BGSCRIPT FOR WI-FI

DEVELOPER GUIDE

Tuesday, 2 June 2015

Version 1.9

Table of Contents

1 Version History

Version	
0.1	Wi-Fi Script documentation compatible with SW version v.0.3.0 (Build 25).
0.2	Added IO, I2C examples and compatible with SW version v.0.4
1.0	Compliant with SW v.1.0.
1.2	Procedure support added to BGScript - Wi-Fi SDK 1.2 beta
1.3	Improved examples (I/O, RTC and Wi-Fi)
1.4	Added firmware compilation and installation instructions
1.5	Updated to reflect v.1.2 software
1.6	Improved WPS example
1.7	Updates for Wi-Fi software v. 1.2.2 and editorial changes Added instructions for splitting BGScript into multiple files through IMPORT and EXPORT directives
1.8	Improvements to BGScript syntax description
1.9	Compliant with 1.3 SW version. The following changes introduced: • Constant strings introduced • Return build-in function added • Clarifications and editorial changes made

2 Introduction to BGScript

BGScript is a simple BASIC-like scripting language intended for simple application programming. BGScript applications can be used to to automate simple application functionality such as open connections, listen for GPIO interrupts and read or write data from interfaces like UART, SPI, I2C or USB. BGScript can also be used for simple data processing such as arithmetic operations, bitwise operations, buffers and data comparison. BGScript scripting language allows complete applications to be implemented without the need for an external host controller (MCU), since the BGScript can typically be executed directly on the Bluegiga wireless module.

There are numerous benefits to building a system with the BGScript scripting language instead of using an external host. Without the host controller, the device size can be reduced as well as the electronic bill-of-materials. The power consumption is lower allowing the end product to either operate longer with the same battery or even to reduce the size and cost of the battery. As the overall complexity of the product decreases, development time and risks are also decreased. BGScript can also be developed with free of charge tools and there is no need to invest in expensive IDEs and debuggers.

Because the BGScript is an interpreted language, its limitations are typically reached in applications in which fast data processing or data collection from peripherals is needed. If your application needs to e.g. read an accelerometer thousands or tens of thousands of times per second and process the data, BGScript applications might not give the desired performance and instead an external host should be used.

The BGScript sits on top of the BGAPI and has access to exactly the same APIs as are available to an external host.
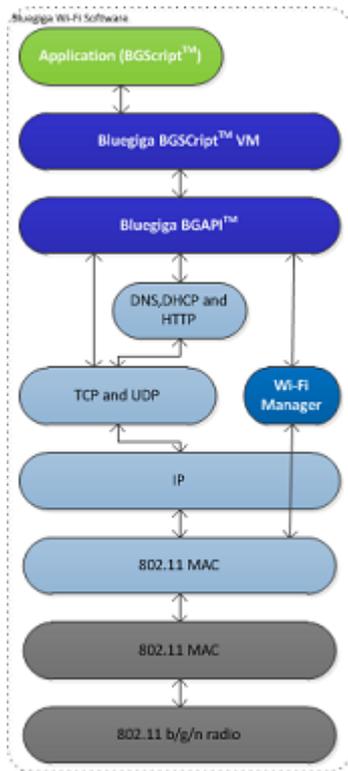
Figure: BGScript System Architecture

2.1 BGScript Example

The example BGScript below shows how to enable the Wi-Fi radio after system start-up and how to initiate a connection procedure to an Access Point.

```
#define variable for call result and out endpoint we use for sending data
dim result,out_ep
event system_boot(major,minor,patch,build,bootloader,tcpip,hw)
#system has booted, start Wi-Fi subsystem by using BGAPI command
    call sme_wifi_on()
end
event sme_wifi_is_on(result)
#Wi-Fi has started, connect to Access Point by using SSID
    call sme_connect_ssid(5,"nakki")
end
...
```

3 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 successive 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 commands and events are specific to each technology.

Below is a conceptual code example of a simple BGScript based 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
```

3.1 Comments

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

```
x = 1 # comment
```

3.2 Variables and Values

3.2.1 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
```

3.2.2 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
```

3.2.3 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
```

3.2.4 Constant Values

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
```

3.2.5 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.

Similar to 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
```

Buffer index and size are optional and if left empty default values are used. Default value for index is 0 and default value for size is maximum size of buffer.

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
```

3.2.6 Strings

Buffers can be initialized 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.

3.2.7 Constant Strings

Constant strings must be defined before use.

```
const str() = "test string"
```

And can be used in place of buffers. Note that in following example index and size of buffer is left as default values.

```
call endpoint_send(11, str(:))
```

3.3 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:

Operation	Symbol
AND	&
OR		
XOR	^
Shift left	<<
Shift right	>>

The following **logical operators** are supported:

Operation	Symbol
AND	&&
OR	||

3.4 Commands

3.4.1 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 (Virtual Machine) 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
```

3.4.2 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
```

Note! BGScript uses **C language operator precedence**. This means that bitwise **&** and **|** operators have lower precedence than the comparison operator, and so comparisons are handled first if present in the same expression. This is important to know when creating more complex conditional statements. It is a good idea to include explicit parentheses around expressions which you need to be evaluated first.

```
if $0f & $f0 = $f0 then
    # will match because ($f0 = $f0) is true, and then ($0f & true) is true
end if

if ($0f & $f0) = $f0 then
    # will NOT match because ($0f & $f0) is $00, and $00 != $f0
end if
```

3.4.3 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
```

3.4.4 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)
```

If buffer or string is needed as parameter, then the buffer size is set in previous parameter.

```
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)
    call endpoint_send(0,13,"Hello, world!")
end
```

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.

3.4.5 let <variable> = <expression>

Optional command to assign an expression to a variable.

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

3.4.6 return

This command returns from an event or a procedure.

```
event hardware_io_port_status(delta, port, irq, state)
    if state = 0
        return #returns from event
    end if
    tmp(0:1) = 2
    tmp(1:1) = 60 * 32768 / delta
    call attributes_write(xgatt_hr, 2, tmp(0:2))
end
```

3.4.7 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**

3.4.8 float(mantissa , exponent)

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

3.4.9 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)
```

3.4.10 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
```

3.4.11 memset(buffer , value , length)

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

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

3.5 Procedures

BGScript supports procedures which can be used to implement 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.

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 Wi-Fi 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:

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
```

3.6 Using multiple script files

3.6.1 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
```

3.6.2 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
```

4 BGScript Limitations

4.1 32-bit resolution

All values used in BGScript must fit into 32 bits, the limitation affects for example very long timer intervals.

4.2 Performance

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

5 Examples

This section contains examples on how to perform various actions using BGScript.

5.1 Basics

This section contains very basic Wi-Fi BGScript examples.

5.1.1 Catching system startup

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,bootloader_version,ipstack_version,hw)
    # System started - start Wi-Fi radio
    call sme_wifi_on()
end
```

5.1.2 Performing a System Reset

This example shows how to perform a system reset and restart the firmware.

System start-up

```
# Something went wrong and system needs to be reset
call system_reset(0)
```

5.1.3 Handling command responses

All BGScript API commands follow either a command-response or a command-response-event pattern. Typically a command response contains a status code indicating whether the command was executed successfully or not. Most of the BGScript examples in this document omit checking of response status codes in order to keep the examples short and to improve readability. **It is however vital that the actual implementation validates all responses**. If a command responds with an error, events tied to the command may or may not occur at all.

> ⚠ In BGScript, command response parameters refer to variables where the values will be copied. References to buffers are given using the BUFFER(<*expression*>:<*size*>) notation. Buffers must contain enough space for the response data, otherwise a buffer overflow will occur.

Handling a command response

```
dim result
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Try to initiate a scan without enabling Wi-Fi. In this case the command response contains
  # only one parameter which will be copied to variable "result".
  call sme_start_scan(0, 0, 0)(result)
  if result != 0
    # Scan command failed. Handle the failure gracefully.
  end if
end
# Event received when a scan has been completed.
event sme_scanned(status)
  # This event will never be triggered. If the implementation didn't check the command response,
  # it would get stuck waiting for this event to occur.
end
```

Handling a command response containing a buffer

```
dim result
dim value_len
dim value_data(4)
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Notice the usage of variable "value_len" in "value_data" reference.
  call flash_ps_load(FLASH_PS_KEY_MODULE_SERVICE)(result, value_len, value_data(0:value_len))
end
```

When using command-response-event commands, special care needs to be paid making sure commands are not executed in parallel, i.e. calling a new command while still waiting for an event from the previous one is not recommended. **Parallel commands may have unintended consequences and may or may not work.** The exception to this rule is a stop command to the corresponding start command. The stop command is always safe to call while the start command is still ongoing.

