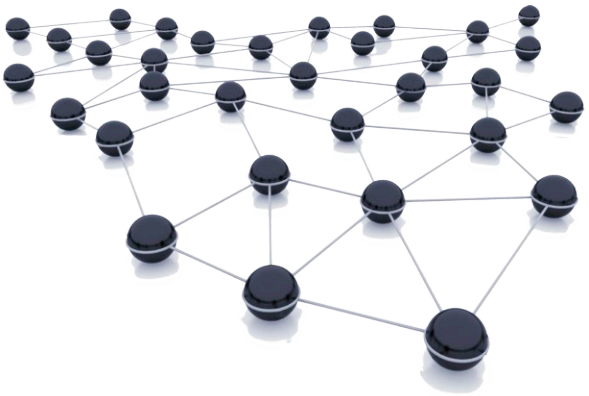
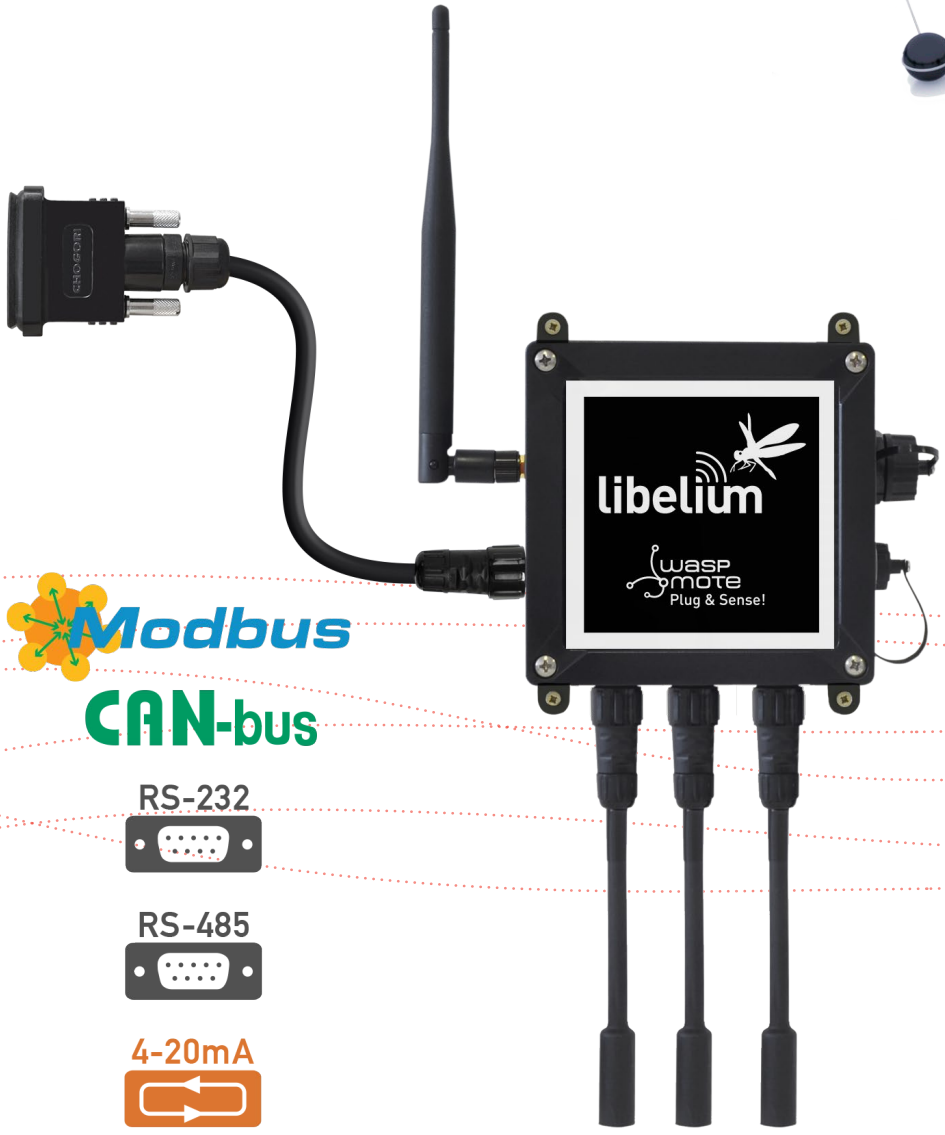
Can Bus

