> AS INTEGER <ul style="list-style-type: none"> <li>0: Module stays in normal running mode</li> <li>1: Module goes into Deep Sleep Mode</li> </ul>

For the following test the time had been preset to 23:55:00 08/01/15 by another *smart*BASIC application. RTCSetAlarm() was then set up as the \$autorun\$.sb application and left to run.

In this test application, the alarm has been set to trigger at 00:02:00 09/01/15. The module wakes up and runs the application printing the downloaded time at that moment. The one second delay is the time it takes the module to run the application from reset.

The application then attempts to set the alarm with the same time. As this time has already passed an error message is returned and the application closes.

```
//*****RTCSetAlarm.sb*****
dim rc
dim strAlarm$
```

```
function handleTimer0()  
  print "\nsleep"  
  rc = RTCSetAlarm(0,15,1,9,0,2,1) // set to wake up at 00:02 9/1/15  
  if rc != 0 then  
    print "\nfailed to set alarm ",integer.h' rc  
  endif  
endfunc 0  
  
onevent EvTmr0 call handleTimer0  
  
rc = RTCSetFormat(4)  
rc = RTCGetTime$(strAlarm$)  
print "\n";strAlarm$  
TimerStart(0,1000,0)  
  
WAITEVENT  
print "\nfinished"
```

#### Expected Output:

```
23:55:11 08/01/15  
sleep  
00:02:01 09/01/15  
sleep  
failed to set alarm      00005246  
finished
```

## RTCSetAlarmDuration

### FUNCTION

This function sets the alarm as a duration from the current time. The will automatically retrieve the current time and calculate the desired alarm time using the passed in parameters. This functionality is restricted to just 23 hours and 59 minutes in advance of the current time.

**Note:** The RTC alarm used in this function can only trigger on full minutes. Therefore the hour and minute set as the duration may be rounded down. For example if the time was 09:00:55 and a duration of 2 minutes is set, the module will wake up at 09:02:00, ie after 1 minute 5 seconds.

If this function is used in a recurring manner, subsequent wakeups will happen on time as the next wake up time will be calculated from the time the module wakes up and so the rounding down will not occur.

## RTCSETALARM DURATION (nHandle,nMin,nSleep)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVal nHandle AS INTEGER</b> Id of the alarm. This parameter is not used in the BT900 and should be set to 0.
<b>nHour</b>	<b>byVal nHour AS INTEGER</b> The number of hours before the alarm is triggered.
<b>nMin</b>	<b>byVal nMin AS INTEGER</b> The number of minutes before the alarm is triggered.
<b>nSleep</b>	<b>byVal nSleep AS INTEGER</b> <ul style="list-style-type: none"> <li>0: Module stays in normal running mode</li> <li>1: Module goes into Deep Sleep Mode</li> </ul>

For the following test the time had been preset to 23:55:00 08/01/15 by another *smart*BASIC application. RTCSetMinute() was then set up as the \$autorun\$.sb application and left to run.

The alarm is to trigger every six minutes. Once the alarm has been set, the alarm time is read back. You will notice that the first period has been rounded down to 05:49. This is explained in the not above. After the first iteration the module is woken every 6 minutes as requested.

```
//*****RTCSetDuration.sb*****
dim rc
dim strAlarm$

function handleTimer0()
    strAlarm$ = ""
    rc = RTCSetAlarmDuration(0,0,6,0) // set to trigger in 6 minutes
    rc = RTCGetAlarm$(0,strAlarm$)
    print "\nAlarm set to ";strAlarm$
    print "\nsleep"
    rc = SystemStateSet(0)
    // or
    // rc = RTCSetAlarmDuration(0,0,6,1)
endfunc 0

onevent EvTmr0 call handleTimer0

rc = RTCSetFormat(4)
rc = RTCGetTime$(strAlarm$)
print "\n";strAlarm$
TimerStart(0,1000,0)

WAITEVENT
print "\nfinished"
```

### Expected Output:

```
23:55:11 08/01/15
Alarm set to 00:01:00 09/01/15
sleep
00:01:01 09/01/15
Alarm set to 00:07:00 09/01/15
sleep
00:07:01 09/01/15
Alarm set to 00:13:00 09/01/15
sleep
00:13:01 09/01/15
Alarm set to 00:19:00 09/01/15
sleep
```

## RTCGetAlarm\$

### FUNCTION

This function returns the date and time when the alarm will be triggered. This is only relevant to instances where the alarm has been set by the RTCSETALARM() or RTCSETALARMDURATION() commands.