```
# This is example demonstrates how parallel commands may have unintended consequences.
# DO NOT FOLLOW THIS EXAMPLE!
dim device_mac(6)
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
   # Retrieve the device MAC address. This call will trigger config_mac_address() event.
   call config_get_mac(0)
   # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
   call sme_wifi_on()
   # We are now attempting to execute two commands in parallel.
end
# Event received when the device MAC address has been retrieved.
event config_mac_address(hw_interface, mac)
   # In this particular scenario this event is correctly triggered.
   device_mac(0:6) = mac(0:6)
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
   # In this particular scenario this event will never occur.
end
```

5.2 Wi-Fi

This section of the manual shows simple examples how to handle Wi-Fi related events with BGScript.

5.2.1 Catching Wi-Fi connection event

When a Wi-Fi Access Point connection is established a **sme_connected(...)** event is generated.

The example below shows how to detect when the module has established successfully a Wi-Fi connection.

```
dim connected
# AP connection event listener
event sme_connected(status, hw_interface, bssid)
    # if connection status = connected and HW interface is Wi-Fi
    if (status = 0 && hw_interface = 0) then
       # AP connection established
       connected = 1
    end if
end
```

5.2.2 Catching Wi-Fi disconnection event

When a Wi-Fi Access Point connection is lost an **sme_disconnected** event is created.

Entering advertisement mode after disconnect

```
dim connected
# Disconnection event
event sme_disconnected(reason, hw_interface)
    # check if Wi-Fi interface caused the event
    if (hw_interface = 0) then
        #AP connection disconnected, turn off Wi-Fi radio
        connected = 0
        call sme_wifi_off()
end
```

5.2.3 Catching a failed Wi-Fi connection event

Sometimes the Wi-Fi connections to an Access Point fail and an event will be generated allowing one to catch a failed connection. This is indicated with **sme_connect_failed** event.

Entering advertisement mode after disconnect

```
# Event received after a connection attempt fails.
event sme_connect_failed(reason, hw_interface)
  # increase re-connection counter by one
  reconnect_count = reconnect_count + 1

  # check if MAX number of re-connection attempts have been reached
  if(reconnect_count < MAX_RECONNECTS)
    # Try to reconnect
    call call sme_connect_ssid(...)
  else
    # Do something else
  end if
end
```

5.2.4 Performing a Wi-Fi scan

A Wi-Fi scan is initiated by calling **sme_start_scan(...)** command. The command can only be issued when Wi-Fi has been enabled and the device is not connected to a Wi-Fi network or operating as an Access Point. Scan results are returned as events where each event represents an Access Point. Since it's possible that a Wi-Fi network may consist of many Access Points, the results may contain events where the network name is identical but BSSID is different. The results are returned in a random order with no duplicate entries.

In case there is a need to generate a list of Wi-Fi networks sorted by signal strength, BGScript API contains **sme_scan_results_sort_rssi(...)** command for requesting scan results sorted by signal strength. The results are returned as events where the first event is the Wi-Fi network with the strongest signal.

> ⚠️ Where sme_start_scan(...) returns a list of Access Points, sme_scan_results_sort_rssi(...) returns a list of Wi-Fi networks where the signal strength and BSSID are that of the strongest Access Point in that particular network.

Performing a Wi-Fi scan

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # No results received yet.
  results = 0
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Initiate a scan. This call will trigger sme_scanned() event once done. The results
  # are returned as sme_scan_result() events.
  call sme_start_scan(0, 0, 0)
end
# Event received when a scan has been completed.
event sme_scanned(status)
  # Scanning completed.
end
# Event received for each Access Point discovered during the scan.
event sme_scan_result(bssid, channel, rssi, snr, secure, ssid_len, ssid_data)
end
```

Generating a list of Wi-Fi networks sorted by signal strength

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Initiate a scan. This call will trigger sme_scanned() event once done.
  call sme_start_scan(0, 0, 0)
end
# Event received when a scan has been completed.
event sme_scanned(status)
  # Scanning completed. Request a list of ten strongest Wi-Fi networks.
  call sme_scan_results_sort_rssi(10)
end
# Event received during a scan results sort for each Wi-Fi network.
event sme_scan_sort_result(bssid, channel, rssi, snr, secure, ssid_len, ssid_data)
end
# Event received when a scan results sort has been completed.
event sme_scan_sort_finished()
  # List of networks received.
end
```

5.2.5 Connecting to a Wi-Fi network

Connecting to a Wi-Fi network can be done either by using **sme_connect_bssid(...)** or **sme_connect_ssid(...)** command. The difference is that the former connects to a specified Wi-Fi Access Point identified by the BSSID parameter while the latter connects using the network name and selection of the strongest Wi-Fi Access Point automatically in case there are multiple possibilities.

> ⚠️ Connecting to a specific BSSID requires that a scan has been performed before issuing the connect command.

Connecting using a specific BSSID

```
dim connected
dim connect_bssid(6)
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Device is not connected yet.
  connected = 0
  # BSSID to connect to (1C:BD:B9:93:B4:24).
  connect_bssid(0:6) = "\x1C\xBD\xB9\x93\xB4\x24"
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Connecting a specific BSSID requires a scan. This call will trigger sme_scanned() event once
done.
  call sme_start_scan(0, 0, 0)
end
# Event received when a scan has been completed.
event sme_scanned(status)
  # Connect to the specified BSSID. This call will trigger sme_connected() event on success.
  call sme_connect_bssid(connect_bssid(0:6))
end
# Event received after a connection attempt succeeds.
event sme_connected(status, hw_interface, bssid)
  # Device is connected.
  connected = 1
end
```

Connecting using a network name

```
dim connected
dim connect_ssid(32)
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Device is not connected yet.
  connected = 0
  # SSID to connect to (Test_Open).
  connect_ssid(0:9) = "Test_Open"
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Connect using a network name. This call will trigger sme_connected() event on success.
  call sme_connect_ssid(9, connect_ssid(0:9))
end
# Event received after a connection attempt succeeds.
event sme_connected(status, hw_interface, bssid)
  # Device is connected.
  connected = 1
end
```

5.2.6 Creating a Wi-Fi Access Point

A Wi-Fi Access Point can be created using **sme_start_ap_mode(...)** command. Before the Access Point can be created, the device needs to be switched to Access Point mode using **sme_set_operating_mode(...)** command. While the operating mode can be set at any point, the mode will take effect when Wi-Fi is enabled. If the mode needs to be changed after Wi-Fi has been enabled, Wi-Fi needs to disabled, mode changed and Wi-Fi re-enabled.

Creating a Wi-Fi Access Point

