BGSCRIPT SCRIPTING LANGUAGE

DEVELOPER GUIDE

Wednesday, 28 May 2014

Version 3.6

Table of Contents

1 Version History

Version	Comments
2.3	BGScript limitations updated with performance comments
2.4	Added new features included in v.1.1 software. Small improvements made into BGScript examples Added a 4-channel PWM example
2.5	Reading ADC does not disable IO interrupts
2.6	Added battery reading example using the internal battery monitor
2.7	Updated ADC internal reference to 1.24V (was 1.15V)
3.0	BLE SW1.2 additions and changes: • Procedure support added • Memset support for buffer handling added • Limitations section aligned with the new SW enhancements In addition, editorial improvements are done within the document.
3.1	Improved BGScript syntax documentation
3.2	I2C example improved and corrected
3.3	Splitting BGScript into multiple files through IMPORT and export directive made possible
3.4	Improvements to BGScript syntax description
3.5	Bluetooth Smart Software 1.3.0 compatible document version. The limitation for the maximum size of all DIM variables is removed.
3.6	Editorial changes

2 Introduction -- BGscript

This document briefly describes the Bluegiga BGScript programming language for Bluegiga *Bluetooth* Smart Products. The document briefly explains what BGScript programming language is, what are the benefits, for what it can be used for and what are the limitations. The document also contains multiple examples of BGScript code and API and how it can be used to perform various task such as detecting *Bluetooth* connections, receiving and transmitting data and managing the hardware interfaces like UART, SPI and I2C.

3 What is BGScript?

3.1 BGScript Scripting Language

Bluegiga BGScript is a simple BASIC-style programming language that allows end-user applications to be embedded to the Bluegiga *Bluetooth* Smart modules. The benefit of using BGScript is that one can create fully standalone *Bluetooth* Smart devices without the need of an external MCU and this enables further size, cost and power consumption reductions. Although being a simple and easy-to-learn programming language BGScript does provide features and functions to create fairly complex and powerful applications and it provides the necessary APIs for managing Bluetooth connections, security, data transfer and various hardware interfaces such as UART, USB, SPI, I2C, GPIO, PWM and ADC.

BGScript is fully event based programming language and code execution is started when events such as system start-up, *Bluetooth* connection, I/O interrupt etc. occur.

BGScript applications are developed with Bluegiga's free-of-charge *Bluetooth* Smart SDK and the BGScript applications are executed in the BGScript Virtual Machine (VM) that is part of the Bluegiga Bluetooth Smart software. The Bluetooth Smart SDK comes with all the necessary tools for code editing and compilation and also the needed tools for installing the complied firmware binary to the Bluegiga Bluetooth Smart modules. Multiple example applications and code snipplets are also available for Bluegiga implementing applications like thermometers, heart rate transmitters, medical sensors and iBeacons just to mention a few.

The illustration below describes the Bluegiga Bluetooth Smart software, API and how BGScript VM and applications interface to it.

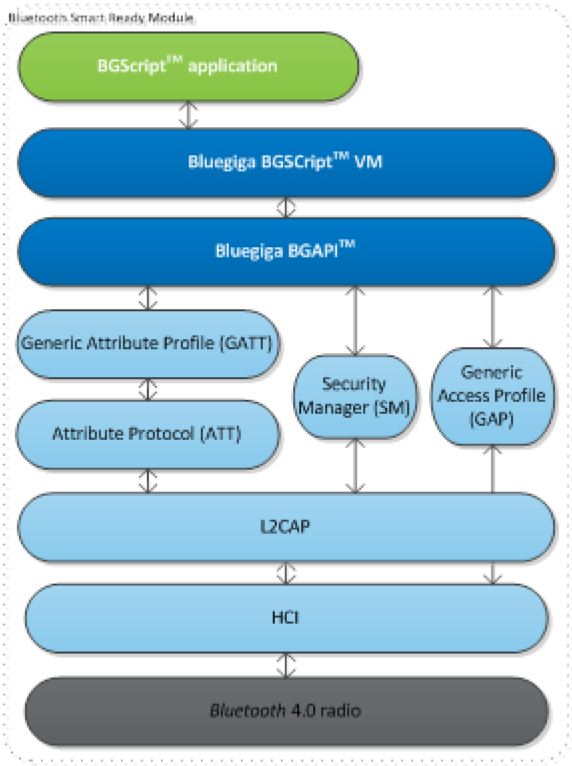

Figure: BGScript System Architecture

A simple BGScript code example:

```
# system started, occurs on boot or reset
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)

    # Enable BLE advertising mode
    call gap_set_mode(gap_general_discoverable, gap_undirected_connectable)

    # Enable BLE bonding mode
    call sm_set_bondable_mode(1)

    # Start a repeating timer at 1-second interval (32768Hz = crystal frequency)
    call hardware_set_soft_timer(32768, 0, 0)
end
```

3.2 BGScript vs. BGAPI

BGScript applications are just one way of controlling the Bluegiga Bluetooth Smart modules and it may not be usable in every use case. For example the amount of available hardware interfaces, RAM or Flash may limit you to implement and execute your application on the microcontroller on-board the Bluegiga Bluetooth Smart modules. If this is the case an alternate way of controlling the module is the BGAPI protocol. BGAPI protocol is a simple binary based protocol that works over the physical UART and USB interfaces available on the Bluetooth Smart modules. An external host processor can be used to implement the end user application and this application can control the Bluetooth Smart modules using the BGAPI protocol.

> **When BGScript is enabled, the BGAPI protocol is disabled.** BGScript cannot be used at the same time as BGAPI control from an external host.

4 BGScript Syntax

The BGScript scripting language has BASIC-like syntax. Code is executed only in response to **events,** and each line of code is executed in order, starting from the beginning of the **event** definition and ending at a **return** or **end** statement. Each line represents a single command.

BGScript scripting language is currently supported by multiple Bluegiga's Bluetooth Smart and Wi-Fi products and the BGScript command and events are specific to the technology.

Below is a conceptual example of simple BGScript usage with Bluegiga Wi-Fi software. The code below is executed at the system start i.e. when the device is powered up and the code will start the Wi-Fi subsystem and connects to a Wi-Fi access point with the SSID "**test_ssid**".

