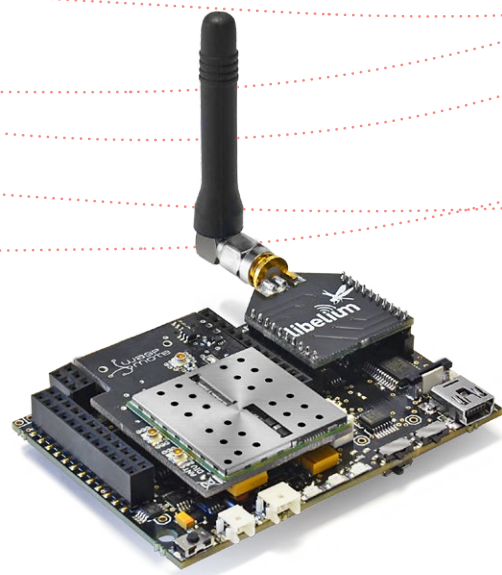
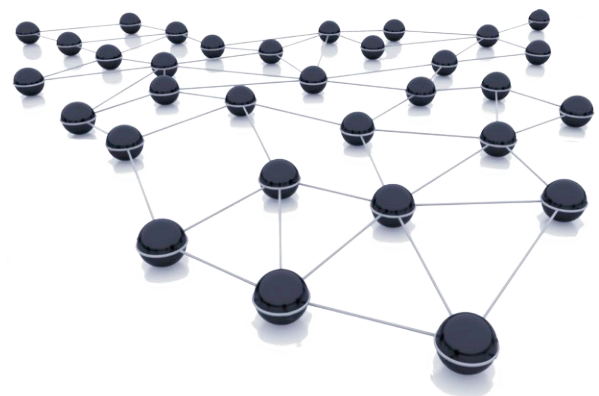
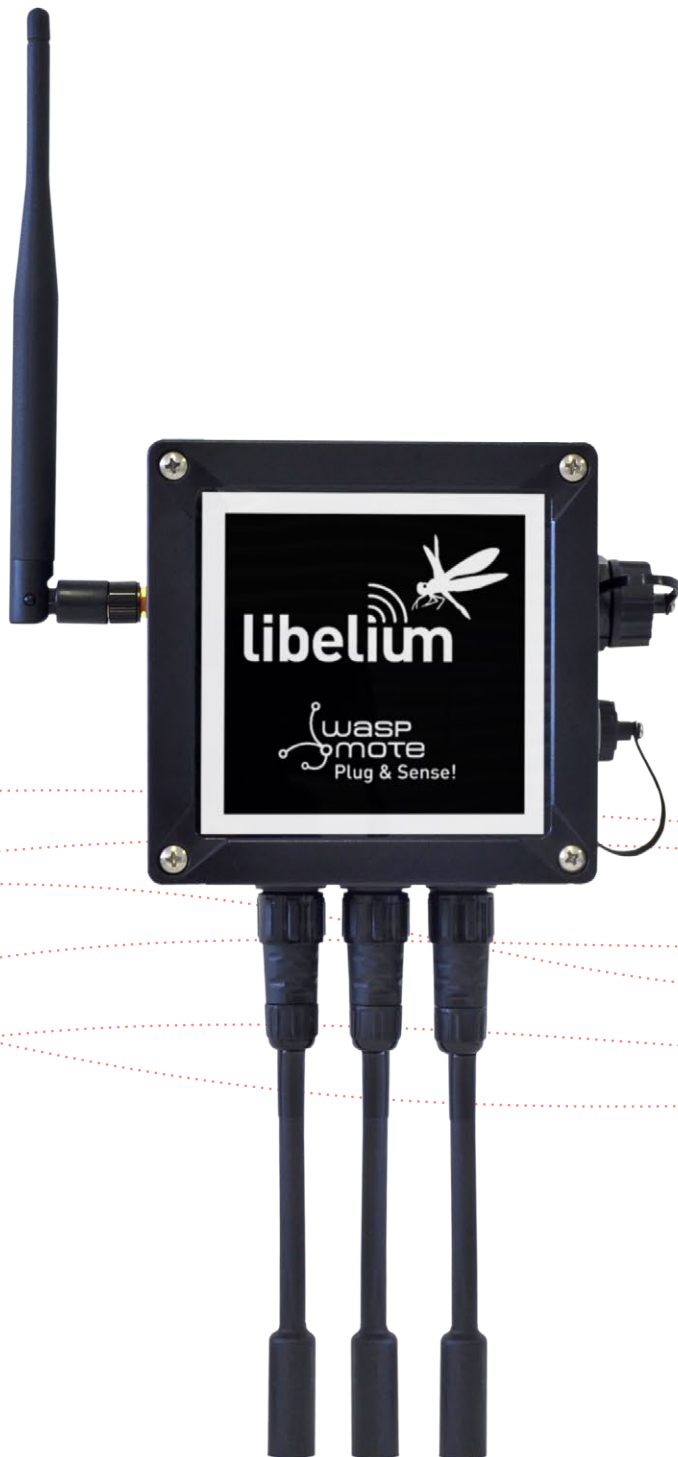