```
dim ap_channel
dim ap_security
dim ap_ssid(32)
dim ap_ssid_len
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # AP parameters to use. Channel 11, no encryption, SSID "Bluegiga".
  ap_channel = 11
  ap_security = 0
  ap_ssid_len = 8
  ap_ssid(0:ap_ssid_len) = "Bluegiga"
  # Set Wi-Fi operating mode to Access Point (2). This needs to be called before enabling Wi-Fi.
  call sme_set_operating_mode(2)
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Start Wi-Fi Access Point mode. This call will trigger sme_ap_mode_started() event on success.
  call sme_start_ap_mode(ap_channel, ap_security, ap_ssid_len, ap_ssid(0:ap_ssid_len))
end
# Event received after AP mode has been started.
event sme_ap_mode_started(hw_interface)
  # Wi-Fi Access Point mode started.
end
```

Creating a secure Wi-Fi Access Point

```
dim ap_channel
dim ap_security
dim ap_ssid(32)
dim ap_ssid_len
dim ap_password(63)
dim ap_password_len
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # AP parameters to use. Channel 11, WPA2 security, SSID "Bluegiga", password "testtest".
  ap_channel = 11
  ap_security = 2
  ap_ssid_len = 8
  ap_ssid(0:ap_ssid_len) = "Bluegiga"
  ap_password_len = 8
  ap_password(0:ap_password_len) = "testtest"
  # Set operating mode to Access Point (2). This needs to be called before enabling Wi-Fi.
  call sme_set_operating_mode(2)
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Set Access Point password. This needs to called before starting the Access Point mode.
  call sme_set_ap_password(ap_password_len, ap_password(0:ap_password_len))
  # Start the Access Point mode. This call will trigger sme_ap_mode_started() event on success.
  call sme_start_ap_mode(ap_channel, ap_security, ap_ssid_len, ap_ssid(0:ap_ssid_len))
end
# Event received after AP mode has been started.
event sme_ap_mode_started(hw_interface)
  # Wi-Fi Access Point created.
end
```

5.2.7 Using Wi-Fi Protected Setup

Wi-Fi Protected Setup allows the device to obtain the network name and password of a compatible Wi-Fi network without having the user manually input them. The process is started by issuing **sme_start_wps(...)** command.

Using Wi-Fi Protected Setup with PushButton method

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Start Wi-Fi Protected Setup using PushButton method. This call will trigger sme_wps_completed()
event on success.
  call sme_start_wps()
end
# Event received for Wi-Fi network name.
event sme_wps_credential_ssid(hw_interface, ssid_len, ssid)
end
# Event received for Wi-Fi password.
event sme_wps_credential_password(hw_interface, password_len, password_data)
end
# Event received after Wi-Fi Protected Setup has been completed.
event sme_wps_completed(hw_interface)
  # Wi-Fi Protected Setup completed.
end
```

5.3 Hardware Interfaces

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

5.3.1 IO

Writing I/O Status

The example below shows how to write the status of D4 to D7 pins, which are connected to LEDs (Light Emitting Diodes) on DKWF121 (Development kit for WF121).

Enabling and catching IO interrupts

```
dim b
# boot event listener
event system_boot(major,minor,patch,build,bootloader,tcpip,hw)

    # Config IO port direction - Set port D pins D4 to D7 as outputs
    call hardware_io_port_config_direction(3,$00f0,$0000)
    b=0
    # Start a continuous software timer
    call hardware_set_soft_timer(1000,0,0)
end
# software timer event listener
event hardware_soft_timer(handle)

    if (b = 0) then
        # IO port write - Turn ON all the LEDs (pins D4 to D7)
        call hardware_io_port_write(3,$00f0,$00f0)
        b=1
    else
        b=0
        # IO port write - Turn OFF all the LEDs (pins D4 to D7)
        call hardware_io_port_write(3,$00f0,$0000)
    end if
end
```

Catching I/O Interrupts

The example below shows how to catch I/O interrupts.

Enabling and catching IO interrupts

```
dim b
# boot event listener
event system_boot(major,minor,patch,build,bootloader,tcpip,hw)

 # Config IO port direction - Set port D pins D10, D0, D9 and D11 as inputs
 call hardware_io_port_config_direction(3,$0e01,$0e01)
 # enable interrupts on pins RD0/INT0, RD9/INT2, RD10/INT3, RD11/INT4
 # INT0 0x0, INT2 = 0x4, INT3 = 0x8 and INT4=0x10
 call hardware_external_interrupt_config($1d, $1d)
 # Config IO port direction - Set port D pins  4 to 7 as output
 call hardware_io_port_config_direction(3,$00f0,$0000)
 b=0
end
# IO interrupt listener
event hardware_external_interrupt(irq, timestamp)

 if (b = 0) then
     # IO port write - Turn ON all the LEDs
     call hardware_io_port_write(3,$00f0,$00f0)
     b=1
 else
     # IO port write - Turn OFF all the LEDs
     call hardware_io_port_write(3,$00f0 ,$0000)
     b=0
 end if
end
```

5.3.2 I2C

When I2C is enabled and configured in the hardware configuration file it appears as an endpoint. Data can be sent to I2C using **Endpoint Send** command and received via the **Endpoint Data** event.
WF121 automatically handles I2C clock stretching. Also repeated starts are created when starting I2C write or read without stopping the last transfer.

Writing to Serial EEPROM

The example below shows to to write data to a serial EEPROM using I2C interface.

```
I2C write operation
```

```
#Start write sequence to EEPROM at I2C address 0x50. I2C endpoint is at index 4.
call i2c_start_write(4,$50)
#EEPROM requires 2byte address to write to.
call endpoint_send(4,2,"\x00\x00")
#Write data to eeprom
call endpoint_send(4,13,"Hello, World!")
#Stop write sequence
call i2c_stop(4)
```

Reading from Serial EEPROM

The example below on the other hand shows to to read data from EEPROM using I2C interface.

```
I2C read operation
```

```
#Start write sequence to eeprom at I2C address 0x50. I2C endpoint is at index 4.
call i2c_start_write(4,$50)
#EEPROM requires 2byte address to read from.
call endpoint_send(4,2,"\x00\x00")
#Start read sequence. A repeated start is automatically generated. Read operation also requires
the number of bytes to be read as a parameter.
call i2c_start_read(4,$50,13)
#Stop read sequence
call i2c_stop(4)
```

Data read from the EEPROM will be received via the **endpoint_data** event.

5.3.3 RTC Usage

The example below shows how to initialize the RTC (Real Time Clock) and configure it to generate alarms.

I2C write operation

```
dim i,i2,l,m
dim result,year,month,day,weekday,hour,minute,second
# BGScript function to print the RTC value to a human readable timestamp.
procedure print_int(int,digits)
    i=int
    l=digits
    m=1
    while l>1
        m=m*10
        l=l-1
    end while
    while m>0
        i2=i/m
        i=i-i2*m
        call endpoint_send(0,1,$30+i2)
        m=m/10
    end while
end
# Catching system start-up
event system_boot(major,minor,patch,build,bootloader,tcpip,hw)
    # Initialize RTC
    call hardware_rtc_init(1,0)
    # configure the starting time for RTC
    call hardware_rtc_set_time(2013,6,3,1,11,0,0)

    # Enable RTC alarm to generate 5 RTC events once every ten seconds
    call hardware_rtc_set_alarm(6,3,1,11,0,3,hardware_alarm_every_ten_seconds,5)
end
# Catch the RTC alarm event and print the data to UART endpoint
event hardware_rtc_alarm()
    call hardware_rtc_get_time()(result,year,month,day,weekday,hour,minute,second)
    call print_int(year,4)
    call endpoint_send(0,1,"-")
    call print_int(month,2)
    call endpoint_send(0,1,"-")
    call print_int(day,2)
    call endpoint_send(0,1," ")
    call print_int(hour,2)
    call endpoint_send(0,1,":")
    call print_int(minute,2)
    call endpoint_send(0,1,":")
    call print_int(second,2)
    call endpoint_send(0,2,"\r\n")
end
```

⚠ For the data to be visible in the UART the following hardware configuration needs to exist:

```
<hardware>
  ...
  <uart channel="0" baud="500000" api="false" />
  <uart channel="1" baud="500000" api="true" handshake="true" />
</hardware>
```

5.3.4 Ethernet

The Ethernet interface can be bridged either to the device's local TCP/IP stack or to a connected Wi-Fi network. The desired route is set using **ethernet_set_dataroute(...)** command. When the interface is bridged to the local stack, Wi-Fi Access Point needs to be active. BGScript API also provides **ethernet_connected(...)** command that can be used check whether the Ethernet cable is connected. In case there is a need to disable the Ethernet link, **ethernet_close(...)** command can be used.