Communication Guide





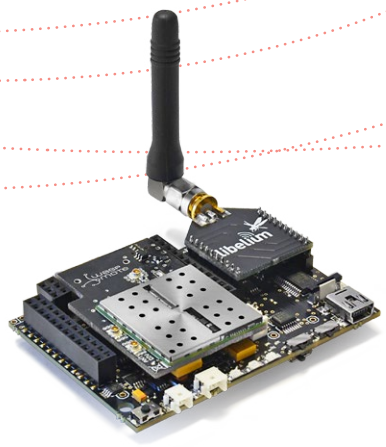
Modbus

CAN-bus

RS-232

RS-485

4-20mA



libelium

wasp mote

Document version: v7.1 - 02/2017

INDEX

1. Introduction

This guide explains the CAN Bus module features and functions. This product was designed for Waspmote v12 and continues with no changes for Waspmote v15. There are no great variations in this library for our new product line Waspmote v15, released on October 2016.

Anyway, if you are using previous versions of our products, please use 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".

1.1. The standard

This guide describes all features of the CAN Bus module which has been mainly designed to perform vehicle diagnostics in real time. This module also allows the interconnection of different devices to work with this communications protocol for applications such as industrial networks, home automation, etc.

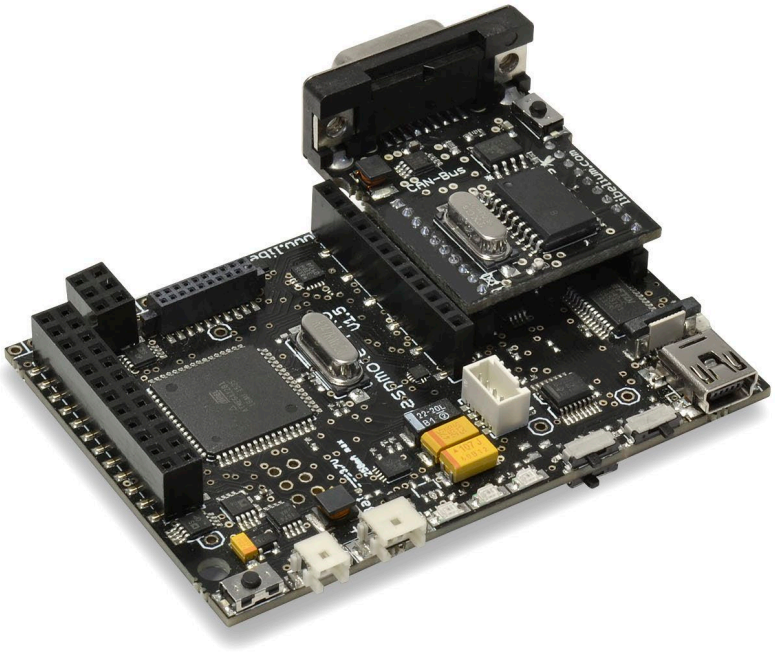


Figure : The CAN Bus module for Waspmote

The CAN Bus (Controller Area Network) is a vehicle bus standard designed to allow micro-controllers and devices to communicate with each other within a vehicle without a host computer. CAN is a multi-master broadcast serial bus standard for connecting electronic control units (ECUs). Each node is able to send and receive messages, but not simultaneously. A message consists primarily of an ID (identifier), which represents the priority of the message. A CAN message that is transmitted with highest priority will succeed and the node transmitting the lower priority message will sense this and back off and wait.