Wasmote Utilities Programming Guide



Document Version: v7.0 - 02/2017

© Libelium Comunicaciones Distribuidas S.L.

INDEX

1. Introduction	4
2. USB Library.....	4
2.1. Waspote Libraries.....	4
2.1.1. Waspote USB Files	4
2.1.2. Constructor.....	4
2.1.3. Pre-Defined Constants	4
2.2. Initialization	4
2.3. Reading data.....	5
2.4. Printing data	5
3. Utilities Library	7
3.1. Waspote Libraries.....	7
3.1.1. Waspote Utilities Files	7
3.1.2. Constructor.....	7
3.1.3. Pre-Defined Constants	7
3.2. Using LEDs.....	7
3.2.1. Change the state of the LEDs.....	8
3.2.2. Blink the LEDs	8
3.3. Using EEPROM.....	9
3.4. Reading Serial ID	9
3.5. Converting types.....	9
4. Input/Output pins.....	11
4.1. Reading ANALOG inputs	12
4.2. Digital I/O	12
4.2.1. Setting DIGITAL pin mode	12
4.2.2. Reading DIGITAL Inputs	12
4.2.3. Writing DIGITAL Outputs	12
4.2.4. DIGITAL1 pin as PWM output	13
5. Getting Free RAM	14
5.1. Waspote Files	14
5.2. freeMemory() function.....	14
6. Reading basic sensors	15
6.1. DS18B20 Temperature Sensor.....	15
7. Waspote Stack EEPROM	16
7.1. Waspote Files	16

7.2. Constructor	16
7.3. Pre-Defined Constants	16
7.4. Initializing the stack	17
7.5. Initializing the block size	17
7.6. Push	18
7.7. Pop	18
8. Code examples and extended information	19
9. API changelog	20

1. Introduction

This guide explains the a set of libraries. These libraries were designed for our original product lines Waspote v12 and Plug & Sense! v12, and has evolved for the new lines Waspote v15 and Plug & Sense! v15, released on October 2016.

These libraries are compatible with Waspote v12 and v15 and Plug & Sense! v12 and v15. This guide is oriented to v15 users so if you are using v12 versions of our products, please read the corresponding guides, available on our [Development website](#).

You can get more information about the generation change on the document "[New generation of Libelium product lines](#)".

Waspote provides some libraries to manage USB interaction and other general tasks (blinking LEDs, some string conversions, sensor reading, EEPROM use, etc).

To manage these tasks, some functions have been developed and two libraries have been created: 'USB Library' and 'Utils Library'.

2. USB Library

2.1. Waspote Libraries

2.1.1. Waspote USB Files

WaspUSB.h ; WaspUSB.cpp

2.1.2. Constructor

To start using Waspote USB library, an object from class 'WaspUSB' must be created. This object, called 'USB', is created inside the Waspote [USB](#) library and it is public to all libraries. It is used through the guide to show how the Waspote USB library works.