> ⚠️ The following configuration needa to be in the *project.xml / hardware.xml* to enable the Ethernet interface and allow BGScript to access it.
>
> ```xml
> <hardware>
>   ...
>   <ethernet enable="1"/>
> </hardware>
> ```

Verifying Ethernet cable is connected

```
dim state
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  call ethernet_connected()(state)
  if state = 0
    # Ethernet cable is not connected.
  else
    # Ethernet cable is connected.
  end if
end
```

Bridging Ethernet interface to the local TCP/IP stack

```
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Set operating mode to Access Point (2). This needs to be called before enabling Wi-Fi.
  call sme_set_operating_mode(2)
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Start the Access Point mode. This call will trigger sme_ap_mode_started() event on success.
  call sme_start_ap_mode(11, 0, 8, "Bluegiga")
end
# Event received after AP mode has been started.
event sme_ap_mode_started(hw_interface)
  # Wi-Fi Access Point started, start routing Ethernet traffic to the local TCP/IP stack.
  # This call will also initialise the link and trigger ethernet_link_status() when the Ethernet
  # link is ready.
  call ethernet_set_dataroute(2)
end
# Event received when Ethernet link status changes.
event ethernet_link_status(state)
  if state = 0
    # Ethernet link is down.
  else
    # Ethernet link is up.
  end if
end
```

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Enable Wi-Fi, this call will trigger sme_wifi_is_on() event.
  call sme_wifi_on()
end
# Event received after Wi-Fi has been switched on.
event sme_wifi_is_on(state)
  # Connect using a network name. This call will trigger sme_connected() event on success.
  call sme_connect_ssid(9, "Test_Open")
end
# Event received after a connection attempt succeeds.
event sme_connected(status, hw_interface, bssid)
  # Device is connected to a Wi-Fi network, start bridging Ethernet traffic to the network.
  # This call will also initialise the link and trigger ethernet_link_status() when the Ethernet
  # link is ready.
  call ethernet_set_dataroute(1)
end
# Event received when Ethernet link status changes.
event ethernet_link_status(state)
  if state = 0
    # Ethernet link is down.
  else
    # Ethernet link is up.
  end if
end
```

5.4 Timers

This section describes how to use timers with BGScript.

5.4.1 Continuous Timer Generated Interrupt

This example will generate a regular interrupt between specified intervals continuously at every 500 ms.

```
dim count
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # No events received yet.
  count = 0
  # Schedule timer #0 events to occur every 500ms.
  call hardware_set_soft_timer(500, 0, 0)
end
# Event received when a timer is triggered.
event hardware_soft_timer(handle)
  if handle = 0
    # Timer 0 has been triggered, increase count.
    count = count + 1
    if count >= 10
      # Cancel the timer.
      call hardware_set_soft_timer(0, 0, 0)
    end if
  end if
end
```

5.4.2 Single Timer Generated Interrupt

This example will generate a single interrupt after the timer has elapsed the selected time (here 500 ms).

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Schedule a single-shot timer #0 event to occur in 500ms
  call hardware_set_soft_timer(500, 0, 1)
end
# Event received when a timer is triggered.
event hardware_soft_timer(handle)
  if handle = 0
    # Timer 0 has been triggered.
  end if
end
```

5.5 Endpoints

This section contains examples on how to utilize endpoints using BGScript.

5.5.1 UART endpoint

An UART endpoint can operate in two different modes: streaming or BGAPI. In streaming mode any incoming UART data is transparently routed to another endpoint. A typical use case for this is sending and receiving TCP /IP data through UART. In BGAPI mode, data written to UART is handled as BGAPI commands. While the operating mode can be set using **endpoint_set_streaming(...)** command, it's typically set in *project.xml / hardware.xml*.

When operating in streaming mode, incoming UART data is discarded by default. The endpoint data is routed to can be adjusted using **endpoint_set_streaming_destination(...)** command.

> ⚠ The following configurations need to be in the *project.xml / hardware.xml* to enable the UART interface (s) and allow BGScript to access it.
>
> ```
> <hardware>
>   ...
>   <uart channel="0" baud="5000000" api="false" handshake="true" />
>   <uart channel="1" baud="5000000" api="true" handshake="true" />
> </hardware>
> ```

Writing to UART endpoint

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Write data to UART 0.
  call endpoint_send(0, 6, "Hello\n")
end
```

Setting UART operating mode

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Set UART 0 to streaming mode.
  call endpoint_set_streaming(0, 1)
  # Set UART 1 to BGAPI mode.
  call endpoint_set_streaming(1, 0)
end
```

```
# PREREQUISITE: A TCP connection is active and the TCP endpoint is stored in variable "client_endpo
int".
dim client_endpoint
  ...
  # Route incoming UART 0 data to TCP endpoint.
  call endpoint_set_streaming_destination(0, client_endpoint)
  # Route incoming TCP endpoint data to UART 0.
  call endpoint_set_streaming_destination(client_endpoint, 0)
  ...
```

5.5.2 USB endpoint

An USB endpoint can operate in two different modes: streaming or BGAPI. In streaming mode any incoming USB data is transparently routed to another endpoint. A typical use case for this is sending and receiving TCP /IP data through USB. In BGAPI mode, data written to USB is handled as BGAPI commands. Unlike UART endpoints, the operating mode cannot be adjusted on the fly and must be set in *project.xml / hardware.xml* instead.

When operating in streaming mode, incoming USB data is discarded by default. The endpoint data is routed to can be adjusted using **endpoint_set_streaming_destination(...)** command.

⚠️ The following configurations need to be in the *project.xml / hardware.xml* to enable the USB interface and allow BGScript to access it.

```
<hardware>
  ...
  <usb descriptor="cdc.xml" api="false" />
</hardware>
```

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Write data to USB.
  call endpoint_send(3, 6, "Hello\n")
end
```

```
# PREREQUISITE: A TCP connection is active and the TCP endpoint is stored in variable "client_endpo
int".
dim client_endpoint
  ...
  # Route incoming USB data to TCP endpoint.
  call endpoint_set_streaming_destination(3, client_endpoint)
  # Route incoming TCP endpoint data to USB.
  call endpoint_set_streaming_destination(client_endpoint, 3)
  ...
```

5.5.3 Drop endpoint

In addition to hardware, TCP and UDP endpoints there is a special Drop endpoint. Drop endpoint is always in endpoint index 31. Any data sent to the endpoint is discarded. The endpoint cannot be used as an input. The endpoint routing can be adjusted using **endpoint_set_streaming_destination(...)** command.

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Route incoming UART 0 data to Drop endpoint. This discards the data.
  call endpoint_set_streaming_destination(0, 31)
end
```

5.6 PS store

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

5.6.1 Reading a PS key

This example shows how to read PS keys.