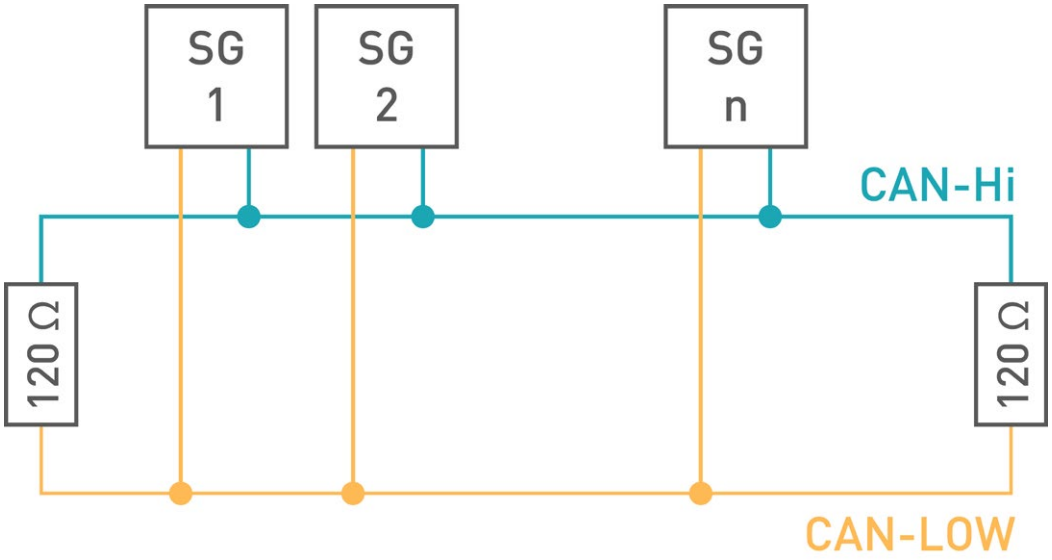


Figure : CAN Bus electrical sample topology with terminator resistors

The information is transmitted by two twisted wires that connect all system modules. It is transmitted by voltage difference between the two levels. The high voltage value represents 1 and low 0. Its combination forms an appropriate message.

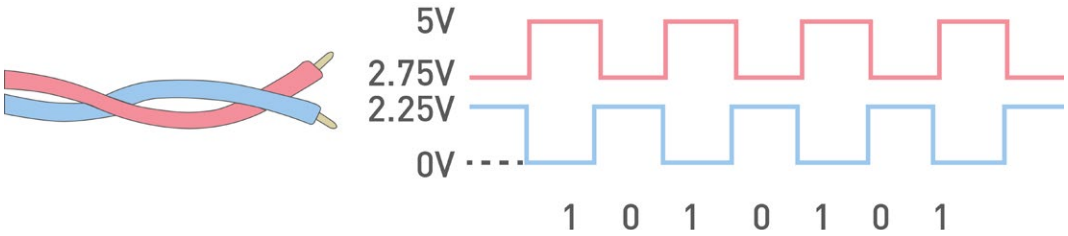


Figure : Twisted wires and tension levels

The CAN Bus standard is a high-level protocol which offers various benefits compared to other simple wired protocols. For example, all the devices can send information whenever they want (if the bus is not busy) without the need of a master. CAN Bus offers addressing, acknolowledge, retry services. Besides, the protocol sets a standard frame that can be adapted in a flexible way. Last, it is easy to reach several hundreds of metres.

This list includes some of the most common uses of the standard:

- Automotive applications
- Home automation
- Industrial networking
- Factory automation
- Marine electronics
- Medical equipment
- Military uses

1.2. Data parameters

A CAN network can be configured to work with two different message (or "frame") formats:

- The standard or base frame format (or CAN 2.0 A)
- The extended frame format (or CAN 2.0 B)

The only difference between the two formats is that the "CAN base frame" supports a length of 11 bits for the identifier, and the "CAN extended frame" supports a length of 29 bits for the identifier, made up of the 11-bit identifier ("base identifier") and an 18-bit extension ("identifier extension"). The distinction between CAN standard frame format and CAN extended frame format is made by using the IDE bit, which is transmitted as dominant in case of an 11-bit frame, and transmitted as recessive in case of a 29-bit frame.

