UG170: Wizard Gecko BGScript™ User's Guide

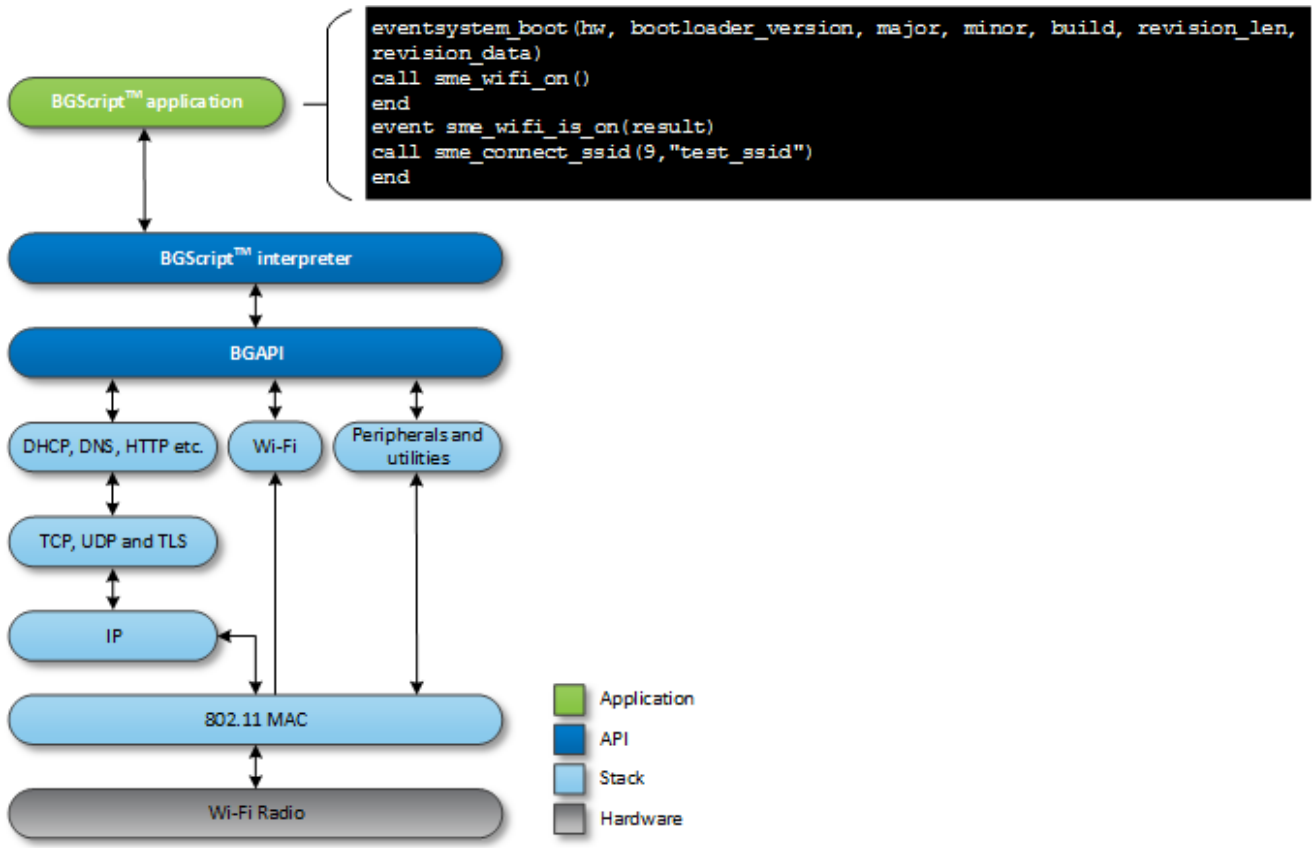
BGScript is a scripting language intended for programming simple applications. BGScript applications can be used to automate application functionalities, such as opening a connection, listening for GPIO interrupts, and even for reading and writing data via interfaces, such as UART, I2C or USB. BGScript can also be used for simple data processing, using the available BGScript arithmetic, bitwise, buffer, and data comparison operations.

BGScripting allows complete applications to be implemented without the need for an external host controller (MCU), since BGScript can be executed directly on the Silicon Labs wireless module.

This User's Guide describes BGScript syntax and contains useful code sniplets for some of the most common functionalities used with the Silicon Labs Wizard Gecko WGM110 Wi-Fi® Module.

KEY POINTS

- Create stand-alone applications
- Minimal programming experience needed
- Simple syntax
- Standard programming language features and operations
- Powerful functions to simplify complex tasks
- Easy access to common BGAPI features



1. Introduction

This section describes the benefits and limitations of BGScript.