Simple BGScript syntax example

```
# system start-up event listener
event system_boot(major, minor, patch, build, bootloader, tcpip, hw)
    # Turn Wi-Fi subsystem on
    call sme_wifi_on()
end

# Wi-Fi ON event listener
event sme_wifi_is_on(result)
    # connect to a network
    call sme_connect_ssid(9, "test_ssid")
end
```

4.1 Comments

Anything after a # character is considered as a comment, and ignored by the compiler.

```
x = 1 # comment
```

4.2 Variables and Values

4.2.1 Values

Values are always interpreted as integers (no floating-point numbers). Hexadecimal values can be expressed by putting **$** before the value. Internally, all values are 32-bit signed integers stored in memory in little-endian format.

```
x = 12      # same as x = $0c
y = 703710  # same as y = $abcde
```

IP addresses are automatically converted to their 32-bit decimal value equivalents.

```
x = 192.168.1.1  # same as x = $0101A8C0
```

4.2.2 Variables

Variables (not buffers) are signed 32-bit integer containers, stored in little-endian byte order. Variables must be defined before usage.

```
dim x
```

Example

```
dim x
dim y

x = (2 * 2) + 1
y = x + 2
```

4.2.3 Global Variables

Variables can be defined globally using **dim** definition which must be used outside an **event** block.

```
dim j

# software timer listener
event hardware_soft_timer(handle)
     j = j + 1
    call attributes_write(xgatt_counter, 2, j)
end
```

4.2.4 Constant Variables

Constants are signed 32-bit integers stored in little-endian byte order and they also need to be defined before use.. Constants can be particularly useful because they do not take up any of the limited RAM that is available to BGScript applications and instead constant values are stored in flash as part of the application code.

```
const x = 2
```

4.2.5 Buffers

Buffers hold 8-bit values and can be used to prepare or parse more complex data structures. For example a buffer might be used in a Bluetooth Smart on-module application to prepare an attribute value before writing it into the attribute database.

Like variables buffers need to be defined before usage. Currently the maximum size of a buffer is 256 bytes.

```
event hardware_io_port_status(delta, port, irq, state)
    tmp(0:1) = 2
    tmp(1:1) = 60 * 32768 / delta

    call attributes_write(xgatt_hr, 2, tmp(0:2))
end
```

```
dim u(10)
```

Buffers use an index notation with the following format:

BUFFER(<***expression***>:<***size***>)

The <***expression***> is used as the index of the first byte in the buffer to be accessed and <***size***> is used to specify how many bytes are used starting from the location defined by <***expression***>. Note that this ***<size>*** is **not** the end index position.

```
u(0:1) = $a
u(1:2) = $123
```

The following syntax could be used with the same result due to little-endian byte ordering:

```
u(0:3) = $1230a
```

When using constant numbers to initialize a buffer, only **four** (4) bytes may be set at a time. Longer buffers must be written in multiple parts or using a string literal (see **Strings** section below).


```
u(0:4) = $32484746
u(4:1) = $33
```

Using Buffers with Expressions

Buffers can also be used in mathematical expressions, but only a maximum of **four** (4) bytes are supported at a time since all numbers are treated as signed 32-bit integers in little-endian format. The following examples show valid use of buffers in expressions.

```
a = u(0:4)
a = u(2:2) + 1
u(0:4) = b
u(2:1) = b + 1
```

The following example is **not valid**:

```
if u(0:5) = "FGH23" then
    # do something
end if
```

This is because the mathematical equality operator ("=") interprets both sides as numerical values and in BGScript numbers are always 4 bytes (32 bits). This means you can only compare (with '=') buffer segments which are exactly four (4) bytes long. If you need to compare values which are not four (4) bytes in length you must use the **memcmp** function, which is described later in this document.

```
if u(1:4) = "GH23" then
    # do something
end if
```

4.2.6 Strings

Buffers can be initializated using literal string constants. Using this method more than four (4) bytes at a time may be assigned.

```
u(0:5) = "FGH23"
```

Literal strings support C-style escape sequences, so the following example will do the same as the above:

```
u(0:5) = "\x46\x47\x48\x32\x33"
```

Using this method you can assign and subsequently compare longer values such as 128-bit custom UUIDs for example when scanning or searching a GATT database for proprietary services or characteristics. However keep in mind that the data must be presented in little-endian format, so the value assigned here as a string literal should be the reverse of the 128-bit UUID entered into the **gatt.xml** UUID attributes if that is what you are searching for.

4.3 Expressions

Expressions are given in infix notation.

```
x = (1+2) * (3+1)
```

The following **mathematical operators** are supported:

Operation	Symbol
Addition:	+
Subtraction:	-
Multiplication:	*
Division:	/
Less than:	<
Less than or equal:	<=
Greater than:	>
Greater than or equal:	>=
Equals:	=
Not equals:	!=
Parentheses	()

Note

Currently there is no support for **modulo** or **power** operators.

The following **bitwise operators** are supported:

<table>
<tr><th>Operation</th><th>Symbol</th></tr>
<tr><td>AND</td><td>&</td></tr>
<tr><td>OR</td><td>|</td></tr>
<tr><td>XOR</td><td>^</td></tr>
<tr><td>Shift left</td><td><<</td></tr>
<tr><td>Shift right</td><td>>></td></tr>
</table>

The following **logical operators** are supported:

<table>
<tr><th>Operation</th><th>Symbol</th></tr>
<tr><td>AND</td><td>&&</td></tr>
<tr><td>OR</td><td>||</td></tr>
</table>

4.4 Commands

4.4.1 event <event_name> (< event_parameters >)

A code block defined between **event** and **end** keywords will be run in response to a specific event. Execution will stop when reaching **end** or **return**. BGScript VM queues each event generated by the API and executes them in FIFO order, atomically (one at a time and all the way through to completion or early termination).

This example shows a basic system boot event handler for the *Bluetooth* Smart modules. The example will start Bluetooth Smart advertisements as soon as the module is powered on or reset:

```
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)
    call gap_set_mode(gap_general_discoverable, gap_undirected_connectable)
end
```

BGScript event timeouts

The BGScript interpreter on the **Bluetooth Smart** modules has a default execution timeout value of **1000 steps**. This value does not correspond directly to lines of code or to a unit of time, but rather to executed opcode steps. If a particular event handler takes more than the specified number of steps, then the event will simply cut off immediately and the next event in the queue (if present) will begin. This value may be controlled with the **<script_timeout>** tag in the optional **config.xml** file as described in the **Bluetooth Smart Configuration Guide**. You can increase the timeout, or disable it entirely if desired.

4.4.2 if <expression> then [else] end if

Conditions can be tested with **if** clause. Any commands between **then** and **end if** will be executed if < **expression**> is true (or non-zero).

```
if x < 2 then
    x = 2
    y = y + 1
end if
```

If **else** is used and if the condition is success, then any commands between **then** and **else** will be executed. However if the condition fails then any commands between **else** and **end if** will be executed.

```
if x < 2 then
    x = 2
    y = y + 1
else
    y = y - 1
end if
```

4.4.3 while <expression> end while

Loops can be made using **while**. All commands on lines between **while** and **end while** will be executed while < **expression**> is true (or non-zero).

```
a = 0
while a < 10
    # will loop 10 times
    a = a + 1
end while
```

> **Timeouts with WHILE loops**
>
> It is important to remember that the default BGScript behavior on **Bluetooth Smart** modules involves a script timeout that will terminate an event early if the event handler code takes too long to execute. This can be very easy to do with long-running **while** loops. For example, a simple UART output command inside a **while** loop can run through approximately **75** iterations when the default script timeout value of 1000 is used. If you need a longer loop than this on your **Bluetooth Smart** BGScript application, please see the **Bluetooth Smart Configuration Guide** for information on how to increase or disable the script timeout.