Note that this command can only be used if the nSleep parameter in the set alarm commands is set to 0. If this parameter is set to 1 then the module will go into Deep Sleep mode and no further commands will be accessed.

### RTCGETALARM\$(nHandle,strTime\$)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVal nHandle AS INTEGER</b> ID of the alarm. This parameter is not used in the BT900 and should be set to 0.
<b>strTime\$</b>	<b>byREF strTime\$ AS STRING</b> Returns the date and/or time, formatted as defined in the RTCSETFORMAT() command, that the alarm is to trigger when set by the RTCSETALARM() or RTCSETALARMDURATION() commands.

See the above section for an example of how to use this function in the RTCAlarmDuration() example application.

## RTCGetAlarm

### FUNCTION

This function returns the date and time when the alarm will be triggered. This is only relevant to instances where the alarm has been set by the RTCALARM() or RTCALARMDURATION() commands.

Note that this command can only be used if the nSleep parameter in the set alarm commands is set to 0. If this parameter is set to 1 then the module will go into Deep Sleep mode and no further commands will be accessed.

## RTCGETALARM (strnHandle,nYear,nMonth,nDay,nHour,nMin)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nHandle</b>	<b>byVal nHandle AS INTEGER</b> Id of the alarm. This parameter is not used in the BT900 and should be set to 0.
<b>nYear</b>	<b>byREF nYear AS INTEGER</b> Returns year the alarm is set to trigger.
<b>nMonth</b>	<b>byREF nMonth AS INTEGER</b> Returns month the alarm is set to trigger.
<b>nDay</b>	<b>byREF nYear AS INTEGER</b> Returns day the alarm is set to trigger.
<b>nHour</b>	<b>byREF nYear AS INTEGER</b> Returns hour the alarm is set to trigger.
<b>nMin</b>	<b>byREF nYear AS INTEGER</b> Returns minute the alarm is set to trigger.

See the above section for an example of how to use this function in the RTCSetAlarmDuration() example application.

## RTCSetFormat

### FUNCTION

This function sets the format the the returned date/time string from the RTCGETTIME() and RTCGETALARM commands().

## RTCSETFORMAT (nFormat)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b>nFormat</b>	<b>byVal nFormat AS INTEGER</b> Defines the format of the Date/Time string returned by RTCGETTIME() and RTCGETALARM() <ul style="list-style-type: none"> <li>Time only - hh:mm:ss</li> <li>Date (UK) only - dd/mm/yy</li> <li>Date (US) only - yy/mm/dd</li> <li>Time and Date(UK) - hh:mm:ss dd/mm/yy</li> <li>Time and Date(US) - hh:mm:ss yy/mm/dd</li> <li>Date(UK) and Time - dd/mm/yy hh:mm:ss</li> <li>Date(US) and Time - yy/mm/dd hh:mm:ss</li> </ul>

```
// ***** RTCSetFormat.sb*****
dim rc
dim strTime$
dim nFormat
dim nYear
dim nMonth
dim nDay
dim nHour
dim nMin
dim nSec
```

```
function handleTimer0()  
    strTime$ = ""  
    nFormat = nFormat + 1  
    if nFormat > 7 then  
        exitfunc 0  
    endif  
    rc = RTCSetFormat(nFormat)  
    rc = RTCGetTime$(strTime$)  
    print "\n";strTime$  
    rc = RTCGetTime(nYear,nMonth,nDay,nHour,nMin,nSec)  
    print "\n";nHour;" ";nMin;" ";nSec;" ";nDay;" ";nMonth;" ";nYear  
endfunc 1  
  
onevent EvTmr0 call handleTimer0  
  
rc = RTCSetFormat(4)  
rc = RTCSetTime(15,1,8,23,55,0)    // Sets date to 23:55:00 25/2/11  
print "\nSetting Time ";integer.h' rc  
rc = RTCGetTime$(strTime$)  
print "\n";strTime$  
rc = RTCGetTime(nYear,nMonth,nDay,nHour,nMin,nSec)  
print "\n";nHour;" ";nMin;" ";nSec;" ";nDay;" ";nMonth;" ";nYear  
nFormat = 0  
  
TimerStart(0,1000,1)  
  
WAITEVENT  
print "\nfinished"
```