1.1 Benefits of BGScript

There are numerous benefits to building a system with the BGScript scripting language instead of using an external host. Without the host controller, the size of the end application device and the number of components in the bill-of-materials can be reduced. Power consumption is lower, which allows the end product to operate longer with the same battery or allows the reduction of the size and thus cost of the battery. As the overall complexity of the end application product decreases, development time and risks are also decreased. BGScript can be developed with free-of-charge tools, so there is no need to invest in expensive IDEs and debuggers.

BGScript gives access to exactly the same APIs as are available over a host interface (e.g. BGAPI). This means that developers who are familiar with the BGAPI can easily migrate to BGScript and vice versa. All commands and events are the same, so if the requirements of the project grow to exceed what the BGScript can provide, developers have the flexibility to switch from BGScript to an external host controller without having to learn new APIs.

1.2 Limitations of BGScript

Because BGScript is an interpreted language, its limitations are typically reached in applications where fast data processing or data collection from peripherals is needed. If your application, for example, needs to read an accelerometer thousands or tens of thousands of times per second and process the data, BGScript based applications will most likely not offer enough performance.

1.3 BGScript as Part of the Wizard Gecko Wi-Fi Stack

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

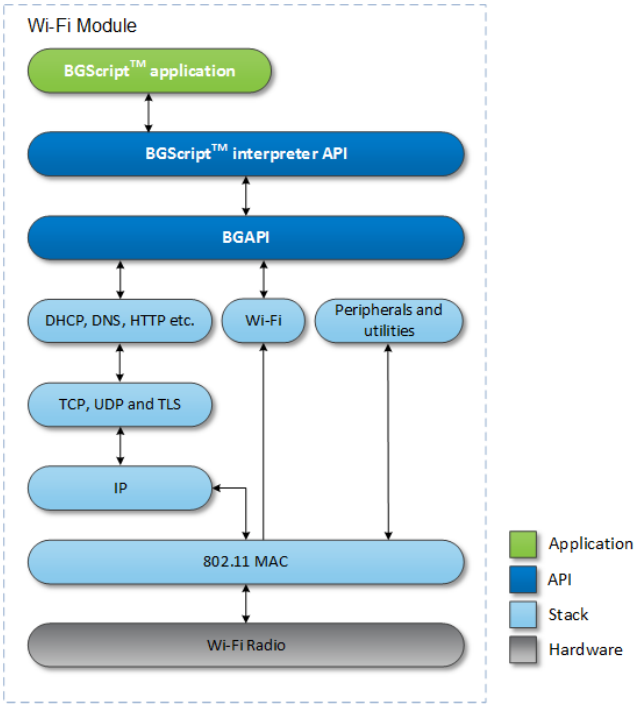


Figure 1.1. BGScript Architecture in Wizard Gecko Wi-Fi Software

1.4 BGScript in Practice

The example BGScript below automatically connects to an access point and a TCP server and then sends some data to the server.

Example: Connecting to an Access Point and to a TCP Server and Sending Data to a TCP server

```
#define variable for call result and out endpoint we use for sending data
dim result,out_ep

#system has booted, start WiFi subsystem by using BGAPI command
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)

    call sme_wifi_on()
end

#WiFi has started, connect to Access Point by using SSID
event sme_wifi_is_on(result)

    call sme_connect_ssid(11,"silabs_test")
end

#interface status has changed, check if it has gone up
event sme_interface_status(hw_interface,status)
    if status = 1
        #connect to TCP server at address 192.168.11.2 in port 1234,
        # as a function call response we get endpoint associated to this connection
        call tcpip_tcp_connect(192.168.11.2,1234,1)(result,out_ep)

    end if
end


# Endpoint status has changed. If it is our tcp connection and it has actived (connected)
# then send string of data to server
event endpoint_status(endpoint,type,streaming,destination,active)
    if endpoint= out_ep && active = 1
        call endpoint_send(out_ep,7, "testing")
    end if
end
```

2. BGScript Syntax

BGScript has a BASIC type of syntax. The code is executed only in response to event messages. Lines are executed in successive order. A line starts from an event definition and is finished by a ***return*** or ***end***. Each line represents a single command.

Below is an example BGScript which automates Wi-Fi startup on a Wizard Gecko WGM110 Wi-Fi Module. The script starts the Wi-Fi subsystem on boot and when booted up connects to an SSID.

Example: Wi-Fi Subsystem Start on Boot and Connection to an SSID After Boot Up

```
eventsystem_boot(hw, bootloader_version, major, minor, build, revision_len,
revision_data)
       call sme_wifi_on()
end

event sme_wifi_is_on(result)
       call sme_connect_ssid(9,"test_ssid")
end
```

2.1 Comments

Anything after a # character to the end of the line is ignored.

```
X=1 #comment
```

2.2 Variables and Values

This section describes the syntax for variables, values, global variables, constant values, buffers, strings, and constant strings in BGScript.