Reading a specific key from the Persistent Store

```
dim result
dim value_len
dim value_data(4)
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Retrieve the operating mode. Keys provided by Bluegiga are stored in well-known indexes.
  call flash_ps_load($5)(result, value_len, value_data(0:value_len))
end
```

Reading a specific key from the Persistent Store using an enumerated key index

```
dim result
dim value_len
dim value_data(4)
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Retrieve the operating mode using an enumerated key index. Enumeration exists for keys
provided by Bluegiga.
  call flash_ps_load(FLASH_PS_KEY_MODULE_SERVICE)(result, value_len, value_data(0:value_len))
end
```

Reading all keys from the Persistent Store

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Retrieve all keys from PS. This call will trigger flash_ps_key() event for each key.
  call flash_ps_dump()
end
# Event received for each PS key.
event flash_ps_key(key, value_len, value_data)
  if key = $FFFF
    # All keys retrieved, this is not a real key.
  else
    # A PS key retrieved.
  end if
end
```

5.6.2 Writing a PS key

This example shows how to write data into PS keys.

> ⚠️ Key indexes 0x8000 through 0x807F are allocated for user data.

Writing a key to the Persistent Store

```
dim value_len
dim value_data(6)
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Data to be written.
  value_len = 6
  value_data(0:value_len) = "MyData"
  # Write the data to key index 0x8000.
  call flash_ps_save($8000, value_len, value_data(0:value_len))
end
```

5.6.3 Deleting a PS key

This example shows how to delete PS keys.

Deleting a key from the Persistent Store

```
# Event received when the system has been successfully started up.
event system_boot(major, minor, patch, build, bootloader_version, tcpip_version, hw)
  # Delete key index 0x8000.
  call flash_ps_erase($8000)
end
```

5.7 TCP/IP

These examples show how to use the built-in TCP/IP stack.

5.7.1 TCP client

A TCP connection is created by issuing the **tcpip_tcp_connect(...)** command. This creates an endpoint which is returned in the command response. The endpoint can be used to control the connection as well as to send and receive data.

Creating a TCP connection

```
dim server_ipaddr
dim server_port
dim client_endpoint
dim result
  ...
  # Create a connection to the server. The created endpoint is stored to variable "client_endpoint"
.
  call tcpip_tcp_connect(server_ipaddr, server_port, -1)(result, client_endpoint)
  ...
# Event received when a TCP/IP endpoint status changes.
event tcpip_endpoint_status(endpoint, local_ip, local_port, remote_ip, remote_port)
  if endpoint = client_endpoint
    # This is a status notification for the TCP/IP endpoint.
  end if
end
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = client_endpoint && active = 1
    # The connection is now active.
  end if
end
```

Receiving and sending TCP data

```
# PREREQUISITE: A TCP connection is active.
dim client_endpoint
  ...
  # Route incoming server TCP traffic to BGAPI endpoint so that the traffic is handled as
  # BGScript endpoint_data() events. This can also done by default by setting the routing
  # parameter to -1 in tcpip_start_tcp_server() command.
  call endpoint_set_streaming_destination(client_endpoint, -1)
  ...
# Event received when BGAPI endpoint receives data
event endpoint_data(endpoint, data_len, data_data)
  if endpoint = client_endpoint
    # Incoming data from the server, send a reply.
    call endpoint_send(client_endpoint, 5, "Hello")
  end if
end
```

Closing a TCP connection

```
# PREREQUISITE: A TCP connection is active.
dim client_endpoint
  ...
  # Close the connection from the client side.
  call endpoint_close(client_endpoint)
  ...
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = client_endpoint && active = 0
    # The TCP connection is now closed.
  end if
end
```

Handling a closing TCP connection

```
# PREREQUISITE: A TCP connection is active.
# ACTION: The server closes the TCP connection.
dim client_endpoint
# Event received an endpoint is closed by the server
event endpoint_closing(reason, endpoint)
  if endpoint = client_endpoint
    # The TCP connection is closing.
  end if
end
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = client_endpoint && active = 0
    # The TCP connection is now closed.
  end if
end
```

5.7.2 TCP server

A TCP server is started by issuing the **tcpip_start_tcp_server(...)** command. This creates a server endpoint which is returned in the command response. The endpoint can be used to stop the TCP server when it's no longer needed. It cannot be used to send or receive data.

When clients connect to the TCP server port, a new endpoint is created for each connection. By default the endpoint traffic is routed to the endpoint given as default_destination argument to tcpip_start_tcp_server(...) command. It's possible to change the routing by calling **endpoint_set_streaming_destination(...)** command.

Starting a TCP server

```
dim server_endpoint
dim result
  ...
  # Start TCP server on port 80. The created endpoint is stored to variable "server_endpoint".
  call tcpip_start_tcp_server(80, -1)(result, server_endpoint)
  ...
# Event received when a TCP/IP endpoint status changes.
event tcpip_endpoint_status(endpoint, local_ip, local_port, remote_ip, remote_port)
  if endpoint = server_endpoint
    # This is a status notification for the server TCP/IP endpoint
  end if
end
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = server_endpoint
    # This is a status notification for the server endpoint
  end if
end
```

Handling an incoming client TCP connection

```
# PREREQUISITE: A TCP server has been started on port 80.
# ACTION: A client connects to the server port.
dim client_endpoint
# Event received when a TCP/IP endpoint status changes.
event tcpip_endpoint_status(endpoint, local_ip, local_port, remote_ip, remote_port)
  if local_port = 80
    # This is an incoming client TCP connection to port 80. Store the endpoint.
    client_endpoint = endpoint
  end if
end
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = client_endpoint && active = 1
      # The client TCP connection is now active.
    end if
  end if
end
```

Receiving and sending TCP data

```
# PREREQUISITE: A client TCP connection is active.
dim client_endpoint
  ...
  # Route incoming client TCP traffic to BGAPI endpoint so that the traffic is handled as
  # BGScript endpoint_data() events. This can also done by default by setting the endpoint
  # parameter to -1 in tcpip_start_tcp_server() command.
  call endpoint_set_streaming_destination(client_endpoint, -1)
  ...
# Event received when BGAPI endpoint receives data
event endpoint_data(endpoint, data_len, data_data)
  if endpoint = client_endpoint
    # Incoming data from the client, send a reply.
    call endpoint_send(client_endpoint, 5, "Hello")
  end if
end
```

Closing a client TCP connection

```
# PREREQUISITE: A client TCP connection is active.
dim client_endpoint
  ...
  # Close the client TCP connection from the server side.
  call endpoint_close(client_endpoint)
  ...
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = client_endpoint && active = 0
    # The client TCP connection is now closed.
  end if
end
```

Handling a closing client TCP connection

```
# PREREQUISITE: A client TCP connection is active.
# ACTION: A client closes the TCP connection.
dim client_endpoint
# Event received an endpoint is closed by the client
event endpoint_closing(reason, endpoint)
  if endpoint = client_endpoint
    # The client TCP connection is closing.
  end if
end
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = client_endpoint && active = 0
    # The client TCP connection is now closed.
  end if
end
```

Stopping a TCP server

```
# PREREQUISITE: A TCP server has been started.
dim server_endpoint
  ...
  # Stop the TCP server.
  call endpoint_close(server_endpoint)
  ...
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = server_endpoint && active = 0
    # The TCP server is now stopped.
  end if
end
```

5.7.3 UDP client

An UDP connection is created by issuing the **tcpip_udp_connect(...)** command. This creates an endpoint which is returned in the command response. The endpoint can be used to control the connection as well as to send data. Unlike a TCP endpoint, an UDP endpoint is not bi-directional, a separate UDP server endpoint needs to be created for the incoming data.

By default an UDP connection is assigned a random source port. In case it needs to be adjusted for some reason, **tcpip_udp_bind(...)** command can be used.

Creating an UDP connection

```
dim server_ipaddr
dim server_port
dim client_endpoint
dim result
  ...
  # Create a connection to the server. The created endpoint is stored to variable "client_endpoint"
.
  call tcpip_udp_connect(server_ipaddr, server_port, -1)(result, client_endpoint)
  ...
# Event received when a TCP/IP endpoint status changes.
event tcpip_endpoint_status(endpoint, local_ip, local_port, remote_ip, remote_port)
  if endpoint = client_endpoint
    # This is a status notification for the TCP/IP endpoint.
  end if
end
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = client_endpoint && active = 1
    # The connection is now active.
  end if
end
```

Sending UDP data