4.4.4 call <command name>(<command parameters>..)[(response parameters)]

The **call** command is used to execute BGAPI commands and receive command responses. Command parameters can be given as expressions and response parameters are variable names where response values will be stored. Response parentheses and parameters can be omitted if the response is not needed by your application.

> **Note**
>
> Note that all response variables must be declared before use.

```
dim r

# write 2 bytes from tmp buffer index 0 to xgatt_hr attribute
# (response will be stored in variable "r")
call attributes_write(xgatt_hr, 2, tmp(0:2))(r)
```

The **call** command can also be used to execute user-defined procedures (functions). The syntax in this case is similar to executing a BGAPI command, except return values are not supported.

4.4.5 let <variable> = <expression>

Optional command to assign expression to variable.

```
let a = 1
let b = a + 2
```

4.4.6 sfloat(mantissa , exponent)

This function changes given mantissa and exponent in to a 16bit IEEE-11073 SFLOAT value which has base-10. Conversion is done using following algorithm:

	Exponent	Mantissa
Length	4 bits	12 bits
Type	2's-complement	2's-complement

Mathematically the number generated by **sfloat**() is calculated as **<mantissa> * 10^<exponent>**. The return value is a 2-byte uint8 array in the SFLOAT format. Below are some example parameters, and their resulting decimal sfloat values:

Mantissa	Exponent	Result (actual)
-105	-1	-10.5
100	0	100
320	3	320,000

Use the **sfloat()** function as follows, assuming that **buf** is already defined as a 2-byte uint8s array (or bigger):

```
buf(0:2) = sfloat(-105, -1)
```

The **buf** array will now contain the SFLOAT representation of **-10.5**.

Some reserved special purpose values:

- **NaN** (not a number)
  - exponent **0**
  - mantissa **0x007FF**
- **NRes** (not at this resolution)
  - exponent **0**
  - mantissa **0x00800**
- **Positive infinity**
  - exponent **0**
  - mantissa **0x007FE**
- **Negative infinity**
  - exponent **0**
  - mantissa **0x00802**
- Reserved for future use
  - exponent **0**
  - mantissa **0x00801**

4.4.7 float(mantissa , exponent)

Changes given mantissa and exponent in to 32-bit IEEE-11073 FLOAT value which has base-10. Conversion is done using following algorithm:

	Exponent	Mantissa
Length	8 bits	24 bits
Type	signed integer	signed integer

Some reserved special purpose values:

- **NaN** (not a number)
  - exponent **0**
  - mantissa **0x007FFFFF**
- **NRes** (not at this resolution)
  - exponent **0**
  - mantissa **0x00800000**
- **Positive infinity**
  - exponent **0**
  - mantissa **0x007FFFFE**
- **Negative infinity**
  - exponent **0**
  - mantissa **0x00800002**

- Reserved for future use
    - exponent **0**
    - mantissa **0x00800001**

4.4.8 memcpy(destination, source , length)

The **memcpy** function copies bytes from the source buffer to destination buffer. Destination and source should not overlap. Note that the buffer index notation only uses the **start** byte index, and should not also include the **size** portion, for example "**dst(start)**" instead of "**dst(start:size)**".

```
dim dst(3)
dim src(4)
memcpy(dst(0), src(1), 3)
```

4.4.9 memcmp(buffer1 , buffer2 , length)

The **memcmp** function compares ***buffer1*** and ***buffer2***, for the length defined with ***length***. The function returns 1 if the data is identical.

```
dim x(3)
dim y(4)
if memcmp(x(0), y(1), 3) then
    # do something
end if
```

4.4.10 memset(buffer , value , length)

This function fills ***buffer*** with with the data defined in ***value*** for the length defined with ***length***.

```
dim dst(4)
memset(dst(0), $30, 4)
```

4.5 Procedures

BGScript supports procedures which can be used to implementing subroutines. Procedures differ from functions used in other programming languages since they do not return a value and cannot be used expressions. Procedures are called using the **call** command just like other BGScript commands.

> **Note**
>
> On *Bluetooth* Smart products the amount of RAM allocated to the call stack is 100 bytes. If your program flow results in nested procedure calls (i.e. event handler calls **procedure1**, which calls **procedure2**, which calls **procedure3**) and/or you define procedures which use **uint8array** buffer parameters and then send long buffer values you can run out of stack space. The effect of this is typically that user-defined variables will have some or all of their data overwritten with temporary call stack data.
>
> In order to avoid this problem when using BGScript on *Bluetooth* Smart modules:
>
> - Try to keep your procedure calls as flat as possible (avoid nested calls)
> - Avoid the use of **uint8array** parameters whenever possible
> - When using uint8array parameters, avoid sending long buffers whenever possible

Procedures are defined by procedure command as shown below. Parameters are defined inside parentheses the same way as in event definition. Buffers are defined as last parameter and requires a pair of empty parentheses.

Example using procedures to print MAC address (Wifi modules only due to "endpoint_send" command and wifi-specific events):

MAC address output on Wifi modules

```
dim n, j

# print a nibble
procedure print_nibble(nibble)
    n = nibble
    if n < $a then
        n = n + $30
    else
        n = n + $37
    end if
    call endpoint_send(0, 1, n)
end

# print hex values
procedure print_hex(hex)
    call print_nibble(hex/16)
    call print_nibble(hex&$f)
end

# print MAC address
procedure print_mac(len, mac())
    j = 0
    while j < len
        call print_hex(mac(j:1))
        j = j + 1
        if j < 6 then
            call endpoint_send(0, 1, ":")
        end if
    end while
end

# boot event listener
event system_boot(major, minor, patch, build, bootloader, tcpip, hw)
    # read mac address
    call config_get_mac(0)
end

# MAC address read event listener
event config_mac_address(hw_interface, mac)
    # print the MAC address
    call print_mac(6, mac(0:6))
end
```

Example using single procedure to print arbitrary hex data in ASCII with optional separator (BLE modules only due to specific API commands):