### Expected Output:

```
Setting Time 00000000
23:55:00 08/01/15
23 55 0 8 1 15
23:55:01
23 55 1 8 1 15
08/01/15
23 55 2 8 1 15
15/01/08
23 55 3 8 1 15
23:55:04 08/01/15
23 55 4 8 1 15
23:55:05 15/01/08
23 55 5 8 1 15
08/01/15 23:55:06
23 55 6 8 1 15
15/01/08 23:55:07
23 55 7 8 1 15
finished
00
```

## RTCSetMinuteAlarm

### FUNCTION

This function sets the alarm to trigger when the minute value changes.

### RTCSETMINUTEALARM(*nHandle*,*nSleep*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nHandle</i></b>	<b>byVal <i>nHandle</i> AS INTEGER</b> ID of the alarm. This parameter is not used in the BT900 and should be set to 0.
<b><i>nSleep</i></b>	<b>byVal <i>nSleep</i> AS INTEGER</b> <ul style="list-style-type: none"> <li>0: Module stays in normal running mode</li> <li>1: Module goes into Deep Sleep Mode</li> </ul>

For the following test the time had been preset to 23:55:00 08/01/15 by another *smart*BASIC application. RTCSetMinuteAlarm() was then set up as the \$autorun\$.sb application and left to run.

In the expected output you will see that the module wakes up every minute and prints out the current time before going back to sleep. The reason that the recorded time does not show zeroes is that it takes the module a second to wake up and to start running the *smart*BASIC application.

```
/*******RTCSetMinuteAlarm.sb*****
```

```
dim rc
dim strAlarm$

function handleTimer0()
  print "\nsleep"
  rc = RTCSetMinuteAlarm(0,1)
  // or
  // rc = RTCSetMinuteAlarm(0,0)
  // rc = SystemStateSet(0)
endfunc 0

onevent EvTmr0 call handleTimer0

rc = RTCSetFormat(4)
rc = RTCGetTime$(strAlarm$)
print "\n";strAlarm$
TimerStart(0,1000,0)

WAITEVENT
print "\nfinished"
```

#### Expected Output:

```
23:55:11 08/01/15
sleep
23:56:01 08/01/15
sleep
23:57:01 08/01/15
sleep
23:58:01 08/01/15
sleep
23:59:01 08/01/15
sleep
```

## RTCSetHourAlarm

### FUNCTION

This function sets the alarm to trigger when the hour value changes.

### RTCSETHOUR ALARM(*nHandle*,*nSleep*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nHandle</i></b>	<b>byVal <i>nHandle</i> AS INTEGER</b> ID of the alarm. This parameter is not used in the BT900 and should be set to 0.
<b><i>nSleep</i></b>	<b>byVal <i>nSleep</i> AS INTEGER</b> <ul style="list-style-type: none"> <li>0: Module stays in normal running mode</li> <li>1: Module goes into Deep Sleep Mode</li> </ul>

As with the previous section, the RTC was set to 23:55:00 08/01/15 and the RTCSetHourAlarm() function was set up as the \$autorun\$ function.