2.2.1 Values

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

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

2.2.2 Variables

Variables (excluding buffers) are signed 32-bit integer containers, stored in little-endian byte order. Variables must be defined before usage.

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

2.2.3 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 flash_ps_save(FLASH_PS_KEY_CNT, 4, j)
end
```

2.2.4 Constant Values

Constants are signed 32-bit integers stored in little-endian byte order. 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 for BGScript applications. Instead, constant values are stored in flash memory as part of the application code.

```
const x = 2
```

2.3 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 Wizard Gecko Wi-Fi Module application to prepare data before sending it via TCP.

As with variables, buffers need to be defined before use. The maximum size of a buffer is 256 bytes.

```
dim ssid(32)        # SSID string
dim ssid_len        # SSID string length

ssid_len = 14
ssid(0:ap_ssid_len) = "WGM110_Example"
call sme_connect_ssid(ssid_len, ssid)
```

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: The ***<size>*** parameter 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 as indicated in the example below:

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

Buffer index and size are optional. Default values, which are 0 for index and size defined in the variable declaration for size, are used in the place of left out values.

2.3.1 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"
    # do something
end if
```

This previous example is invalid, 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 that 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"
   # do something
end if
```

2.3.2 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 example:

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

2.3.3 Constant Strings

Constant strings must be defined before use. The maximum size of a constant string depends on application and stack usage. For standard Wi-Fi applications, a safe size is around 64 bytes.

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

Constant strings can be used in place of buffers. Note that in the following example the index and the size of the buffer are left as default values.

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

2.4 Expressions

Expressions are given using infix notation.

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

Table 2.1. Supported Mathematical Operators

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Less than	<
Is less than or equal to	<=
Is greater than	>
Is greater than or equal to	>=
Is equal to	=
Is not equal to	!=
Parentheses	( )

Note: Currently there is no support for modulo or power operators.

Table 2.2. Supported Bitwise Operators

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

Table 2.3. Supported Logical Operators

Operation	Symbol
AND	&&
OR	||

2.5 Commands

This section describes BGScript commands and their syntax.

2.5.1 event <event_name> (<event_parameters>)

A code block defined between ***event*** and ***end*** keywords is an event listener and will be run in response to a specific event. BGscript allows implementing multiple listeners for a single event. Each event listener will be executed in the order in which they appear in BGScript source code. Execution will stop when reaching the ***end*** keyword of the last event listener or the ***return*** keyword in any event listener. This event listeners chaining makes it possible to implement common script files (helper libraries) for generally used operations.

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).

```
# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)

# Config IO port direction - Set port C pin 0 as output (LED)
   call hardware_configure_gpio(GPIO_PORTC, 0, GPIO_MODE_OUTPUT, GPIO_OUTPUT_LOW)
end
```

2.5.2 if <expression> [else] end if

Conditions can be tested with the ***if*** command. Commands between ***if*** and ***end if*** will be executed if the ***<expression>*** is true.

```
if x<2

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

If ***else*** is used, then if the <*expression*> is true, commands between ***then*** and ***else*** will be executed. If the <*expression*> is false, commands between ***else*** and ***end if*** will be executed.

```
if x<2
    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 compare operator. This means that 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 need to be evaluated first.

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

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

2.5.3 while <expression> end while

Loops can be constructed using the ***while*** command. Command lines between ***while*** and ***end while*** will be executed when the ***<expression>*** is true.

```
a=0
while a<10
     a=a+1
end while
```

2.5.4 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 while response parameters are variable names into which response values will be stored. Response parentheses and parameters can be omitted if the response is not needed by your application.

Note: All response variables must be declared before use.

```
const FLASH_PS_KEY_AP_IPV4_ADDR = $8006
const ap_ipv4_addr =  192.168.1.1
# Store the default to PS since this key is not automatically updated by the stack.
 call flash_ps_save(FLASH_PS_KEY_AP_IPV4_ADDR, 4, ap_ipv4_addr)
```

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 that return values are not supported.

2.5.5 let <variable> = <expression>

The ***let*** command is an optional command which can be used to assign an expression to a variable.

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

2.5.6 return

This command causes a return from an event or a procedure.