When creating this constructor, one variable is initialized. This variable specifies the number of the UART that USB is going to use (UART0 in this case).

2.1.3. Pre-Defined Constants

There are some constants defined in 'WaspUSB.h' related with the different kind of numbers that can be printed on the screen.

2.2. Initialization

Two functions have been created to open and close the UART used to communicate via USB.

Example of use:

```
{
  USB.ON(); // Opens the UART at 115200 bps by default
  USB.OFF(); // Closes the UART
}
```

2.3. Reading data

Two functions have been developed for reading data or checking if data is available in the UART. One more function has been developed to free the UART buffer.

Example of use:

```
{
  int data_read=0;
  if(USB.available())
  {
    // If data is available '1' is returned
    data_read=USB.read(); // Reads data from UART0
  }
  USB.flush(); // Frees the UART buffer. All the data unread are lost.
}
```

2.4. Printing data

Some functions have been created to write data to the UART.

Example of use:

Print a **character**

```
{
  char charac='a';
  USB.print(charac); // Writes the char 'a' to the UART
  USB.println(charac); // Writes the char 'a' to the UART adding an EOL
  USB.println(); // Writes an EOL
}
```

Print a **string**

```
{
  char string[] = "Hello";
  USB.print(string); // Writes a string to the UART
  USB.println(string); // Writes a string to the UART adding an EOL
}
```

Print **uint8_t** (1 byte)

```
{
  uint8_t unsigned=3;
  USB.print(unsigned); // Writes the number '3' to the UART
  USB.println(unsigned); // Writes the number '3' to the UART adding an EOL
}
```

Print **int** (2 bytes)

```
{
  int integer=54345;
  USB.print(integer); // Writes the number '54345' to the UART
  USB.println(integer); // Writes the number '54345' to the UART adding an EOL
}
```

Print **long** (4 bytes)

```
{
  long long_int=1234567;
  USB.print(long_int); // Writes the number '1234567' to the UART
  USB.println(long_int); // Writes the number '1234567' to the UART adding an EOL
}
```

Print **float** (4 bytes)

```
{
  float float_num=1.23456;
  USB.print(float_num); // Writes the number '1.23456' to the UART
  USB.println(float_num); // Writes the number '1.23456' to the UART adding an EOL
}
```

 Print **hexadecimal number** (1 byte)

```
{
  uint8_t num = 0x14;
  USB.printHex(num); // Writes the number 0x14 in base 16
}
```

 Print **arrays**:

```
{
  uint8_t array[5] = { 0x31, 0x32, 0x33, 0x34, 0x35 };
  USB.print( array, 5 );
  USB.println( array, 5 ); // adding EOL
}
```

 Print **arrays in HEX format**:

```
{
  uint8_t array[5] = { 0x31, 0x32, 0x33, 0x34, 0x35 };
  USB.printHex( array, 5 );
  USB.printHexln( array, 5 ); // adding EOL
}
```

 Print **numbers** in different formats

```
{
  USB.println(78); // Writes 78 in integer format: '78'
  USB.println(78, BIN); // Writes 78 in binary format: '1001110'
  USB.println(78, OCT); // Writes 78 in octal format: '116'
  USB.println(78, DEC); // Writes 78 in integer format: '78'
  USB.println(78, HEX); // Writes 78 in hexadecimal format: '4E'
}
```