CAN standard has four frame types:

- **Data frame:** a frame containing node data for transmission
- **Remote frame:** a frame requesting the transmission of a specific identifier
- **Error frame:** a frame transmitted by any node detecting an error
- **Overload frame:** a frame to inject a delay between data and/or remote frame

Data frame

The data frame is the only frame for actual data transmission. There are two message formats:

- Standard frame format: with 11 identifier bits
- Extended frame format: with 29 identifier bits

The CAN standard requires the implementation must accept the base frame format and may accept the extended frame format, but must tolerate the extended frame format.

Standard frame format

Field name	Length (bits)	Purpose
Start-of-frame	1	Denotes the start of frame transmission
Identifier	11	A (unique) identifier for the data which also represent the message priority
Remote transmission request (RTR)	1	Dominant (0) (see Remote Frame below)
Identifier extension bit (IDE)	1	Must be dominant (0). Optional.
Reserved bit (r0)	1	Reserved bit (it must be set to dominant (0), but accepted as either dominant or recessive)
Data length code (DLC)*	4	Number of bytes of data (0–8 bytes)
Data field	0–64 (0-8 bytes)	Data to be transmitted (length in bytes dictated by DLC field)
CRC	15	Cyclic Redundancy Check
CRC delimiter	1	Must be recessive (1)
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK delimiter	1	Must be recessive (1)
End-of-frame (EOF)	7	Must be recessive (1)

Extended frame format

Field name	Length (bits)	Purpose
Start-of-frame	1	Denotes the start of frame transmission
Identifier A	11	First part of the (unique) identifier for the data which also represents the message priority
Substitute remote request (SRR)	1	Must be recessive (1). Optional.
Identifier extension bit (IDE)	1	Must be recessive (1). Optional.
Identifier B	18	Second part of the (unique) identifier for the data which also represents the message priority
Remote transmission request (RTR)	1	Must be dominant (0)
Reserved bits (r0, r1)	2	Reserved bits (it must be set dominant (0), but accepted as either dominant or recessive)
Data length code (DLC)*	4	Number of bytes of data (0–8 bytes)
Data field	0–8 bytes	Data to be transmitted (length dictated by DLC field)
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be recessive (1)
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK delimiter	1	Must be recessive (1)
End-of-frame (EOF)	7	Must be recessive (1)

- The two identifier fields (A & B) combine to form a 29-bit identifier.
- *It is physically possible for a value between 9–15 to be transmitted in the 4-bit DLC, although the data is still limited to eight bytes. Certain controllers allow the transmission and/or reception of a DLC greater than eight, but the actual data length is always limited to eight bytes.

1.3. Frame types

Error frame

The error frame consists of two different fields:

- The first field is given by the superposition of ERROR FLAGS (6–12 dominant/recessive bits) contributed from different stations.
- The following second field is the ERROR DELIMITER (8 recessive bits).

There are two types of error flags:

- Active Error Flag: six dominant bits, transmitted by a node detecting an error on the network that is in error state "error active".
- Passive Error Flag: Six recessive bits, transmitted by a node detecting an active error frame on the network that is in error state "error passive".

Overload frame

The overload frame contains the two bit fields Overload Flag and Overload Delimiter. There are two kinds of overload conditions that can lead to the transmission of an overload flag:

- The internal conditions of a receiver, which requires a delay of the next data frame or remote frame.
- Detection of a dominant bit during intermission.

The start of an overload frame due to case 1 is only allowed to be started at the first bit time of an expected intermission, whereas overload frames due to case 2 start one bit after detecting the dominant bit. Overload Flag consists of six dominant bits. The overall form corresponds to that of the active error flag. The overload flag's form destroys the fixed form of the intermission field. As a consequence, all other stations also detect an overload condition and on their part start transmission of an overload flag. Overload Delimiter consists of eight recessive bits. The overload delimiter is of the same form as the error delimiter.