```
event sme_interface_status(hw_interface, status)
  if hw_interface != 0
    return
  end if
    #action for interface 0
end
```

2.5.7 float (mantissa,exponent)

The float command can be used to change a given mantissa and exponent into a 32-bit base-10 IEEE 11073 SFLOAT value. Conversion is done using the following algorithm:

	Exponent	Mantissa
Length	8-bit	24-bit
Type	signed integer	signed integer

The list below defines 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`

2.5.8 memcpy(destination,source,length)

The ***memcpy*** command copies bytes from the defined source buffer to the defined destination buffer. The destination and source should not overlap.

Note: The buffer index notation only uses the start byte index and should not include the size portion.

Example: `dst(start)` instead of `dst(start:size)`.

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

2.5.9 memcmp(buffer1,buffer2,length)

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

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

2.5.10 memset(buffer , value , length)

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

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

2.6 Procedures

BGScript supports procedures which can be used to implement subroutines. Procedures differ from functions used in other programming languages, because they do not return a value and cannot be used in expressions.

Procedures are called using the ***call*** command, just like other BGScript commands.

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

Example: Printing the MAC Address of the Wi-Fi Module

```
dim n,j

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

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

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

event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
      call config_get_mac(0)
end

event config_mac_address(hw_interface, mac)
      call print_mac(6,mac(0:6))
end
```

2.7 Using Multiple BGscript files

In BGScript other BGScript files may be included in a project by using the *import* directive. Also variables and procedures from another BGScript file may be used by other BGScript files by using the *export* directive. These directives are explained in more detail in the following sections.

2.7.1 Import

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

main.bgs

```
import "other.bgs"

event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # wifi module has booted
end
```

2.7.2 Export

By default all code and data are local to each BGscript file.
The ***export*** directive allows use of variables and procedures in another script file, which imports the script file in which the variables and procedures are actually defined.

other.bgs

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

main.bgs

```
import "other.bgs"
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    call init_hex()
end
```

3. BGScript Limitations

There are some practical limitations to using BGScript which the user should keep in mind when deciding whether to realize an application using BGScript or whether an external host is perhaps needed. The main limitations to consider are described in the following sections.

3.1 32-bit Resolution

All values used in BGScript must fit into 32 bits. This affects, for example, definitions of very long timer intervals.

3.2 Performance

BGScript has limited performance (commands/operations per time unit), which might prevent some applications from being implemented using BGScript.

BGScript can typically execute commands/operations in the order of thousands of commands per second.

4. Practical BGScript Examples

This section contains a number of examples describing how various actions can be programmed using BGScript.

4.1 Basic examples

This section contains some very basic Wi-Fi BGScript examples.

4.1.1 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.

Example: Catching system start-up

```
# Boot event listener
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
     # System started - start Wi-Fi radio
     call sme_wifi_on()
end
```

4.1.2 Performing a System Reset

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

Example: System reset and firmware restart

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

4.1.3 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 the response status codes in order to keep the examples short and to improve readability of the examples. 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.

Note: In BGScript command response, the ***parameters*** refer to variables into which the values will be copied.
References to buffers are given using the `BUFFER(< expression >:< size >)` notation. Buffers need to contain enough space for the response data to prevent a buffer overflow from occurring.

Example: Command response handling

```
dim result

# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

Example: Command response handling with a buffer

```
dim value_data(4)

# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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, parallel execution of commands must be avoided. In other words, calling a new command while still waiting for an event from the previous command is not recommended.
Parallel commands may have unintended consequences and may or may not work.
The exception to this rule is the ***stop*** command used in connection with a previous ***start*** command. The ***stop*** command is always safe to call as long as the ***start*** command is still ongoing.

Example: Parallel commands may have unexpected results

```
# 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(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

4.2 Wi-Fi

This section shows simple BGScript code examples for handling Wi-Fi related events.

4.2.1 Catching a 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 that the module has established successfully a Wi-Fi connection.

Exampe: Catching a Wi-Fi connected event

```
dim connected

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

4.2.2 Catching a Wi-Fi Disconnection Event

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

The example below shows how to detect that the module has lost a Wi-Fi connection.

Example: Catching a disconnection of a Wi-Fi connection

```
dim connected

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

4.2.3 Catching a Failed Wi-Fi Connection Event

Sometimes the Wi-Fi connections to an Access Point fail, in which case an event is generated allowing the catching of a failed connection, which is indicated by the *sme_connect_failed* event.

The example below shows how to detect that a Wi-Fi connection between the Module and an Access Point was lost.

Example: Catching a fail of a Wi-Fi connection

```
# 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
```

4.2.4 Performing a Wi-Fi Scan

A Wi-Fi scan is initiated by calling the `sme_start_scan(...)` command. This command can only be issued when Wi- Fi has been enabled and the Module 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 in which the 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 the `sme_scan_results_sort_rssi(...)` command for requesting scan results sorted by signal strength.
The results are returned as events, and the Wi-Fi network with the strongest signal (highest RSSI value) is listed first.