 Print **formatted data**

```
{
  int var1=0xABCD;
  int var2=3.1416;
  int var3=32767; //max signed int
  int var4=32768; //overflows signed int range
  long var5=2147483647; //max signed long int
  long var6=2147483648; //overflows signed long range
  unsigned long var7=4294967295; //max unsigned long

  USB.printf("%s\n", "Hello world"); // Writes "Hello World" with EOL
  USB.printf("millis: %lu\n", millis()); // Writes "millis: 356" with EOL
  USB.printf("hexadecimal: %x\n", var1); // Writes "hexadecimal: abcd" with EOL
  USB.printf("decimal: %d\n", var1); // Writes "decimal: -21555" with EOL
  USB.printf("unsigned int: %u\n", var1); // Writes "unsigned int: 43981" with EOL
  USB.printf("float: %f\n", var2); // It is not possible to print floats this way
  USB.printf("int: %d\n", var3); // Writes "int: 32767" with EOL
  USB.printf("'32768' overflows: %d\n", var4); // Writes "'32768' overflows: -32768"
  USB.printf("signed long: %ld\n", var5); // Writes "signed long: 2147483647" with EOL
  USB.printf("'2147483648' overflows: %ld\n", var6); // Writes "'2147483648' overflows: -2147483648"
  USB.printf("unsigned long: %lu\n", var7); // Writes "unsigned long: 4294967295"
}
```

- USB function example:

<http://www.libelium.com/development/waspmote/examples/usb-01-usb-functions>

- Formatted print function example:

<http://www.libelium.com/development/waspmote/examples/usb-02-usb-printf-function>

3. Utilities Library

3.1. Waspote Libraries

3.1.1. Waspote Utilities Files

WaspUtils.h ; WaspUtils.cpp

3.1.2. Constructor

To start using the Waspote Utilities library, an object from class `'WaspUtils'` must be created. This object, called `Utils`, is created inside the Waspote Utilities library and it is public to all libraries. It is used through the guide to show how the Waspote Utilities library works.

When creating this constructor, no variables are initialized.

3.1.3. Pre-Defined Constants

There are some constants defined in `'WaspUtils.h'` used to make it easier the understanding of the code.

3.2. Using LEDs

These functions are capable of changing the state of the LEDs. There are two programmable LEDs in Waspote: a green LED and a red LED. `LED1` refers to the green LED. `LED0` refers to the red LED. It is possible to change their state, to get their state and to blink both LEDs for a specific time.

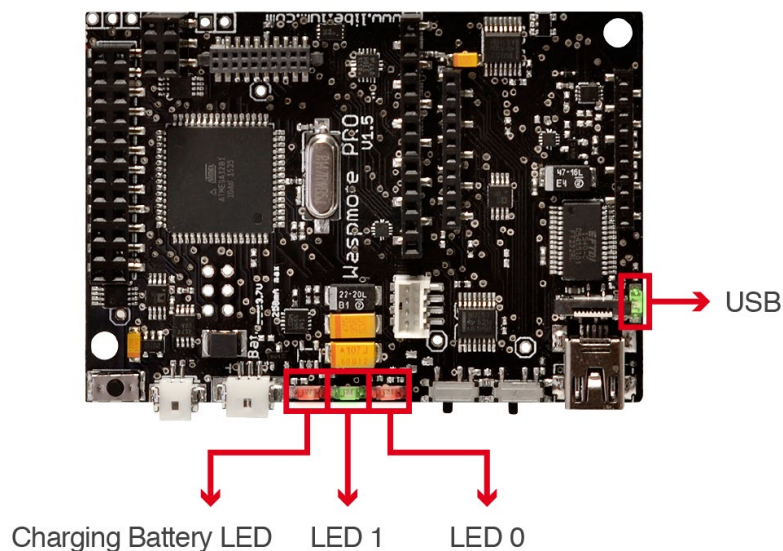


Figure : Programmable LEDs in Waspote

3.2.1. Change the state of the LEDs

The function `setLED()` changes the state of the LEDs. It is necessary to indicate two different inputs: the LED which is set and the state to be set.