MAC address output on BLE modules

```
# flexible procedure to display %02X byte arrays
dim hex_buf(3)  # [0,1] = ASCII hex representation, [2]=separator
dim hex_index   # byte array index
procedure print_hex_bytes(endpoint, separator, reverse, b_length, b_data())
    hex_buf(2:1) = separator
    hex_index = 0
    while hex_index < b_length
        if reverse = 0 then
            hex_buf(0:1) = (b_data(hex_index:1)/$10) + 48 + ((b_data(hex_index:1)/$10)/10*7)
            hex_buf(1:1) = (b_data(hex_index:1)&$f)  + 48 + ((b_data(hex_index:1)&$f )/10*7)
        else
            hex_buf(0:1) = (b_data(b_length - hex_index - 1:1)/$10) + 48 + ((b_data(b_length -
hex_index - 1:1)/$10)/10*7)
            hex_buf(1:1) = (b_data(b_length - hex_index - 1:1)&$f)  + 48 + ((b_data(b_length -
hex_index - 1:1)&$f )/10*7)
        end if
        if separator > 0 && hex_index < b_length - 1 then
            call system_endpoint_tx(endpoint, 3, hex_buf(0:3))
        else
            call system_endpoint_tx(endpoint, 2, hex_buf(0:2))
        end if
        hex_index = hex_index + 1
    end while
end

dim mac_addr(6) # MAC address container
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)
    # get module's MAC address (will be little-endian byte order)
    call system_address_get()(mac_addr(0:6))

    # output HEX representation (will look like "00:07:80:AA:BB:CC")
    # endpoint=UART1, separator=":", reverse=enabled, length=6, data="mac_addr" buffer
    call print_hex_bytes(system_endpoint_uart1, ":", 1, 6, mac_addr(0:6))
end
```

4.6 Using multiple script files

4.6.1 import

The **import** directive allows you to include other script files.

main.bgs

```
import "other.bgs"

event system_boot(major, minor, patch, build, bootloader, tcpip, hw)
    # wifi module has booted
end
```

4.6.2 export

By default all code and data are local to each script file. The **export** directive allows accessing variables and procedures from external files.

hex.bgs

```
export dim hex(16)
export procedure init_hex()
    hex(0:16) = "0123456789ABCDEF"
end
```

main.bgs

```
import "hex.bgs"
event system_boot(major, minor, patch, build, ll_version, protocol, hw)
    call init_hex()
end
```

5 BGScript Limitations

5.1 32-bit resolution

All operations in BGScript must be done using values that fit into 32 bits. The limitation affects for example long timer intervals. Since the soft timer has a 32.768kHz tick speed, it is possible in theory to have maximum interval of (2^32-1)/32768kHz = 36.4h. If longer timer periods are needed, incremental counters need to be used.

In particular with *Bluetooth* LE products, timer is 22 bits, so the maximum value with BLE112 is 2^22 = 4194304/32768Hz = 128 seconds, while with BLED112 USB dongle the maximum value is 2^22 = 4194304/32000Hz = 131 seconds

5.2 Declaration required before use

All data and procedures needs to be declared before usage.

5.3 DIM variable size

The largest size of a DIM variable is **255 bytes**. This limitation is in place to ensure that the small amount of RAM on the internal 8051 processor is not used entirely by user space variables and enough RAM is available for the Bluetooth Smart stack to maintain connections and transmission buffers.

5.4 Reading internal temperature meter disabled IO interrupts

Reading BLE112 internal temperature sensor value

```
call hardware_adc_read(14,3,0)
```

5.5 Writing data to an endpoint, which is not read

If the USB interface is enabled and the USB is connected to a USB host, there needs to be an application reading the data written to the USB. Otherwise the BGAPI messages will fill the buffers and cause the firmware to eventually freeze.

5.6 No interrupts on Port 2

Currently I/O interrupts cannot be enabled on any of the Port 2 pins. Interrupts are only only supported on Port 0 or Port 1.

5.7 Performance

BGScript has limited performance, which might prevent some applications to be implemented using BGscript. Typically, BGScript can execute commands/operations in the order of thousands per second.

6 Example BGscripts

This section contains useful BGScript examples.

6.1 Basics

This section contains very basic BGScript examples.

6.1.1 Catching system start-up

This example shows how to catch a system start-up. This event is the entry point to all BGScript code execution and can be compared to main() function in C.

System start-up

```
# Boot event listener
event system_boot(major,minor,patch,build,ll_version,protocol,hw)

   # System started, enable advertising and allow connections
   call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
   ...
end
```

6.1.2 Catching Bluetooth connection event

When a *Bluetooth* connection is received a **connection_status(...)** event is generated.

The example below shows how to enable advertisements to make the device connectable and how to catch a *Bluetooth* connection event.

Entering advertisement mode after disconnect

```
dim connected

# System start/boot listener
event system_boot(major,minor,patch,build,ll_version,protocol,hw)

   # Device is not connected yet
   connected = 0

   # Set advertisement interval to 20 to 30ms. Use all advertisement channels
   call gap_set_adv_parameters(32,48,7)

   # Start advertisement (generic discoverable, undirected connectable)
   call gap_set_mode(2,2)
end

# Connection event listener
event connection_status(connection, flags, address, address_type, conn_interval, timeout, latency,
bonding)

   # Device is connected.
   connected = 1
end
```

6.1.3 Catching Bluetooth disconnection event

When a *Bluetooth* connection is lost a **connection_disconnected** event is created.

Entering advertisement mode after disconnect

```
# Disconnection event
event connection_disconnected(handle, result)
   #connection disconnected, continue advertising
   call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
end
```

6.2 Hardware interfaces

This section contains basic examples to use hardware interfaces like I2C, SPI, AIO etc. from the BGScript.

6.2.1 ADC

ADC events can be cached with **hardware_adc_result(...)** event listener and the read operations on the other hand are called with **call hardware_adc_read(...)** function.

The example below shows how to read the internal temperature monitor and how to convert the value into Celsius

ADC read

```
dim celsius
dim offset
dim tmp(5)

# System boot event generated when the device is stared
event system_boot(major ,minor ,patch ,build ,ll_version ,protocol_version ,hw )

   # Call ADC read.
   # 14 = internal temperature sensor
   # 3 = 12 effective bits
   # 0 = Internal 1.24V reference
   call hardware_adc_read(14,3,0)
end

# ADC event listener
event hardware_adc_result(input,value)

   # ADC value is 12 MSB
   celsius = value / 16

   # Calculate temperature
   # ADC*V_ref/ADC_max * T_coeff + offset
   celsius = (10*celsius*1240/2047) * 10/45 + offset

   # set flags according to Health Thermometer specification
   # 0 = Temperature in Celsius
   tmp(0:1)=0

   # Convert to float
   tmp(1:4)=float(celsius, -1)
end
```


The example below shows how to read the internal battery monitor and how to convert the battery voltage level into percentage. A full example is included in the Bluetooth Smart SDK v.1.1 or newer.