Example: Performing a Wi-Fi scan

```
# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

Example: Sorting detected networks according to signal strength (RSSI)

```
# 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
```

4.2.5 Connecting to a Wi-Fi Network

Connecting to a Wi-Fi network can be done either by using `sme_connect_bssid(...)` or the `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 automatically selects the strongest Wi-Fi Access Point in case there are multiple possibilities.

Note: Connecting to a specific BSSID requires that a Wi-Fi scan has been performed before issuing the ***connect*** command.

Example: Connecting using BSSID command

```
dim connected
dim connect_bssid(6)

# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

Example: Connecting using a network name

```
dim connected
dim connect_ssid(32)

# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

4.2.6 Creating a Wi-Fi Access Point

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

Example: 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(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # AP parameters to use. Channel 11, no encryption, SSID "Silabs".
    ap_channel = 11
    ap_security = 0
    ap_ssid_len = 6
    ap_ssid(0:ap_ssid_len) = "Silabs"

    # 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
```

Example: 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(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # AP parameters: channel 11, WPA2 security, SSID "Silabs" and password "testtest".
    ap_channel = 11
    ap_security = 2
    ap_ssid_len = 6
    ap_ssid(0:ap_ssid_len) = "Silabs"
    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
```

4.2.7 Using Wi-Fi Protected Setup

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

Example: Using Wi-Fi Protected Setup with pushbutton method

```
# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

4.3 GPIO and Interfaces

This section contains basic BGScript examples on how to configure and use GPIO, interrupts, and different interfaces.

4.3.1 Configuring I/O Pin Modes and Interrupt Pins

The example below shows how to configure the Module's GPIO pins as outputs or inputs and how to set an input to act as an external interrupt pin.

In the example below, pin **PC0** of the Module, which in the WSTK Development Kit is connected to **LED0**, is configured as an output to enable control of the LED status (on or off). Pin **PD4** of the Module is configured as an external interrupt.

Example: Configuring I/O pin modes and interrupt pins

```
# Constants for GPIO configuration
const GPIO_PORTA = 0
const GPIO_PORTB = 1
const GPIO_PORTC = 2
const GPIO_PORTD = 3
const GPIO_PORTE = 4
const GPIO_PORTF = 5
const GPIO_PORTG = 6
const GPIO_MODE_DISABLED = 0
const GPIO_MODE_INPUT = 1
const GPIO_MODE_INPUT_PULL = 2
const GPIO_MODE_INPUT_PULL_FILTER = 3
const GPIO_MODE_OUTPUT = 4

const GPIO_FILTER_OFF = 0
const GPIO_FILTER_ON = 1
const GPIO_OUTPUT_LOW = 0
const GPIO_OUTPUT_HIGH = 1

const GPIO_TRIGGER_DISABLED = 0
const GPIO_TRIGGER_RISING = 1
const GPIO_TRIGGER_FALLING = 2
const GPIO_TRIGGER_BOTH = 3

# boot event listener
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
      # Config IO port direction - Set port C pin 0 as output (LED)
      call hardware_configure_gpio(GPIO_PORTC, 0, GPIO_MODE_OUTPUT, GPIO_OUTPUT_LOW)

      # Config IO port direction - Set port D pin 4 as input
      call hardware_configure_gpio(GPIO_PORTD, 4, GPIO_MODE_INPUT, GPIO_FILTER_ON)

      # Enable interrupts on pins PD4 (Push Button 0)
      call hardware_configure_gpio_interrupt(GPIO_PORTD, 4, GPIO_TRIGGER_FALLING)
      b=0
end

# IO interrupt listener
event hardware_interrupt(interrupts, timestamp)
      # PD4 is interrupt 4 which bit-mask is binary 10000 and decimal 16
      if (interrupts & 16)
            if (b = 0)
                  # IO port write - Turn ON LED1 (PC0)
                  call hardware_write_gpio(GPIO_PORTC, PIN_PC0, PIN_PC0)
                  b=1
            else
                  # IO port write - Turn OFF LED1 (PC0)
                  call hardware_write_gpio(GPIO_PORTC, PIN_PC0, $0000)
                  b=0
            end if
      end if
end
```

4.3.2 I2C

The Module automatically handles I2C stretching. Repeated starts are also created when starting the I2C write or read without stopping the last transfer.

In the examples below, data is read from and data is written to an external EEPROM memory with I2C interface.

Example: I2C EEPROM write

```
#Start write sequence to EEPROM at I2C address 0x50.
call i2c_start_write(I2C_SLAVE_CHANNEL, I2C_SLAVE_ADDRESS, 2, $50)

#EEPROM requires a 2 byte address to write to.
call i2c_start_write(I2C_SLAVE_CHANNEL, I2C_SLAVE_ADDRESS, 2, "\x00\x00")

#Write data to EEPROM
call i2c_start_write(I2C_SLAVE_CHANNEL, I2C_SLAVE_ADDRESS, 13, "Hello, World!")

#Stop write sequence
call i2c_stop(I2C_SLAVE_CHANNEL)
```