Example of use:

```
{
  // 1. set LEDs ON
  Utils.setLED( LED1, LED_ON); // Sets the green LED ON
  Utils.setLED( LED0, LED_ON); // Sets the red LED ON

  // 2. set LEDs OFF
  Utils.setLED( LED1, LED_OFF); // Sets the green LED OFF
  Utils.setLED( LED0, LED_OFF); // Sets the red LED OFF

  // 3. Get LEDs state
  uint8_t state1=Utils.getLED(LED1); // Gets the state of LED1
  uint8_t state0=Utils.getLED(LED0); // Gets the state of LED0

  // 4. Blink LEDs
  Utils.blinkLEDs(1000); // Blink LEDs using a delay of 1000ms for blinking
}
```

• Using LEDs example:

<http://www.libelium.com/development/waspmote/examples/ut-02-using-leds>

3.2.2. Blink the LEDs

The function `blinkLEDs()` blinks both LEDs once using the time input specified as argument in this function in milliseconds units.

Example of use:

```
{
  Utils.blinkLEDs(1000); // Blink both LEDs during 1000 ms
}
```

The function `blinkRedLED()` blinks the **red LED** once during 200 ms. It is possible to modify the number of times it blinks and the amount of time used in each blinking.

Example of use:

```
{
  Utils.blinkRedLED();           // blink once
  Utils.blinkRedLED(500);       // blink once during 500 ms
  Utils.blinkRedLED(500, 3);    // blink 3 times during 500 ms each time
}
```

The function `blinkGreenLED()` blinks the **green LED** once during 200 ms. It is possible to modify the number of times it blinks and the amount of time used in each blinking.

Example of use:

```
{
  Utils.blinkGreenLED();         // blink once
  Utils.blinkGreenLED(500);     // blink once during 500 ms
  Utils.blinkGreenLED(500, 3);  // blink 3 times during 500 ms each time
}
```

• Using LEDs example:

<http://www.libelium.com/development/waspmote/examples/ut-02-using-leds>

3.3. Using EEPROM

This is the microcontroller's EEPROM (4 KB) non-volatile memory. EEPROM addresses from 0 to 1023 are reserved by Waspote API to save important data, so they must not be over-written. Thus, the available storage addresses go from 1024 to 4095.

RESERVED		AVAILABLE	
0	1023	1024	4095

Figure: EEPROM availability overview

The function that writes the EEPROM is `Utils.writeEEPROM()`. This function does not permit to write reserved EEPROM addresses.

Example of use:

```
{
  Utils.writeEEPROM(1024, 'B'); // Write the value 'B' in the address 1024
  uint8_t data=Utils.readEEPROM(1024); // Reads the value stored in the address 1024
}
```

• Using EEPROM example:

<http://www.libelium.com/development/waspote/examples/ut-01-using-EEPROM>

3.4. Reading Serial ID

This function reads the Waspote unique serial identifier. This identifier is composed by 8 bytes.

Example of use:

```
{
  unsigned long id = 0;
  id = Utils.readSerialID();
}
```

• Using EEPROM example:

<http://www.libelium.com/development/waspote/examples/ut-03-reading-serial-id>

3.5. Converting types

Convert from **long int** to **string**:

```
{
  char number[20];
  Utils.long2array(1356, number); // Gets the number '1356' into the string
}
```

Convert from **float** to **string**:

```
{
  char number[20];
  Utils.float2String(134.54342, number, 3); // Convert 134.54342 to string (3 decimals)
}
```

AVR Libc Library allows the user to convert between different variable types. This is a list with some supported function prototypes:

Convert **string** to **int** (2 bytes):

```
{  
  int number = atoi("2341");  
}
```

Convert from **string** to long **integer** (4 bytes):

```
{  
  long int number = atol("143413");  
}
```

• Converting types example:

<http://www.libelium.com/development/waspmote/examples/ut-04-convert-types>

4. Input/Output pins

Wasp mote can communicate with other external devices using different input/output ports.