In the expected output you will see that the module has recorded the time at 00:00:00 09/01/15 as the change of hour has also taken the RTC into the next day.

```

/*****RTCSetHourAlarm.sb*****/
dim rc
dim strAlarm$

function handleTimer0 ()
    print "\nsleep"
    rc = RTCSetHourAlarm(0,1)
    // or
    // rc = RTCSetHourAlarm(0,0)
    // rc = SystemStateSet(0)
endfunc 0

onevent EvTmr0 call handleTimer0

rc = RTCSetFormat(4)
rc = RTCGetTime$(strAlarm$)
print "\n";strAlarm$
TimerStart(0,1000,0)

WAITEVENT
print "\nfinished"

```

## Expected Output:

```
23:55:11 08/01/15
sleep
00:00:01 09/01/15
sleep
```

## RTCSetDayAlarm

### FUNCTION

This function sets the alarm to trigger when the the date changes at midnight, i.e. when the clock changes from 23:59:59 to 00:00:00.

### RTCSETDAYALARM(*nHandle*,*nSleep*)

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
<b>Arguments:</b>	
<b><i>nHandle</i></b>	<b>byVal <i>nHandle</i> AS INTEGER</b> ID of the alarm. This parameter is not used in the BT900 and should be set to 0.
<b><i>nSleep</i></b>	<b>byVal <i>nSleep</i> AS INTEGER</b> <ul style="list-style-type: none"> <li>0: Module stays in normal running mode</li> <li>1: Module goes into Deep Sleep Mode</li> </ul>

This example is very similar to that of the RTCSetHourAlarm() application with the exception that the alarm triggered on the changing day and not the changing hour.

```
/*******RTCSetDayAlarm.sb*****
```

```
dim rc
dim strAlarm$

function handleTimer0()
  print "\nsleep"
  rc = RTCSetDayAlarm(0,1)
  // or
  // rc = RTCSetDayAlarm(0,0)
  // rc = SystemStateSet(0)
endfunc 0

onevent EvTmr0 call handleTimer0

rc = RTCSetFormat(4)
rc = RTCGetTime$(strAlarm$)
print "\n";strAlarm$
TimerStart(0,1000,0)

WAITEVENT
print "\nfinished"
```

### Expected Output:

```
23:55:11 08/01/15
sleep
00:00:01 09/01/15
sleep
```

## RTCReset

### FUNCTION

This function resets the RTC back to the default value of 00:00:00 01/01/00, i.e. midnight on the 1<sup>st</sup> January 00.

### RTCRESET()

<b>Returns</b>	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
----------------	---

#### Arguments:

```
// ***** RTCReset.sb*****
dim rc
dim strTime$

rc = RTCSetFormat(4)           // outputs time as hh:mm:ss dd/mm/yy
rc = RTCSetTime(15,1,8,23,55,0) // Sets date to 23:55:00 08/01/15
print "\nSetting Time ";integer.h' rc
rc = RTCGetTime$(strTime$)
print "\n";strTime$
rc = RTCReset()                // Resets time to 00:00:00 01/01/00
print "\nSetting Time ";integer.h' rc
strTime$ = ""
rc = RTCGetTime$(strTime$)
print "\n";strTime$

print "\nfinished"
```

### Expected Output:

```
Setting Time 00000000
23:55:00 08/01/15
Setting Time 00000000
00:00:00 01/01/00
finished
00
```

## LOW POWER MODES

The BT900 utilises two low power modes: Standby Doze and Deep Sleep.

The module will automatically go into StandbyDoze when it is sitting at a WAITEVENT, and will wake up on any interrupt. If it receives a character over the UART, it will automatically wake up and deal with that character. No UART data will be lost from standby doze mode.

Enter Deep sleep either by putting the UART Rx line into a break condition (signal low) or by calling the SystemStateSet(0) command (either explicitly or through one of the RTC alarms described in the RTC Alarm section. From Deep Sleep the module performs a system reboot. The module can be woken from Deep Sleep mode, either via an RTC interrupt or one of the wake up pins, defined in [SIOPinFunctionality](#) above.