ADC read

```
# This event listener listens for incoming ATT protocol read requests and when the battery
# attribute is read executes an ADC read when the battery value is requested.
event attributes_user_read_request(connection, handle, offset, maxsize)
  batconn_handle=connection
  #start measurement, read VDD/3, 9 effective bits
  call hardware_adc_read(15,3,0)
end


# This event listener catches the ADC result
event hardware_adc_result(input,value)
  #scale value to range 0-100
  #measurement range is 32768 = 1.24V*3 = 3.72V
  #new battery ADC measurement is 22198=2.52V
  #minimum battery voltage is 2.0 volts=2.0V/3.72V*32768= 17617
  #22198 - 17617 = 4580
  batresult=(value-17617)*100/4580

  #clip to 100%
  if batresult>100 then
    batresult=100
  end if
  if batresult<0 then
    batresult=0
  end if

  tmp(0:1)=batresult

  if batconn_handle<$ff then
    #if connection handle is valid
    call attributes_user_read_response(batconn_handle,0,1,tmp(0:1))
    batconn_handle=$ff
  end if
end
```

⚠ The above example requires the Bluetooth Smart SDK v.1.1 or newer in order to work properly. The code automatically turns off the external DC/DC (if used) when the battery reading is made and then re-enables it after the reading is complete.

6.2.2 I2C

BLE112 has a software implementation (bit-bang) of I2C which uses fixed pins. For communicating over the I2C bus following hardware setup is needed:

- **P1_6**: I2C data
- **P1_7**: I2C clock

Pull-ups must be enabled on both the pins.

BLE113 has a hardware implementation of I2C (only master-mode is supported). I2C pins are the following:

- **Pin 14**: I2C clock
- **Pin 15**: I2C data

> ⚠ No UART or SPI can be used in channel 1 with alternative 2 configuration when I2C is used.

I2C operations

```
# Reading 2 bytes from device which has I2C address of 128.
# I2C stop condition is sent after the transmission.
# Result 0 indicates successful read.
call hardware_i2c_read(128,1,2)(result,data_len,data)

# Write to address 128 one byte (0xf5).  I2C stop condition is sent after the transmission.
# written indicates how many bytes were successfully written.
call hardware_i2c_write(128,1,1,"\xf5")(written)
```

6.2.3 IO

IO wake-up

When the device has no active tasks or timers running it can go to power mode 3 (PM3), which is the lower power mode consuming about 400nA. PM3 power save mode however requires an external wake-up using an IO pin.

The example here shows and IO interrupt can be used to wake up the device and start advertisements for 5 seconds and then go back to PM3.

Enabling and catching IO interrupts

```
# Boot event listener
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)

   # Enable IO interrupts from PORT 0 PINs P0_0 and P0_1 on rising edge
   call hardware_io_port_config_irq(0,$3,0)
end

# HW interrupt listener
event hardware_io_port_status(delta, port, irq, state)

   # Configure advertisement parameters
   call gap_set_adv_parameters(40, 40, 7)
   # Start advertisements
   call gap_set_mode(2, 2)
   # Start a 5 second, one stop timer
   call hardware_set_soft_timer($27FFB, 0 ,1)
end

# Timer event listener
event hardware_soft_timer(handle)

   #Stop advertisements and allow the device to go to PM3
   call gap_set_mode(0, 0)
end
```

To enable PM3 and configure the wake-up pin the following configurations need to be used in the **hardware.xml** file.

<hardware>
<sleeposc enable="true" ppm="30" />
<wakeup_pin enable="true" port="0" pin="0" />
<usb enable="false" endpoint="none" /> <txpower power="15" bias="5" />

<port index="0" tristatemask="0" pull="down" />
<script enable="true" />
<slow_clock enable="true" />
</hardware>

Writing IO status

The example below shows how to write the P0_0 status.

Enabling and catching IO interrupts

```
# Boot event listener
event system_boot(major ,minor ,patch ,build ,ll_version ,protocol_version ,hw )

   # Configure the P0_0 as output
   call hardware_io_port_config_direction(0, 1)
   # Enable P0_0 pin
   call hardware_io_port_write(0, 1, 1)

   # Start a 5 second, one stop timer
   call hardware_set_soft_timer($27FFB, 0 ,1)
end

# Timer event listener
event hardware_soft_timer(handle)
   # When timer expires disable P0_0 pin
   call hardware_io_port_write(0, 1, 0)
end
```

6.2.4 SPI

Writing SPI

SPI interface can be used as a peripheral interface for example to connect to sensors like accelerometers or simple displays. The example below shows how to write data to SPI interface.

Writing to SPI

```
# Boot event listener
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)

   # Writing 5 bytes to SPI
   call hardware_spi_transfer(0,5,"\x01\x02\x03\x04\x05")

   # Writeing a "Hello world\!" string to SPI
   call hardware_spi_transfer(0,12,"Hello world\!")
end
```

The following configurations need to be in the **hardware.xml** to enable the SPI interface and BGScript execution.

<hardware>
...
<usart channel="0" mode="spi_master" alternate="2" polarity="positive" phase="1" endianness="msb" baud="57600" endpoint="none" />
<script enable="true" />
</hardware>

Reading SPI

The example below shows how to read data from SPI interface. SPI interface returns you as many bytes as you write to it. In this example two (2) bytes are written to SPI interface and the return values return the read result. The read data is stored in the **tmp**-array and it has length on two (2) bytes.

Reading SPI interface

```
dim tmp(10)
dim result
dim channel
dim tlen

call hardware_spi_transfer(0,2,"\x01\x02")(result,channel,tlen,tmp(0))
```

⚠ The following configurations need to be in the hardware.xml to enable the SPI interface and BGScript execution.

<hardware>
...
<usart channel="0" mode="spi_master" alternate="2" polarity="positive" phase="1" endianness="msb" baud="57600" endpoint="none" />
<script enable="true" />
</hardware>

6.2.5 Generating PWM signals

In order to generate PWM signals output compare mode needs to be used. PWM output signals can be generated using the **timer modulo mode** and when **channels 1** and **2** are in **output compare mode 6 or 7**.

For detailed instructions about PWM please refer to chapter **9.8 Output Compare Mode** in CC2540 user guide.

In order to generate a 4 channel PWM signal the following example can be used.

A 4 channel PWM signal

```
# Boot event listener
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)
    call hardware_timer_comparator(1, 0, 6, 32000)
    call hardware_timer_comparator(1, 1, 6, 16000)
    call hardware_timer_comparator(1, 2, 6, 10000)
    call hardware_timer_comparator(1, 3, 6, 8000)
    call hardware_timer_comparator(1, 4, 6, 4000)
end
```

The example uses Timer 1 in alternate 2 configuration with four (4) PWM channels in pins p1.1, p1.0, p0.7 and p0.6

The following configurations need to be in the **hardware.xml** to enable the timer and BGScript execution.

<hardware>
...
<timer index =" 1 " enabled_channels =" 0x1f " divisor =" 0 " mode =" 2 " alternate =" 2 " />
</hardware>

Notice that PWMs do not work when the device is in a sleep mode.

6.3 Timers

This section describes how to use timers with BGscript.