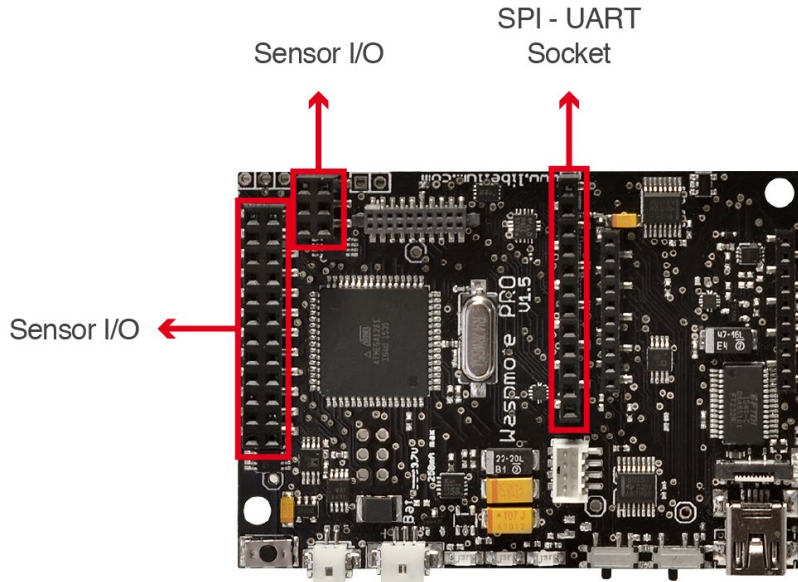


Figure : I/O sockets in Wasp mote

ANALOG	■ ■	3V3 SENSOR POWER
DIGITAL 8	■ ■	GND
DIGITAL 6	■ ■	DIGITAL 7
DIGITAL 4	■ ■	DIGITAL 5
DIGITAL 2	■ ■	DIGITAL 3
RESERVED	■ ■	DIGITAL 1
ANALOG 6	■ ■	ANALOG 7
ANALOG 4	■ ■	ANALOG 5
ANALOG 2	■ ■	ANALOG 3
3V3 SENSOR POWER	■ ■	ANALOG 1
GPS POWER	■ ■	5V SENSOR POWER
SDA	■ ■	SCL

Figure : Description of sensor socket pins

GND	■ ■	GND
ANALOG 6	■ ■	ANALOG 7
3V3 SENSOR	■ ■	3V3 SENSOR

Figure : Description of sensor socket pins

4.1. Reading ANALOG inputs

It gets the value read by the corresponding analog input. Wasmote has 7 analog inputs in the sensor connector. Each input is directly connected to the microcontroller. The microcontroller uses a 10-bit analog to digital converter (ADC). The reference voltage value for the inputs is 0 V (GND). The maximum value of input voltage is 3.3 V. To obtain input values, the function `analogRead()` is used. The value read from this function will be an integer number between 0 and 1023 bits, where 0 corresponds to 0 V and 1023 to 3.3 V.

Example of use :

```
{
  int val1 = analogRead(ANALOG1); // Read the input ANALOG1 and store its value in val1
  int val2 = analogRead(ANALOG2); // Read the input ANALOG2 and store its value in val2
  int val3 = analogRead(ANALOG3); // Read the input ANALOG3 and store its value in val3
  int val4 = analogRead(ANALOG4); // Read the input ANALOG4 and store its value in val4
  int val5 = analogRead(ANALOG5); // Read the input ANALOG5 and store its value in val5
  int val6 = analogRead(ANALOG6); // Read the input ANALOG6 and store its value in val6
  int val7 = analogRead(ANALOG7); // Read the input ANALOG7 and store its value in val7
}
```

4.2. Digital I/O

4.2.1. Setting DIGITAL pin mode

The `pinMode()` function configures the specified pin as an input or an output.

It returns nothing.

Example of use :

```
{
  pinMode(DIGITAL1, INPUT); // Sets DIGITAL1 as an input
  pinMode(DIGITAL4, OUTPUT); // Sets DIGITAL4 as an output
}
```

4.2.2. Reading DIGITAL Inputs

The `digitalRead()` function reads the value from the specified digital pin.

It returns '0' or '1'.