The module can also be woken up from Deep Sleep using the Reset line, however this method will cause the RTC to reset to 00:00:00 01/01/00.

If you want to wake up via a Wakeup Pin then that pin will need to be set up as described in [GpioSetFunc](#). An example is shown below where the RTC is initially set to 23:55:00 08/01/15. DeepSleepWakeup.sb outputs the time for 10s when it is first run and then sets up Wakeup 1 (sio20) as the wakeup pin. (This is button 2 on the development board.) The module is then put into Deep Sleep mode. Wakeup 1 is then pulsed low, the module is woken up and DeepSleepWakeup.sb ran again. This process continues indefinitely until the autorun option is disabled.

For this example, DeepSleepWakeup.sb is set up as the \$autorun\$ function.

```
// ***** DeepSleepWakeup.sb *****  
  
dim rc  
dim strTime$  
  
function handleTimer0()  
    rc = RTCGetTime$(strTime$)  
    print "\n";strTime$  
    strTime$ = ""  
endfunc 1  
  
function handleTimer1()  
    rc = GpioSetFunc(20,1,16)  
    print "\nSleep"  
    rc = SystemStateSet(0)  
endfunc 0  
  
onevent EvTmr0 call handleTimer0  
onevent EvTmr1 call handleTimer1  
  
TimerStart(0,2000,1)  
TimerStart(1,10000,0)
```

```
WAITEVENT
```

```
print "\nfinished"
```

**Expected Output:**

```
23:55:23 08/01/15
23:55:25 08/01/15
23:55:27 08/01/15
23:55:29 08/01/15
23:55:31 08/01/15
Sleep
23:55:53 08/01/15
23:55:55 08/01/15
23:55:57 08/01/15
23:55:59 08/01/15
23:56:01 08/01/15
Sleep
23:57:11 08/01/15
23:57:13 08/01/15
23:57:15 08/01/15
23:57:17 08/01/15
23:57:19 08/01/15
Sleep
```

## EVENTS AND MESSAGES

*smart*BASIC is designed to be event driven, which makes it suitable for embedded platforms where it is normal to wait for something to happen and then respond.

The event handling is done synchronously, meaning the *smart*BASIC runtime engine has to process a WAITEVENT statement for any events or messages to be processed. This guarantees that *smart*BASIC never needs the complexity of locking variables and objects.

The subsystems which generate events and messages relevant to the routines described in this guide are as follows:

- BLE events and messages as described [here](#).
- Generic Characteristics events and messages as described [here](#).

## MISCELLANEOUS

### Bluetooth Result Codes

There are some operations and events that provide a single byte Bluetooth HCI result code (such as the EVDISCON message). The meaning of the result code is as per the list reproduced from the Bluetooth Specifications below. No guarantee is supplied as to its accuracy. Consult the specification for more.

Result codes in **grey** are not relevant to Bluetooth Low Energy operation.