Example: I2C EEPROM read

```
#Start read sequence from EEPROM at I2C address 0x50.
call i2c_start_write(I2C_SLAVE_CHANNEL, I2C_SLAVE_ADDRESS, 2, $50)

#EEPROM requires a 2 byte address to read from.
call i2c_start_write(I2C_SLAVE_CHANNEL, I2C_SLAVE_ADDRESS, 2, "\x00\x00")

#Stop write sequence
call i2c_stop(I2C_SLAVE_CHANNEL)

#Read data from EEPROM
call i2c_start_read(I2C_SLAVE_CHANNEL, I2C_SLAVE_ADDRESS, 2)(result, value_len, value(0:cmd_value_len))

#Stop read sequence
call i2c_stop(I2C_SLAVE_CHANNEL)
```

4.3.3 RTC

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

Example: Initialize RTC alarm and configure generation of alarms

```
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(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # Initialize RTC
    call hardware_rtc_init(1)

    # configure the starting time for RTC
    call hardware_rtc_set_time(2015,12,01,01,00,00)

    # configure the alarm time for RTC
    call hardware_rtc_set_alarm(2015,12,01,02,00,00)
 end

#RTC alarm raised
event hardware_rtc_alarm()
    call hardware_rtc_get_time()(result,year,month,day,weekday,hour,minute,second)
end
```

4.3.4 ADC

ADC reading is done with *hardware_adc_read*(..)(..), whose response contains the value read from defined ADC channel. The reference source of ADC is defined in project configuration.

Example to read ADC value and write it to a file of SD card

```
#Script to save ADC values in the file

const TIMER_MEASUREMENT = 0
const MEASUREMENT_INTERVAL = 1000
const ADC_CHANNEL

dim result
dim input
dim value
dim filename(32)
dim filename_len
dim file_handle

dim adc_val(32)
dim adc_val_len

dim measure_cnt

event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    measure_cnt = 0
    filename_len = 7
    filename(0:filename_len) = "adc.txt"

    #Open file for writing
    call sdhc_fopen($02,filename_len,filename(0:filename_len))(result)
    if(result)
       call sdhc_fopen($0a, filename_len, filename(0:filename_len))
    end if
    # Raises event sdhc_ffile

    #Start timer to 1 second interval
    call hardware_set_soft_timer(MEASUREMENT_INTERVAL, TIMER_MEASUREMENT, 0)
    #Raises event hardware_soft_timer
end


#Event carries file information
event sdhc_ffile(fhandle, fsize, fattrib, fname_len, fname_data)
    file_handle= fhandle
end


#Raised when SDHC operation ready
event sdhc_ready(fhandle, operation, res)
    measure_cnt = measure_cnt+1
    if measure_cnt > 10
        call hardware_set_soft_timer(0, TIMER_MEASUREMENT, 0)
        call sdhc_fclose(file_handle)
    end if
end


#Event raised when timer trigged
event hardware_soft_timer(handle)
    #Read ADC
    call hardware_adc_read(ADC_CHANNEL)(result,input, value)

    #Write as string
    call util_itoa(value)(adc_val_len,adc_val(0:adc_val_len))
    call sdhc_fwrite(file_handle,adc_val_len, adc_val(:))
    # Raises event sdhc_ready
end
```

4.4 Timers

This section describes how to use timers with BGScript.

4.4.1 Continous and Single Interrupt Timers

The examples below show how to generate continuous and single timer interrupts.

Example: Generating continuous timer interrupts

```
dim count
# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    #init timer
    call hardware_timer_init(0,0,1,0)
    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
```

Example: Generating a single timer interrupt

```
# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    #init timer
    call hardware_timer_init( 0,0,1,1)
end

# Event received when a timer is triggered.
event hardware_soft_timer(handle)
    if handle = 0
        # Timer 0 has been triggered.
    end if
end
```

4.5 Endpoints

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

4.5.1 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 the `endpoint_set_streaming(...)` command, it's typically set in the *projext.xml / hardware.xml* files.

When operating in streaming mode, incoming UART data is discarded by default. The endpoint data is routed to a destination which can be set the `endpoint_set_streaming_destination(...)` command.

Note: The following configurations need to be in the *project.xml / hardware.xml* file to enable the UART interface(s) and to allow BGScript to access it.

Example: Enabling UART interfaces in hardware.xml file to allow BGScript access

```
<hardware>
  ...
  <uart channel="0" baud="115200" api="true" handshake="true" />
  <uart channel="1" baud="115200" api="false" handshake="true" />
</hardware>
```

Example: Writing to a UART endpoint

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

Example: Setting the UART endpoint mode

```
# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

Example: Changing a UART to streaming mode

```
# PREREQUISITE: A TCP connection is active and the TCP endpoint is stored in
# variable "client_endpoint".