Example of use:

```
{
  int val=0;
  pinMode(DIGITAL1, INPUT); // Sets DIGITAL1 as an input
  val=digitalRead(DIGITAL1); // Reads the value from Digital 1
}
```

4.2.3. Writing DIGITAL Outputs

The `digitalWrite()` function writes a **HIGH** or **LOW** value to a digital pin. The pin voltage will be set to 3.3 V for a 'HIGH' value, and 0 V for a 'LOW' value.

Example of use:

```
{
  pinMode(DIGITAL4, OUTPUT); // Sets DIGITAL4 as an output
  digitalWrite(DIGITAL4, HIGH); // Writes 'High' to Digital 4
}
```

4.2.4. DIGITAL1 pin as PWM output

There is one pin that can be used as an analog or digital pin.

This pin is called DIGITAL1.

It returns nothing.

Example of use:

```
{  
  analogWrite(DIGITAL1, 128); // Writes a value comprised between [0-255] in DIGITAL1  
}
```

5. Getting Free RAM

5.1. Waspote Files

Inside the Core directory we can find both MemoryFree.cpp and MemoryFree.h which develop the function that allows the user to get the available free memory in Waspote's RAM.

5.2. freeMemory() function

This is the function that returns the Waspote's available memory in bytes.

Waspote RAM				
.data Variables	.bss Variables	Heap →	Free Memory	← Stack

Figure : RAM Memory spaces

The different sections in RAM are explained below:

- **.data variables/.bss variables:** Memory space for initialized and uninitialized variables.
- **Heap:** Memory space for dynamic memory allocation.
- **Stack:** Memory space used for calling subroutines and storing local (automatic) variables. This memory space grows downwards.

Example of use:

```
{
  USB.print("free Memory (Bytes):");
  USB.println(freeMemory()); // prints free RAM
}
```

- Getting free memory example:

<http://www.libelium.com/development/waspote/examples/ut-05-get-free-memory>

6. Reading basic sensors

There are some basic sensors that can be plugged directly to Waspote except for the light sensor which needs some hardware modifications.

6.1. DS18B20 Temperature Sensor

This temperature sensor also needs a simple hardware modification to be directly plugged to Waspote. Three pins come out of the sensor: Red (3V3), Black (GND) and White (Output pin). A 4K7 resistor must be connected between the red and white pins. The sensor output can be plugged in any Waspote digital pin. When reading the sensor the output pin is selected as input parameter:

Example of use:

```
{  
  // temperature sensor plugged to DIGITAL8 pin  
  float temp;  
  temp = Utils.readTempDS1820(DIGITAL8);  
}
```

• Reading DS18B20 temperature sensor example:

<http://www.libelium.com/development/waspote/examples/bs-05-reading-ds18b20>

7. Waspote Stack EEPROM

The class WaspStackEEPROM has been developed to allow saving frames into the EEPROM memory instead of the SD card.

There two types of stack configuration:

- **LIFO** stack (last input, first output). This means that when we extract data, the most recently stored information will be provided first.
- **FIFO** stack (first input, first output). In this case, data is extracted in the same order as they were inserted inside the stack.

The user may find it interesting for storing, for example, frames which could not be delivered: if even after several retries, the communication was not successful, we can store data in the stack in order to send it next time Waspote wakes up.

Another use would be to store frames which we want to send in the future; this is great for decreasing the use of the communication module, and saving battery.

A complete example of the functions:

www.libelium.com/development/waspote/examples/ut-06-stack-eprom-basic-operation

7.1. Waspote Files

WaspStackEEPROM.h ; WaspStackEEPROM.cpp

7.2. Constructor

To start using the Waspote StackEEPROM library, an object from class WaspStackEEPROM must be created. This object, called `stack`, is created inside the Waspote StackEEPROM library.

When calling this constructor, the `block_size` variable is initialized to `BLOCK_SIZE` constant. Also, `_mode` variable is set to the default stack mode `LIFO_MODE`.

7.3. Pre-Defined Constants

There are some constants defined in 'WaspStackEEPROM.h' used to limit the stack area and the size of each block inside the stack.