<b>BT_HCI_STATUS_CODE_SUCCESS</b>	<b>0x00</b>
<b>BT_HCI_STATUS_CODE_UNKNOWN_BTLE_COMMAND</b>	<b>0x01</b>
<b>BT_HCI_STATUS_CODE_UNKNOWN_CONNECTION_IDENTIFIER</b>	<b>0x02</b>
BT_HCI_HARDWARE_FAILURE	0x03
BT_HCI_PAGE_TIMEOUT	0x04
<b>BT_HCI_AUTHENTICATION_FAILURE</b>	<b>0x05</b>
<b>BT_HCI_STATUS_CODE_PIN_OR_LINKKEY_MISSING</b>	<b>0x06</b>
<b>BT_HCI_MEMORY_CAPACITY_EXCEEDED</b>	<b>0x07</b>
<b>BT_HCI_CONNECTION_TIMEOUT</b>	<b>0x08</b>
BT_HCI_CONNECTION_LIMIT_EXCEEDED	0x09
BT_HCI_SYNC_CONN_LIMIT_TO_A_DEVICE_EXCEEDED	0x0A
BT_HCI_ACL_CONNECTION_ALREADY_EXISTS	0x0B
<b>BT_HCI_STATUS_CODE_COMMAND_DISALLOWED</b>	<b>0x0C</b>
BT_HCI_CONN_REJECTED_DUE_TO_LIMITED_RESOURCES	0x0D
BT_HCI_CONN_REJECTED_DUE_TO_SECURITY_REASONS	0x0E
BT_HCI_BT_HCI_CONN_REJECTED_DUE_TO_BD_ADDR	0x0F
BT_HCI_CONN_ACCEPT_TIMEOUT_EXCEEDED	0x10
BT_HCI_UNSUPPORTED_FEATURE_ONPARAM_VALUE	0x11
<b>BT_HCI_STATUS_CODE_INVALID_BTLE_COMMAND_PARAMETERS</b>	<b>0x12</b>
<b>BT_HCI_REMOTE_USER_TERMINATED_CONNECTION</b>	<b>0x13</b>
<b>BT_HCI_REMOTE_DEV_TERMINATION_DUE_TO_LOW_RESOURCES</b>	<b>0x14</b>
<b>BT_HCI_REMOTE_DEV_TERMINATION_DUE_TO_POWER_OFF</b>	<b>0x15</b>
<b>BT_HCI_LOCAL_HOST_TERMINATED_CONNECTION</b>	<b>0x16</b>
BT_HCI_REPEATED_ATTEMPTS	0x17
BT_HCI_PAIRING_NOTALLOWED	0x18
BT_HCI_LMP_PDU	0x19
<b>BT_HCI_UNSUPPORTED_REMOTE_FEATURE</b>	<b>0x1A</b>
BT_HCI_SCO_OFFSET_REJECTED	0x1B



BT_HCI_SCO_INTERVAL_REJECTED	0x1C
BT_HCI_SCO_AIR_MODE_REJECTED	0x1D
<b>BT_HCI_STATUS_CODE_INVALID_LMP_PARAMETERS</b>	<b>0x1E</b>
<b>BT_HCI_STATUS_CODE_UNSPECIFIED_ERROR</b>	<b>0x1F</b>
BT_HCI_UNSUPPORTED_LMP_PARM_VALUE	0x20
BT_HCI_ROLE_CHANGE_NOT_ALLOWED	0x21
<b>BT_HCI_STATUS_CODE_LMP_RESPONSE_TIMEOUT</b>	<b>0x22</b>
BT_HCI_LMP_ERROR_TRANSACTION_COLLISION	0x23
<b>BT_HCI_STATUS_CODE_LMP_PDU_NOT_ALLOWED</b>	<b>0x24</b>
BT_HCI_ENCRYPTION_MODE_NOT_ALLOWED	0x25
BT_HCI_LINK_KEY_CAN_NOT_BE_CHANGED	0x26
BT_HCI_REQUESTED_QOS_NOT_SUPPORTED	0x27
<b>BT_HCI_INSTANT_PASSED</b>	<b>0x28</b>
<b>BT_HCI_PAIRING_WITH_UNIT_KEY_UNSUPPORTED</b>	<b>0x29</b>
<b>BT_HCI_DIFFERENT_TRANSACTION_COLLISION</b>	<b>0x2A</b>
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
<b>BT_HCI_CONTROLLER_BUSY</b>	<b>0x3A</b>
<b>BT_HCI_CONN_INTERVAL_UNACCEPTABLE</b>	<b>0x3B</b>
<b>BT_HCI_DIRECTED_ADVERTISER_TIMEOUT</b>	<b>0x3C</b>
<b>BT_HCI_CONN_TERMINATED_DUE_TO_MIC_FAILURE</b>	<b>0x3D</b>
<b>BT_HCI_CONN_FAILED_TO_BE_ESTABLISHED</b>	<b>0x3E</b>

## ACKNOWLEDGEMENTS

The following are required acknowledgements to address our use of open source code on the BT900 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.