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)
```

4.5.2 USB Endpoint

A 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 instead must be set in the *project.xml / hardware.xml* file.

When operating in streaming mode, incoming USB data is discarded by default. The endpoint data is routed to a destination which can be set using the `endpoint_set_streaming_destination(...)` command.

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

Example: Enabling UBS interface in hardware.xml file to allow BGScript access

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

Example: Writing to a USB endpoint

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

Example: Changing the USB endpoint to streaming mode

```
# PREREQUISITE: A TCP connection is active and the TCP endpoint is stored in
# variable "client_endpoint".

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)
...
```

4.5.3 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 the `endpoint_set_streaming_destination(...)` command.

Example: Dropping an endpoint

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

4.6 PS Store

This section contains examples on how to use PS keys with BGScript.

PS Store is a module internal nonvolatile memory (NVM) data storage structure, with size of 4 kB, and it is duplicated to avoid data lost. The structure of storage is key – value pairs, where key is unique identifier to data.

4.6.1 Reading PS Keys

The examples below show how to read PS keys.

Example: Reading the value of a single PS key

```
dim result
dim value_len
dim value_data(4)

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

Example: Reading the value of single PS key with an enumerated index

```
dim result
dim value_len
dim value_data(4)

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

Example: Reading the value of all PS keys

```
# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

4.6.2 Writing PS Keys

This example shows how to write data into a single PS key.

Example: Writing data into a single PS key

```
dim value_len
dim value_data(6)

# Event received when the system has been successfully started up.
event system_boot(hw, bootloader_version, major, minor, build, revision_len, revision_data)
    # 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
```

4.6.3 Deleting PS Keys

This example shows how to delete a single PS key.

Example: Deleting a single PS key

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

4.7 TCP/IP

This section contains BGScript examples on how to use the built-in TCP/IP stack.

4.7.1 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.

Example: 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
```

Example: 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
```

Example: 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
```

Example: 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
```

4.7.2 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 is 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*** parameter to the `tcpip_start_tcp_server(...)` command. It is possible to change the routing by calling the `endpoint_set_streaming_destination(...)` command.

Example: 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
```

Eaxmple: 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
```

Example: 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
```

Example: 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
```

Example: 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
```

4.7.3 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 a UDP connection is assigned a random source port. If the source port needs to be changed the command `tcpip_udp_bind(...)` can be used.

Example: Creating a 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
```

Example: Sending UDP data

```
# PREREQUISITE: An UDP connection is active.

dim client_endpoint

...
# Send data to the server.
call endpoint_send(client_endpoint,5,"Hello")
...
```

Example: Closing a 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
```

Example: Changing the source port of a 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)
...
```

4.7.4 UDP Server

A 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.

Example: Starting a 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
```

Example: 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
```

Example: Stopping 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
```

4.7.5 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.

Example: Resolving a DNS name to IP address

```
...
# Query the IP address of "silabs.com".
call tcpip_dns_gethostbyname(10, "silabs.com")
...
# Event called when a DNS resolver query completes.
event tcpip_dns_gethostbyname_result(result, address, name_len, name_data)
    # DNS name resolved.
end
```

4.8 SDHC Card Reader

This section contains examples describing how to read and write from and to a microSD card using BGScript.

4.8.1 Read

To read files from the SD Card, the file must be opened in *read* (0x01) or *read/write* (0x03) mode. In the BGScript, the maximum size of one data block for a single read operation is 256 bytes.

Example: Read all data of a file using BGScript

```
.....
data_left = 0

# Open file for example in boot event or after file listing
call sdhc_fopen($01,filename_len,filename(0:filename_len))(result)
# Raises sdhc_ffile event
if result
    #Process file open error
end if
....


# Event contains file information
event sdhc_ffile(fhandle, fsize, fattrib, fname_len, fname_data)
    # Opened file will have unique handle
    if fhandle != 255
        file_handle = fhandle
        # Read data
        call sdhc_fread(fhandle,128)
        # Raises event sdhc_fdata
    end if
end

# Event contains file data
event sdhc_fdata(fhandle, data_len, data_data)
    if fhandle = file_handle
        data_left = 1
        # Process data
    end if
end

# Event raised when sdhc operation ready
event sdhc_ready(fhandle, operation, result)
    if operation = FILE_READ && fhandle = file_handle
        if data_left
        # Data left, read more
            call sdhc_fread(fhandle,128)
            data_left = 0
        else
            # No more data, close the file
            call sdhc_fclose(fhandle)
        end if
    end if
end
```

4.8.2 Write

The file to be written into must first be opened in *write* (0x02), *read/write* (0x03) or *create new* (0x0a) mode. In BGScript, the maximum size of a data block for a single write is 256 bytes.

Example: Writing data into a file using BGScript

```
.....
# Open file for example in boot event
call sdhc_fopen($02,filename_len,filename(0:filename_len))(result)
if(result)
    # File not exist, create new
    call sdhc_fopen($0a, filename_len, filename(0:filename_len))
    # Raises event sdhc_ffile
end if
.....