`START_STACK` → First position of the stack into the EEPROM. Must be greater than 1023.

`END_STACK` → Last position of the stack into the EEPROM. Max value is 4095.

`BLOCK_SIZE` → It fixes the size of each block inside the EEPROM. Max value is 255

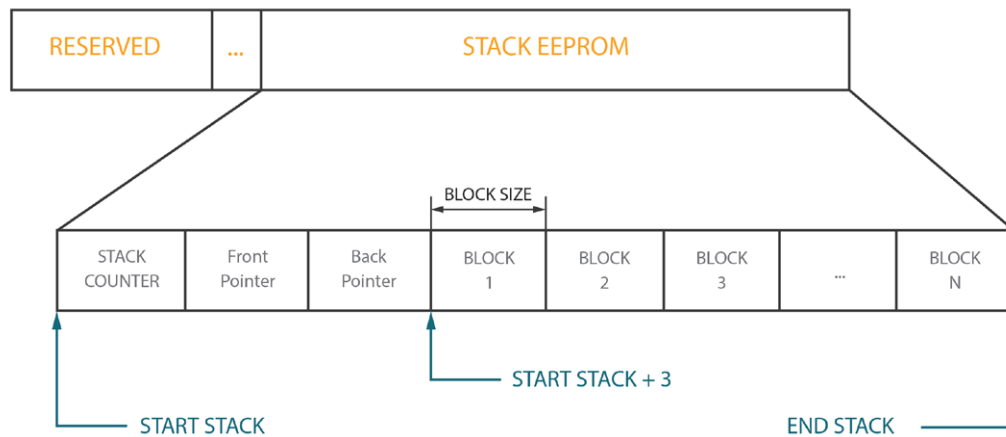


Figure : EEPROM memory map with stack detail

7.4. Initializing the stack

This function cleans the stack area writing 0xFF in each memory position. Also, it initializes the stack counter to 0 and the Front Pointer and Back Pointer of the stack. It is possible to choose the stack mode as input: `LIFO_MODE` or `FIFO_MODE`. By default, LIFO configuration is used. It returns '1' if success and '0' if error.

Example of use:

```
{
// Possibility 1: Initialize stack (LIFO mode by default)
stack.initStack();

// Possibility 2: Initialize with LIFO mode
stack.initStack( LIFO_MODE );

// Possibility 3: Initialize with FIFO mode
stack.initStack( FIFO_MODE );
}
```

It returns nothing.

7.5. Initializing the block size

Before using the stack it's mandatory to initialize the block size. With the function `initBlockSize()`, the user can define the block size. By default, the block size has the value of `BLOCK_SIZE` constant. It returns nothing. Each block includes one byte to indicate the length of the data and the rest of the block can be used for data.

Example of use:

```
{
// Sets the block size to 100 Bytes
stack.initBlockSize(100);
}
```

7.6. Push

To store a block of data into the stack, the `push()` function must be used. The size of the block must be lower than `block_size`. The required are: a vector of data to store and the vector length. The vector of data must be `uint8_t` type.

It returns:

'1' on success

'0' error writing

'2' is full

'3' block size small.

Example of use:

```
{  
  // Pushes a frame into the stack  
  stack.push(frame.buffer, frame.length);  
}
```

7.7. Pop

The `pop()` function extracts a block of data previously stored into the stack. The function requires a pointer to a vector to store the extracted data.

It returns:

'0' if empty

'1' error reading

'2' if error updating the new pointer

Positive value, except 1 or 2, means the number of bytes read

Example of use:

```
{  
  // Extracts a frame from the stack  
  frame.length = stack.pop(frame.buffer);  
}
```

8. Code examples and extended information

In the WaspMote Development section you can find complete examples:

<http://www.libelium.com/development/waspmote/examples>

9. API changelog

Keep track of the software changes on this link:

www.libelium.com/development/waspmote/documentation/changelog/#Utilities