6.3.1 Continuous timer generated interrupt

This example shows how to generate continuous timer generated interrupts

Enabling timer generated interrupts

```
# Boot event listener
event system_boot(major ,minor ,patch ,build ,ll_version ,protocol_version ,hw )
   ...
   #Set timer to generate event every 1s
   call hardware_set_soft_timer(32768, 1, 0)
   ...
end


#Timer event listener
event hardware_soft_timer(handle)
   #Code that you want to execute once per 1s
   ...
end
```

Even with a soft timer running the module can enter sleep mode 2, in which power consumption is about 1µA. Sleep mode 3 is entered only if there are no timers running and the module is not having any scheduled radio activity.

One active timer

There can only be one timer running at the same time. Please stop the currently running timer by issuing **call hardware_set_soft_timer(0, {handle}, {singleshot}**) before launching the next one.

6.3.2 Single timer generated interrupt

The 2nd example shows how to set a timer, which is called only once. This is useful, when some action needs to be implemented only once, like the change of advertisement interval in Proximity profile.

In this example in the beginning the device advertises quickly, but after 30 seconds the advertisement interval is reduced, in order to save battery.

Using timer once

```
# Boot event listener
event system_boot(major,minor,patch,build,ll_version,protocol,hw)

   # Set advertisement parameters according to the Proximity profile
   # Min interval 20ms, max interval 30ms, use all 3 channels
   call gap_set_adv_parameters(32, 48, 7)

   # Enabled advertisement
   # Limited discovery, Undirected connectable
   call gap_set_mode(1, 2)

   # Start timer
   # single shot, 30 secods, timer handle = 1
   call hardware_set_soft_timer($F0000, 1, 1)
end

# Timer event listener
event hardware_soft_timer(handle)

   # run the code only if timer handle is 1
   if handle = 1 then
      # Stop advertisement
      call gap_set_mode(0, 0)

      #Reconfigure parameters
      # Min interval 1000ms, max interval 2500ms, use all 3 channels
      call gap_set_adv_parameters(1600, 4000, 7)

      # Enabled advertisement
      # Limited discovery, Undirected connectable
      call gap_set_mode(1, 2)
   end if

end
```

6.4 USB and UART endpoints

This section describes the usage of endpoints, which can be used to send or receive data from interfaces like UART or USB.

6.4.1 UART endpoint

The example shows how to send data to USART1 endpoint from BGScript.

Writing to USB endpoint

```
# System start/boot listener
event system_boot(major,minor,patch,build,ll_version,protocol,hw)

  # Start continuous timer with 1 second interval. Handle ID 1
  # 1 second = $8000 (32.768kHz crystal)
  call hardware_set_soft_timer($8000, 1, 0)
end

# Timer event(s) listener
event hardware_soft_timer(handle)

  # 1 second timer expired
  if handle = 1 then
    call system_endpoint_tx(5, 14, "TIMER EXPIRED\n")
  end if
end
```

The following configurations need to be in the **hardware.xml** to enable the UART interface and allow BGscript to access it.

<?xml version="1.0" encoding="UTF-8" ?>

<hardware>
...
<usart channel="1" alternate="1" baud="115200" endpoint="none" />
<script enable="true" />
</hardware>

6.4.2 USB endpoint

The example shows how to send data to USB endpoint from BGScript.

Writing to USB endpoint

```
# System start/boot listener
event system_boot(major,minor,patch,build,ll_version,protocol,hw)

  # Start continuous timer with 1 second interval. Handle ID 1
  # 1 second = $8000 (32.768kHz crystal)
  call hardware_set_soft_timer($8000, 1, 0)
end

# Timer event(s) listener
event hardware_soft_timer(handle)

  # 1 second timer expired
  if handle = 1 then
    call system_endpoint_tx(3, 14, "TIMER EXPIRED\n")
  end if
end
```

The following configurations need to be in the **hardware.xml** to enable the USB interface and allow BGscript to access it.

<?xml version="1.0" encoding="UTF-8" ?>

<hardware>
...
<usb enable="true" endpoint="none" />
<script enable="true" />
</hardware>

6.5 Attribute Protocol (ATT)

This section contains BGscript examples related to Attribute Protocol (ATT) events.

6.5.1 Catching attribute write event

The example shows to to catch an event when remote devices writes an attribute over a Bluetooth connection. A simple FindMe example is used where the remote device writes a single value to the local GATT database indicating the alert level.

Catching an attribute write

```
# Listen for GATT write events
event attributes_value(connection, reason, handle, offset, value_len, value)
   # Read the value and enable corresponding alert
   level=value(0:1)
   if level=0 then
      # TODO: Execute an action corresponding "No alert" status.
   end if
   if level=1 then
      # TODO: Execute an action corresponding "Mild alert" status.
   end if
   if level=2 then
      # TODO: Execute an action corresponding "High alert" status.
   end if
end
```

6.6 Generic Attribute Profile (GATT)

This section shows examples how to manager the local GATT database.

6.6.1 Changing device name

The example below shows how to change the device name using BGScript.

In this example we use the following GATT database:

gatt.xml

```
<service uuid="1800">
      <description>Generic Access Profile</description>

      <characteristic uuid="2a00" id="xgatt_name">
        <properties read="true"/>
        <value>01020304050607</value>
      </characteristic>

      <characteristic uuid="2a01">
        <properties read="true" const="true" />
        <value type="hex">4142</value>
      </characteristic>
    </service>
```

To write a new value into the characteristic defined in the **gatt.xm**l following code needs to be used. Please note that the **id** must be the same as in the **gatt.xml**.

script.bgs

```
# Generate Friendly name in ASCII
name(0:1)=$42
name(1:1)=$47
name(2:1)=$53
name(3:1)=$63
name(4:1)=$72
name(5:1)=$69
name(6:1)=$70
name(7:1)=$74

#Write name to local GATT
call attributes_write(xgatt_name, 0, 7, name(0:7))
```

6.6.2 Writing to local GATT database

To write to the local GATT database you first need to define a characteristic under a service in your GATT database (**gatt.xml**). You also need to assign an **id** parameter for the characteristic, which can then be used in BGScript to write the value.

In this example we use the following GATT database:

gatt.xml

```
<service uuid="1809">
    <description>Health Thermometer Service</description>

    <characteristic uuid="2a1c" id="xgatt_temperature_celsius">
        <description>Celsius temperature</description>
        <properties indicate="true"/>
        <value type="hex">0000000000</value>
    </characteristic>
</service>
```

To write a new value into the characteristic defined in the **gatt.xml** following code needs to be used. Please note that the **id** must be the same as in the **gatt.xml**.

script.bgs

```
#write 5 bytes from tmp array to attribute with offset 0
call attributes_write(xgatt_temperature_celsius,0,5,tmp(0:5))
```

6.7 PS store

These examples show how to read and write PS-keys.

6.7.1 Writing a PS keys

The example shows how to write an attribute written by a remote *Bluetooth* device into PS store.