```
# PREREQUISITE: An UDP connection is active.
dim client_endpoint
  ...
  # Send data to the server.
  call endpoint_send(client_endpoint, 5, "Hello")
  ...
```

Closing an UDP connection

```
# PREREQUISITE: An UDP connection is active.
dim client_endpoint
  ...
  # Close the connection.
  call endpoint_close(client_endpoint)
  ...
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = client_endpoint && active = 0
    # The UDP connection is now closed.
  end if
end
```

Changing the source port of an UDP connection

```
# PREREQUISITE: An UDP connection is active.
dim client_endpoint
  ...
  # Change the UDP source port to 8080.
  call tcpip_udp_bind(client_endpoint, 8080)
  ...
```

5.7.4 UDP server

An UDP server is started by issuing the **tcpip_start_udp_server(...)** command. This creates an endpoint which is returned in the command response. The endpoint can be used to control the connection as well as to receive data. Unlike a TCP endpoint, an UDP endpoint is not bi-directional, a separate UDP client endpoint needs to be created for the outgoing data.

Starting an UDP server

```
dim server_endpoint
dim result
  ...
  # Start UDP server on port 80. The created endpoint is stored to variable "server_endpoint".
  call tcpip_start_udp_server(80, -1)(result, server_endpoint)
  ...
# Event received when a TCP/IP endpoint status changes.
event tcpip_endpoint_status(endpoint, local_ip, local_port, remote_ip, remote_port)
  if endpoint = server_endpoint
    # This is a status notification for the server TCP/IP endpoint
  end if
end
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = server_endpoint
    # This is a status notification for the server endpoint
  end if
end
```

Receiving UDP data

```
# PREREQUISITE: An UDP server has been started.
# ACTION: A client sends data to the server port.
dim server_endpoint
  ...
  # Route incoming client UDP traffic to BGAPI endpoint so that the traffic is handled as
  # BGScript tcpip_udp_data() events. This can also done by default by setting the endpoint
  # parameter to -1 in tcpip_start_udp_server() command.
  call endpoint_set_streaming_destination(server_endpoint, -1)
  ...
# Event received when BGAPI endpoint receives UDP data
event tcpip_udp_data(endpoint, source_address, source_port, data_len, data_data)
  if endpoint = server_endpoint
    # Incoming data from a client.
  end if
end
```

Stopping an UDP server

```
# PREREQUISITE: An UDP server has been started.
dim server_endpoint
  ...
  # Stop the UDP server.
  call endpoint_close(server_endpoint)
  ...
# Event received when an endpoint status changes.
event endpoint_status(endpoint, type, streaming, destination, active)
  if endpoint = server_endpoint && active = 0
    # The UDP server is now stopped.
  end if
end
```

5.7.5 DNS resolver

Since TCP and UDP commands accept only IP addresses as parameters, a DNS name needs to be resolved to the corresponding IP address before it can be used. This can be accomplished using the **tcpip_dns_gethostbyname(...)** command.

5.8 Generic Tips and Tricks

This section contains generic BGScript tips and tricks embedded into the listed commands.

5.8.1 HEX to ASCII

This example code converts MAC addresses into ASCII strings.

Printing local BT address on the display in DKBLE112

```
# handle MAC event and covert it to ASCII
dim macaddr
event config_mac_address(hw_interface, mac)
 # Hex to ASCII conversion
 macaddr(10:1) = (mac(5:1)/$10) + 48 + ((mac(5:1)/$10)/10*7)
 macaddr(11:1) = (mac(5:1)&$f)  + 48 + ((mac(5:1)&$f )/10*7)
 macaddr(8:1) = (mac(4:1)/$10) + 48 + ((mac(4:1)/$10)/10*7)
 macaddr(9:1) = (mac(4:1)&$f)  + 48 + ((mac(4:1)&$f )/10*7)
 macaddr(6:1) = (mac(3:1)/$10) + 48 + ((mac(3:1)/$10)/10*7)
 macaddr(7:1) = (mac(3:1)&$f)  + 48 + ((mac(3:1)&$f )/10*7)
 macaddr(4:1) = (mac(2:1)/$10) + 48 + ((mac(2:1)/$10)/10*7)
 macaddr(5:1) = (mac(2:1)&$f)  + 48 + ((mac(2:1)&$f )/10*7)
 macaddr(2:1) = (mac(1:1)/$10) + 48 + ((mac(1:1)/$10)/10*7)
 macaddr(3:1) = (mac(1:1)&$f)  + 48 + ((mac(1:1)&$f )/10*7)
 macaddr(0:1) = (mac(0:1)/$10)+ 48 + ((mac(0:1)/$10)/10*7)
 macaddr(1:1) = (mac(0:1)&$f) + 48 + ((mac(0:1)&$f )/10*7)
end
```

5.8.2 UINT to ASCII

This example code converts a three digit integer into an ASCII string.

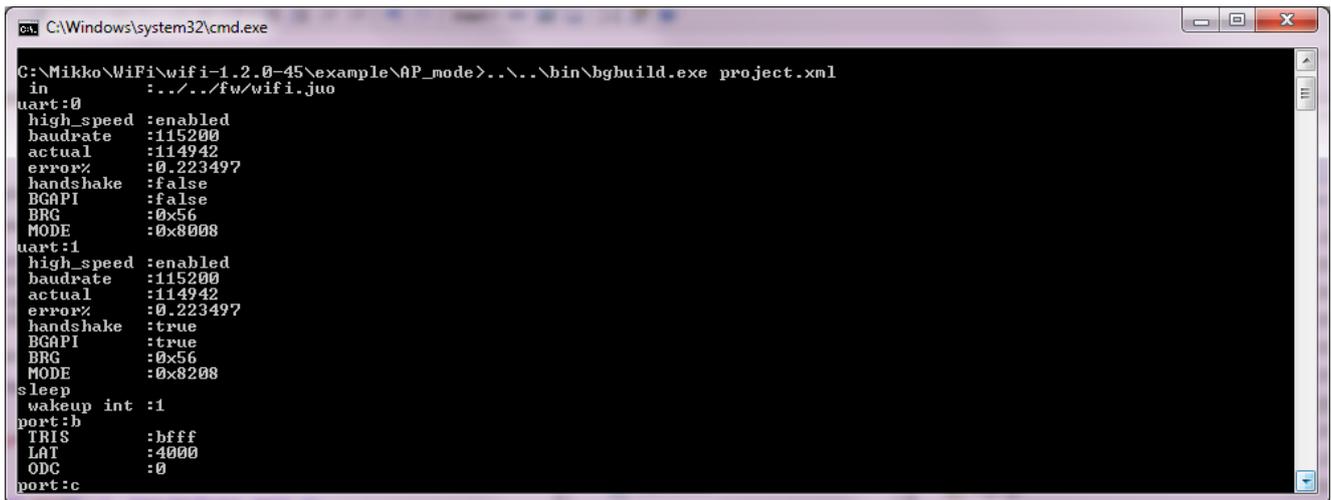**Converting 3 digit integer to ASCII**