# Event contains file information
event sdhc_ffile(fhandle, fsize, fattrib, fname_len, fname_data)
  if fhandle != 255
     txt_len = 9
     txt(0:txt_len) = "test data"
     call sdhc_fwrite(fhandle, txt_len,txt(:))
     # Raises event sdhc_ready
  end if
end

# Event raised when sdhc operation ready
event sdhc_ready(fhandle, operation, result)
    if operation = FILE_WRITE && fhandle = file_handle
         call sdhc_fclose(fhandle)
    end if
end
```

5. Revision History

5.1 Revision 1.1

May 23rd, 2016

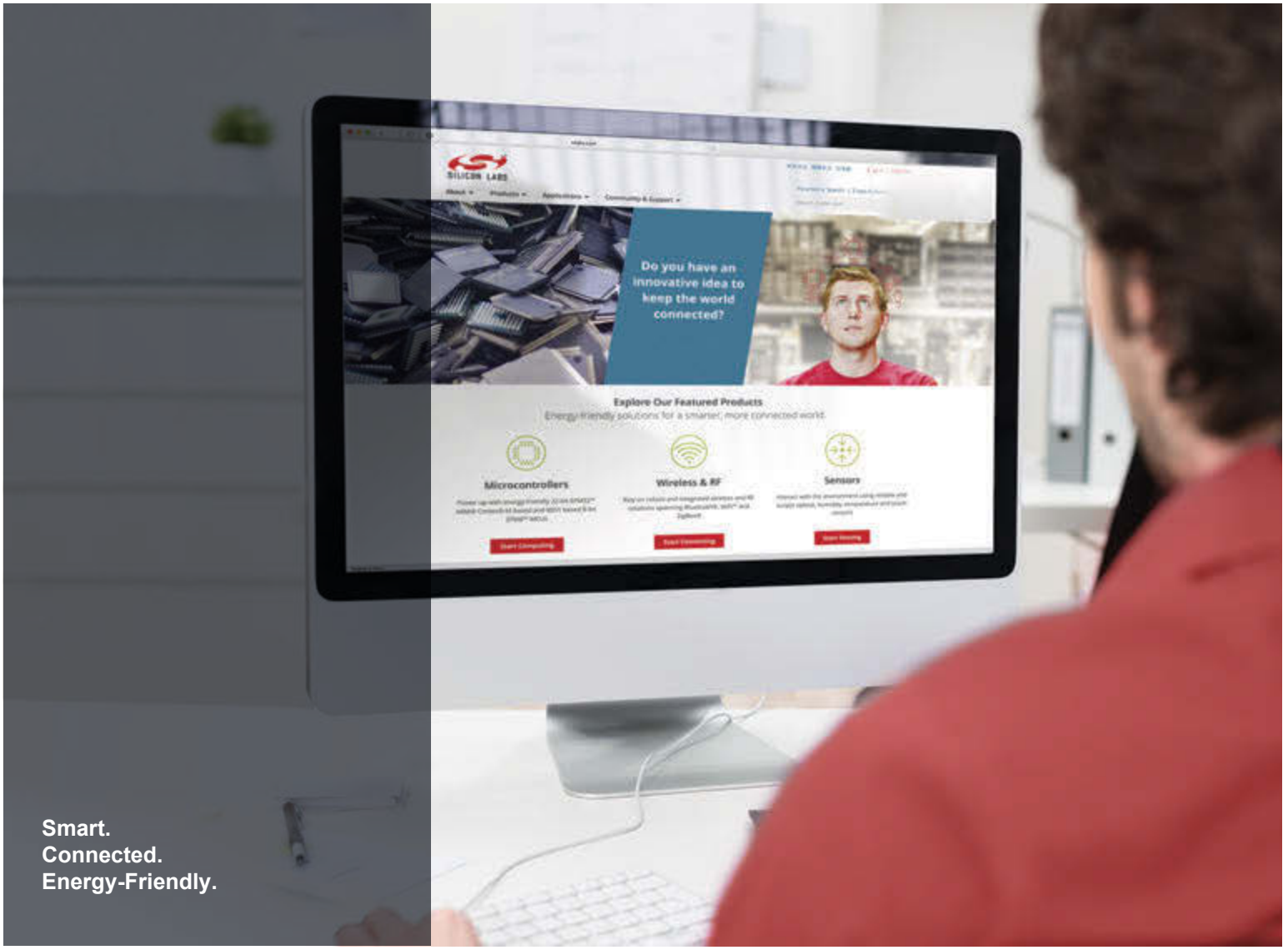
Changed: Some graphic files edited. Code examples written as text and screenshots removed.

5.2 Revision 1.0

February 22nd, 2016

Initial release.

Table of Contents



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

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

http://www.silabs.com