Writing to PS store

```
# Check if remote device writes a value to the GATT and write it to a PS key 0x8000
# Catch an attribute write
event attributes_value(connection, reason, handle, offset, value_len, value_data)

    # Check if handle value 1 is written
    if handle = 1
        # Write attribute value to PS-store
        call flash_ps_save($8000, value_len, value_data(0:value_len))
    end if
end
```

PS keys from 8000 to 807F can be used for persistent storage of user data.
Each key can store up to 32 Bytes.

6.7.2 Reading a PS keys

The example shows how to read a value from the local PS store and write it to GATT database.

Reading PS store

```
#Initialize a GATT value from a PS key, which is 2 bytes long
call flash_ps_load($8000)(result, len1, data1(0:2))


# Write the PS value to handle with ID "xgatt_PS_value"
call attributes_write(xgatt_PS_value, 0, len1, data1(0:len1))
```

PS keys from 8000 to 807F can be used for persistent storage of user data.
Each key can store up to 32 Bytes.

6.8 Advanced scripting examples

This section shoes more advanced scripting examples where several functions are made.

6.8.1 Catching IO events and exposing them in GATT

This example shows hot to catch I/O events and exposing them via a custom service in GATT data base.

he example service look like the one below and the I/O characteristic has ***read*** and ***notify*** properties

gatt.xml

```
<service uuid="00431c4a-a7a4-428b-a96d-d92d43c8c7cf">
        <description>Bluegiga IO service</description>
        <characteristic uuid="f1b41cde-dbf5-4acf-8679-ecb8b4dca6fe" id="xgatt_io">
            <properties read="true" notify="true"/>
            <value type="hex" length="1"></value>
        </characteristic>
    </service>
```

In order to catch the I/O events and write them to GATT database the following event handled is used in BGScript code.

script.bgs

```
#HW interrupt listener
event hardware_io_port_status(delta, port, irq, state)

 # Write I/O status to GATT
 call attributes_write(xgatt_io,0,1,irq)
end
```

On DKBLE112 development kit there are buttons in I/O pins P0_0 and P0_1 and in order for this example to work with DKBLE112 the following configuration is needed in hardware.xml.

hardware.xml

```
<port index="0" pull="down" />
```

6.9 Bluegiga Development Kit Specific Examples

This section contains examples specific to the Bluegiga BLE development kits.

6.9.1 Display initialization

The example below shows how to initialize the display in the BLE development kit and and how to write data to it.

The supported commands can be found from the displays data sheet as well the initialization sequence.

DKBLE112 display initialization

```
# Boot event listener
event system_boot(major,minor,patch,build,ll_version,protocol,hw)

    # Set display to command mode
    call hardware_io_port_write(1,$3,$1)
    call hardware_io_port_config_direction(1,$7)

    # Initialize the display (see NHDC0216CZFSWFBW3V3 data sheet)
    call hardware_spi_transfer(0,11,"\x30\x30\x30\x39\x14\x56\x6d\x70\x0c\x06\x01")


    # Set display to data mode
    # Write "Hello world\!" to the display.
    call hardware_io_port_write(1,$3,$3)
    call hardware_spi_transfer(0,12,"Hello world\!")

end
```

SPI configuration in ***hardware.xml***
<usart channel="0" mode="spi_master" alternate="2" polarity="positive" phase="1" endianness="msb" baud="57600" endpoint="none" />

6.9.2 FindMe demo

The example script implements a simple FindMe profile device. The alert status is displayed on the BLE development kit's display when remote device changes the status.

SPI configuration in ***hardware.xml***
<usart channel="0" mode="spi_master" alternate="2" polarity="positive" phase="1" endianness="msb" baud="57600" endpoint="none" />

DKBLE112 FindMe Target

```
# Boot event listener
event system_boot(major,minor,patch,build,ll_version,protocol,hw)

    # Put display into command mode
    call hardware_io_port_write(1,$3,$1)
    call hardware_io_port_config_direction(1,$7)

    # Configure Display
    call hardware_spi_transfer(0,11,"\x30\x30\x30\x39\x14\x56\x6d\x70\x0c\x06\x01")

    # Put display into data mode and write
    call hardware_io_port_write(1,$3,$3)
    call hardware_spi_transfer(0,12,"Find Me Demo")

    # Set advertisement parameters according to the Proximity profile. Min interval 1000ms, max
interval 2000ms, use all 3 channels
    call gap_set_adv_parameters(1600, 3200, 7)

    # Start advertisement and enable pairing mode
    call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
    call sm_set_bondable_mode(1)
end

# Listen for GATT write events
event attributes_value(connection, reason, handle, value_len, value)

    # Put display to command mode and move cursor to position 40
    call hardware_io_port_write(1,$3,$1)
    call hardware_spi_transfer(0,1,"\xc0")

    #display to data mode
    call hardware_io_port_write(1,$3,$3)

    # Read value and enable corresponding alert
    level=value(0:1)
    if level=0 then
       call hardware_spi_transfer(0,10,"No Alert  ")
    end if
    if level=1 then
        call hardware_spi_transfer(0,10,"Mild Alert")
    end if
    if level=2 then
        call hardware_spi_transfer(0,10,"High Alert")
    end if
end

# Disconnection event listener
event connection_disconnected(handle,result)
    # Restart advertisement
    call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
end
```

6.9.3 Temperature and battery readings to display

The example below shows how to initialize the display in the BLE development kit and and how to write temperature and battery (using potentiometer) readings into it.

The supported commands can be found from the displays data sheet as well the initialization sequence.

> SPI configuration in ***hardware.xml***
> **<usart channel="0" mode="spi_master" alternate="2" polarity="positive" phase="1" endianness="msb" baud="57600" endpoint="none" />**

DKBLE112 display, battery and temperature sensors

```
dim string(3)
dim milliv
dim tmp(4)
dim offset
dim celsius

# Boot event listener
event system_boot(major,minor,patch,build,ll_version,protocol,hw)
    # Initialize the display (see NHD-C0216CZ-FSW-FBW-3V3 data sheet)
    call hardware_io_port_write(1,$7,$1)
    call hardware_io_port_config_direction(1,$7)
    call hardware_spi_transfer(0,11,"\x30\x30\x30\x39\x14\x56\x6d\x70\x0c\x06\x01")
    call hardware_io_port_write(1,$7,$3)

    # Write "Batt.: " to the display.
    call hardware_spi_transfer(0,7,"Batt.: ")

    # Change display data address
    call hardware_io_port_write(1,$7,$1)
    call hardware_spi_transfer(0,1,"\xc0")

    # Write "Temp.: " to the displays 2nd line
    call hardware_io_port_write(1,$7,$3)
    call hardware_spi_transfer(0,7,"Temp.: ")

    # Start timer @ ~2sec interval
    call hardware_set_soft_timer($ffff, 0 ,0)
end

# Timer event listener
event hardware_soft_timer(handle)
    #read potentiometer for battery
    call hardware_adc_read(6,1,2)
    #read internal temperature
    call hardware_adc_read(14,3,0)
end
```