```
dim int
dim string(3)
string(0:1) = (int / 100) + 48
string(1:1) = (int / 10) + (int / -100 * 10) + 48
string(2:1) = int + (int / 10 * -10) + 48
```

6 Compiling a Project to a Firmware Image

The project is compiled with the **bgbuild.exe** compiler and this can for example be done either using Bluegiga WiFiGUI software or using the Windows Command Line Prompt (**cmd.exe).** The example below shows how to compile a Wi-Fi Software project with the BGBuild compiler and cmd.exe to a firmware image which can be installed to a Wi-Fi module.

To compile the firmware binary:

- Open for example Windows command prompt
- Navigate (using '**cd**') to the folder where your project is
- Run the *bgbuild.exe* compiler as shown below, giving the project file as a parameter
- The syntax for the bgbuild compiler is : **bgbuild.exe <*project_file*>**

Figure: Compiling the project with BGBuild compiler

Based on the settings in the **project.xml** file the compiler will output .HEX and/or .DFU files to be installed into the Wi-Fi module with the PICkit 3 programmer or alternatively using the DFU update method.

The BGBuild compiler will output the following information.

Feature	Output	Explanation
uart:0	high_speed : enabled baudrate :115200 actual :114942 error% :0.223497 handshake :false BGAPI :false BRG :0x56MODE : 0x8008	This shows UART1 interface is enabled at 115200 bps baud rate. RTS/CTS handshaking is disabled. BGAPI protocol for this UART is disabled. The endpoint is allocated with ID 0 (shown in **uart:0**) and this ID can be used by the BGScript application or BGAPI commands to send data to it or to route for example UART endpoint to TCP endpoint.
uart:1	high_speed : enabled baudrate :5000000 actual :5000000 error% :0 handshake :true BGAPI :true BRG :0x1MODE : 0x8008	This shows UART2 interface is enabled at 5000000 bps baud rate. RTS/CTS handshaking is enabled. BGAPI protocol for this UART is enabled.
sleep	wakeup int :1	This message tells interrupt INT0 is enabled (pin 38).
port:N	TRIS :ffff LAT :0ODC :0	Shows the default configuration for Port N (B to G) and the setting for tri-state configuration bit mask, open drain configuration, and latched value for the port
script	compiler :c:/WiFi /wifi-1.2.0-42/wifi/bin /script_compiler.exe script :APMode.bgs api :../../api/wifiapi. xmlstack :512	Shows the directory where the bgbuild compiler is located.Shows the BGScript source code file.
Stack size	SW : 359044 HW :130 USB :0 Script:163 Free :152661 /512000(30%)	**SW** shows the flash usage (in B) of the firmware image. **HW** shows the size of the hardware configuration. **USB** shows the size of the USB interface descriptor. **Script** shows the size of BGScript code. **Free** shows the total size of the software and how much flash space is left in the device.

7 Installing the Firmware

The firmware can be installed either using the DFU protocol over UART or USB or via the debug interface using the Microchip **PICkit 3** tool and software.

7.1 Using PICkit 3

- As **PICkit 3** will erase the full flash, please write down the MAC (IEEE) address of your device
- Download and install **PICkit 3** software from Bluegiga web site
- Connect the **PICkit 3** to the debug interface of your WF121 (named ICSP on WF121 development kit) and connect the **PICkit 3** to your PC via USB interface.
- Start **PICkit 3** software
    - From **Device Family** select **PIC32**
    - From **Device** drop down list select model : **PIC32MX695F512H**
    - Verify the **STATUS** led on the PICkit 3 device turns **green**
    - From **File** select **Import Hex**
    - Choose the **.hex** file output by the **BGBuild** compiler
    - Press **Write**
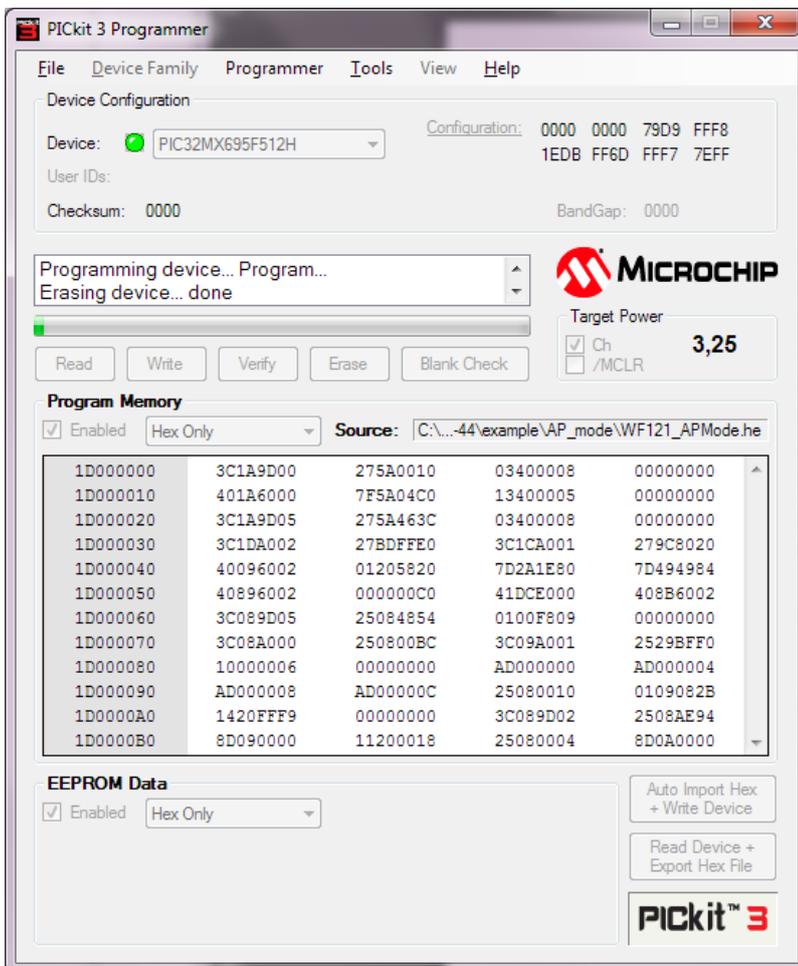- Wait for the programming to be successfully finalized

Figure: Programming firmware using PICkit 3

> ⚠ The MAC address can be restored with WIFIGUI software and by typing the original MAC address to the MAC address field not the Network page.

7.2 Using DFU over UART or USB

In order to install the firmware using DFU protocol, please do the following steps:

- Connect the WF121 Wi-Fi Module to a PC via UART or USB. Selection is done in **project.xml** of the firmware to generate by enabling api on UART
  <uart channel="1" baud="115200" **api="True"** *handshake="True" />*
  or USB
  <usb descriptor="cdc.xml" **api="True"** */>*
- Start **WiFiGUI** software
- Select the correct COM port and baud rate
  - Verify the communication works for example by pressing **Retrieve info** button
- Go to **Firmware** update subpage
  - Press **Boot in DFU** mode button
    - A successful DFU mode is indicated with the event : **wifi_evt_dfu_boot**
  - Select the correct .DFU file using the **Browse**... button
  - Press **Upload**
- Make sure the firmware is uploaded correctly and the device boots normally
  - A successful DFU upload is indicated with event: **wifi_rsp_dfu_flash_upload_finish result: 0**
  - A successful boot is indicated with event: **wifi_evt_system_boot**
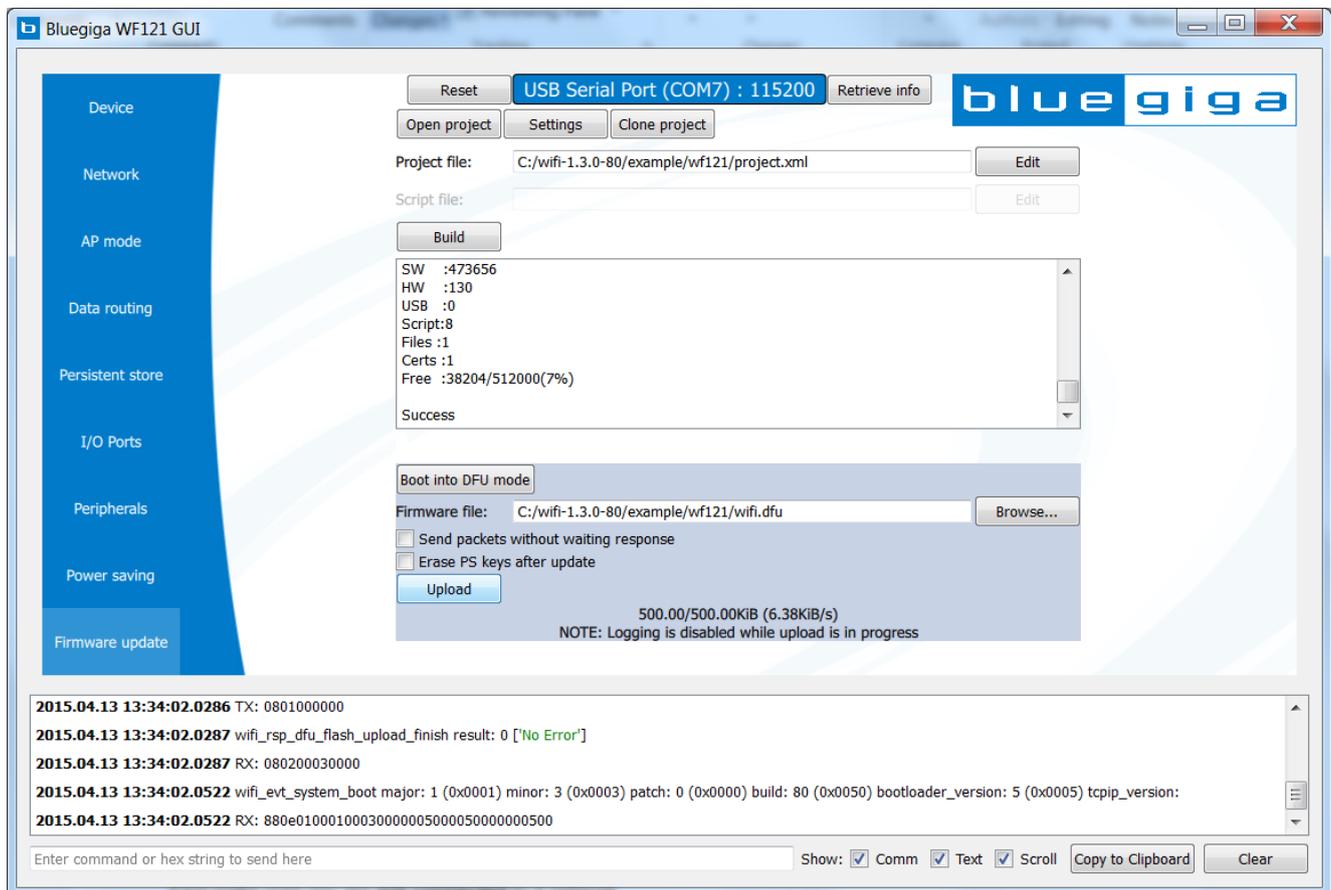

Figure: Updating firmware via DFU

8 BGScript editors

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

8.1 Notepad ++

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

8.1.1 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 BGScript syntax highlighting from www.bluegiga.com
2. Open Notepad++
3. IF USING THE NEWEST NOTEPAD++:
   a. 2a. In the **Language** menu, click **Defined your language...**
4. IF USING AN OLDER NOTEPAD++:
   a. 2b. In the **View** menu, click **User-Defined Dialogue...**
5. Click **Import**... and select the **userDefineLang_BGScript.xml** file
6. Copy **BGScript.xml** file to **<NPP-Install-Dir>\plugins\APIs**
7. In the **Settings** menu, click **Preferences**..., then **Backup/Autocompletion**
8. Enable Auto-Completion options as desired
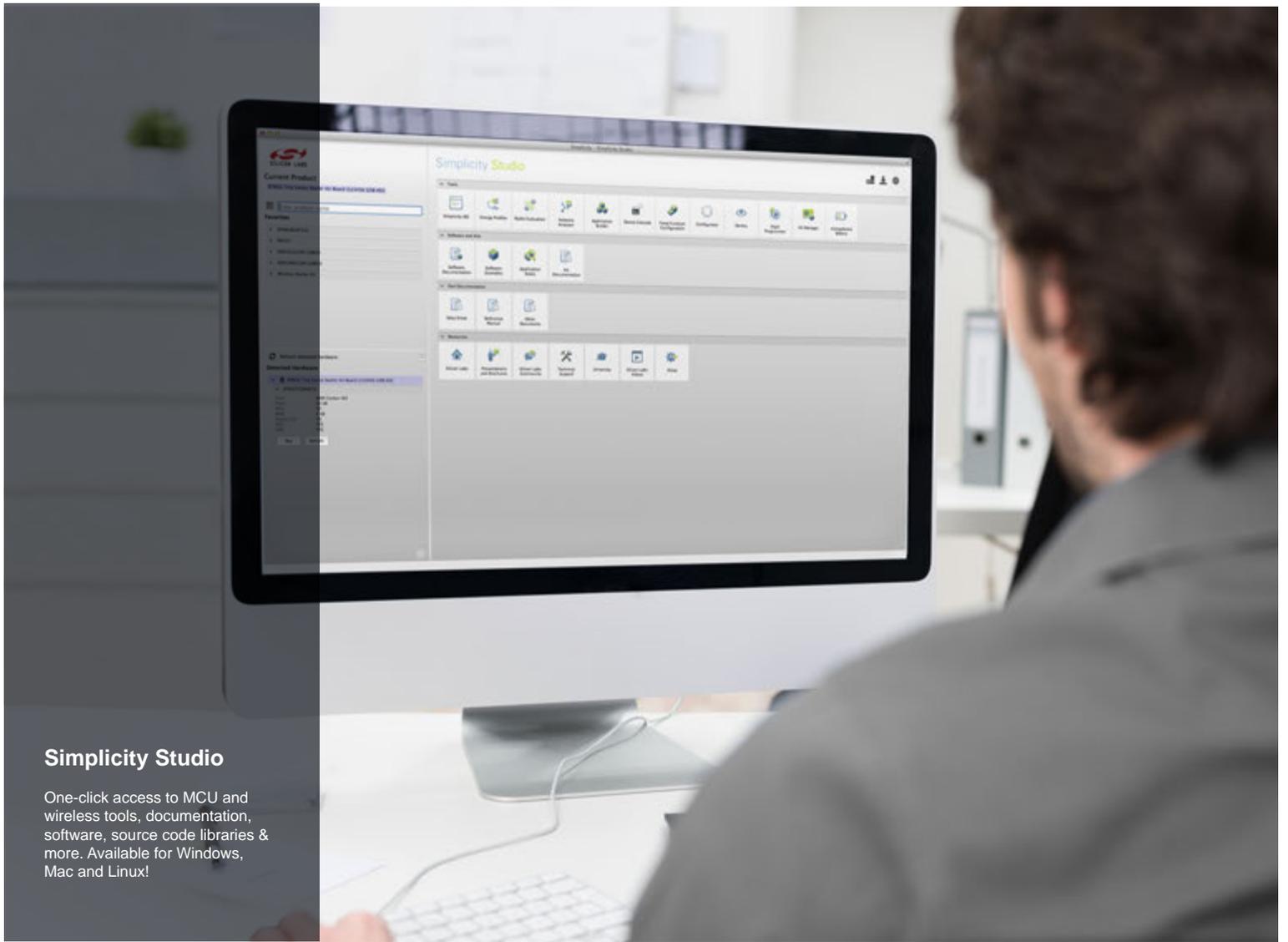9. Close and re-open Notepad++ for all settings to take effect

> ⚠ If you already have a BGScript user-defined language in Notepad++, you MUST remove it first. Also, the 'APIs' folder is typically found at 'C:\Program Files\Notepad++\plugins\APIs'.

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

Simplicity Studio

One-click access to MCU and
wireless tools, documentation,
software, source code libraries &
more. Available for Windows,
Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Disclaimer

Silicon Laboratories intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Laboratories products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Laboratories reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Laboratories shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Laboratories. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Laboratories products are not designed or authorized for military applications. Silicon Laboratories products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information

Silicon Laboratories Inc.® , Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOmodem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® and others are trademarks or registered trademarks of Silicon Laboratories Inc. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com