DKBLE112 display, battery and temperature sensors (CONTINUED)

```
#ADC event listener
event hardware_adc_result(input,value)

    # Received battery reading
    if (input = 6) then
        #Convert HEX to STRING
        milliv=value/11+8
        tmp(0:1) = (milliv/1000) + (milliv / 10000*-10) + 48
        tmp(1:1) = (milliv/100) + (milliv / 1000*-10) + 48
        tmp(2:1) = (milliv/10) + (milliv / 100*-10) + 48
        tmp(3:1) = (milliv) + (milliv / 10*-10) + 48

        # Change display data address
        call hardware_io_port_write(1,$7,$1)
        call hardware_spi_transfer(0,1,"\x87")

        # Write battery value
        call hardware_io_port_write(1,$7,$3)
        call hardware_spi_transfer(0,4,tmp(0:4))
        call hardware_spi_transfer(0,3," mV")
    end if

    # Received temperature reading
    if (input = 14) then
        offset=-1490

        # ADC value is 12 MSB
        celsius = value / 16
        # Calculate temperature
        # ADC*V_ref/ADC_max / T_coeff + offset
        celsius = (10*celsius*1150/2047) * 10/45 + offset

        #Convert HEX to STRING
        string(0:1) = (celsius / 100) + 48
        string(1:1) = (celsius / 10) + (celsius / -100 * 10) + 48
        string(2:1) = celsius + (celsius / 10 * -10) + 48

        # Change display data address
        call hardware_io_port_write(1,$7,$1)
        call hardware_spi_transfer(0,1,"\xc7")

        # Write temperature value
        call hardware_io_port_write(1,$7,$3)
        call hardware_spi_transfer(0,2,string(0:2))
        call hardware_spi_transfer(0,1,".")
        call hardware_spi_transfer(0,1,string(2:1))
        call hardware_spi_transfer(0,1,"\xf2")
        call hardware_spi_transfer(0,1,"C")
    end if
end
```

6.10 BGScript tricks

6.10.1 HEX to ASCII

Printing local BT address on the display in DKBLE112

```
dim t(12)
dim addr(6)
event system_boot(major,minor,patch,build,ll_version,protocol,hw)
    call hardware_io_port_write(1,$7,$1)
    call hardware_io_port_config_direction(1,$7)

    #Initialize the display
    call hardware_spi_transfer(0,11,"\x30\x30\x30\x39\x14\x56\x6d\x70\x0c\x06\x01")
    call hardware_io_port_write(1,$7,$3)

    #Get local BT address
    call system_address_get( )(addr(0:6))

    t(0:1) = (addr(5:1)/$10) + 48 + ((addr(5:1)/$10)/10*7)
    t(1:1) = (addr(5:1)&$f)  + 48 + ((addr(5:1)&$f )/10*7)
    t(2:1) = (addr(4:1)/$10) + 48 + ((addr(4:1)/$10)/10*7)
    t(3:1) = (addr(4:1)&$f)  + 48 + ((addr(4:1)&$f )/10*7)
    t(4:1) = (addr(3:1)/$10) + 48 + ((addr(3:1)/$10)/10*7)
    t(5:1) = (addr(3:1)&$f)  + 48 + ((addr(3:1)&$f )/10*7)
    t(6:1) = (addr(2:1)/$10) + 48 + ((addr(2:1)/$10)/10*7)
    t(7:1) = (addr(2:1)&$f)  + 48 + ((addr(2:1)&$f )/10*7)
    t(8:1) = (addr(1:1)/$10) + 48 + ((addr(1:1)/$10)/10*7)
    t(9:1) = (addr(1:1)&$f)  + 48 + ((addr(1:1)&$f )/10*7)
    t(10:1) = (addr(0:1)/$10)+ 48 + ((addr(0:1)/$10)/10*7)
    t(11:1) = (addr(0:1)&$f) + 48 + ((addr(0:1)&$f )/10*7)

    call hardware_spi_transfer(0,12,t(0:12))
end
```

6.10.2 UINT to ASCII

To display sensor readings in the display, integer values must be converted to ASCII. Currently there is no build-in function for doing this in the BGScript, but the following function can be used to convert integers to ASCII:

a = (rh / 100)

b = (rh / 10) + (rh / -100 * 10)

c = rh + (rh / 10 * -10)

And as BGScript code:

Converting 3 digit interger to ASCII

```
dim data
dim string(3)

string(0:1) = (data / 100) + 48
string(1:1) = (data / 10) + (data / -100 * 10) + 48
string(2:1) = data + (data / 10 * -10) + 48
```

To present the string in the display of the evaluation kit please refer to DKBLE112 display initialization -- BGScript

7 BGScript editors

This section contains different tips and tricks for editors and IDEs.

7.1 Notepad ++

Notepad++ is very flexible text editor for programming purposes. Application and documentation can be downloaded from http://notepad-plus-plus.org/.

7.1.1 Syntax highlight for BGScript

Notepad++ doesn't currently contain syntax highlighting for BGScript by default. You can however download syntax highlighting rules defined by Bluegiga.

Installing the BGScript syntax highlight rules into Notepad++ is easy:

1. Download the syntax highlighting rules from https://www.bluegiga.com/en-US/products/bluetooth-4.0-modules/ble112-bluetooth--smart-module/docume (from the PC Tools section)
2. Import the highlighting rules to Notepad++ : ***View->User-Defined Dialogue->Import.***
3. When editing the code, enable syntax highlighting from : ***Language -> BGscript***

> **Notepad ++: How to create your own Syntax Highlighting scheme**
>
> http://sourceforge.net/apps/mediawiki/notepad-plus/index.php?title=User_Defined_Languages

Contact information

Sales:	sales@bluegiga.com
Technical support:	http://www.bluegiga.com/support/
Orders:	orders@bluegiga.com
WWW:	http://www.bluegiga.com
Head Office / Finland:	Phone: +358-9-4355 060 Fax: +358-9-4355 0660 Sinikalliontie 5 A 02630 ESPOO FINLAND
Head address / Finland:	P.O. Box 120 02631 ESPOO FINLAND
Sales Office / USA:	Phone: +1 770 291 2181 Fax: +1 770 291 2183 Bluegiga Technologies, Inc. 3235 Satellite Boulevard, Building 400, Suite 300 Duluth, GA, 30096, USA
Sales Office / Hong-Kong:	Phone: +852 3182 7321 Fax: +852 3972 5777 Bluegiga Technologies, Inc. Unit 10-18, 32/F, Tower 1, Millennium City 1, 388 Kwun Tong Road, Kwun Tong, Kowloon, Hong Kong