Interframe spacing

Data frames and remote frames, are separated from preceding frames, by a bit field called interframe space. Overload frames and error frames are not preceded by an interframe space and multiple overload frames are not separated by an interframe space. Interframe space contains the bit fields intermission and bus idle and, for error passive stations, which have been transmitter of the previous message, suspend transmission. Interframe space consists of at least three consecutive recessive (1) bits.

Bit stuffing

In CAN frames, a bit of opposite polarity is inserted after five consecutive bits of the same polarity. This practice is called bit stuffing, and is due to the non-return to zero (NRZ) coding adopted. The stuffed data frames are destuffed by the receiver. Since bit stuffing is used, six consecutive bits of the same type (111111 or 000000) are considered an error. Bit stuffing implies that sent data frames could be larger than one would expect by simply enumerating the bits shown in the tables above.

In order to work with the module, we not need to worry about error or overload frames, the frame parameters or as the frame is generated; because this is handled by the hardware (CAN-controller).

2. Hardware

The CAN Bus module has been tested with various devices and cars, and is compatible with the majority of commercial modules, but this does not ensure the working with all of them. Be sure that the CAN Bus module fits your technical requirements. Every car manufacturer has different CAN Bus configurations so that must be studied carefully. It is the task of the CAN Bus user to perform the integration of the CAN Bus module with commercial devices like cars or sensors.

2.1. Main features

- CAN controller MCP2515
- CAN transceiver MCP2551
- **Dimensions:** 33 x 30 x 17 mm
- **Weight:** 10 g
- Female DB9 connector
- Speed up to 1 Mbps

2.2. Electrical characteristics

- **Board power voltage:** 3.3V
- **CAN controller power voltage:** 5 V
- **CAN transceiver power voltage:** 5 V
- **Maximum admitted current (continuous):** 200 mA
- **Maximum admitted current (peak):** 300 mA
- Push button external reset

2.3. Connection diagram

The CAN Bus module uses the SPI pins for communication. The SPI port allows more speed communication and frees up the Waspmote's UART for other purposes. The Expansion Board allows to connect two boards at the same time in the Waspmote sensor platform. This means a lot of different combinations are possible using any of the radios available for Waspmote (802.15.4, ZigBee, DigiMesh, 868 MHz, 900 MHz, WiFi, GPRS, GPRS+GPS, 3G, 4G, Sigfox, LoRaWAN, Bluetooth Pro, Bluetooth Low Energy and RFID/NFC) and the CAN Bus module. The CAN Bus module must be plugged on the socket 0 and cannot be in use with other SPI modules like LoRa or RS-485.

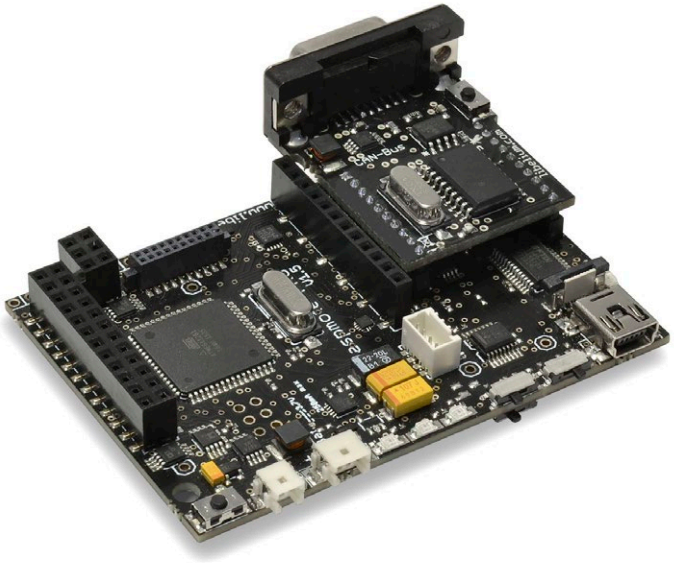


Figure : The CAN Bus module in socket 0

2.4. Consumption

The CAN Bus module uses a low power transceiver. The board is guaranteed to run at data rates of 1 Mbps. The typical consumption on the board is 7mA but this consumption can increase, due to current peaks while the module is transmitting data trough the bus. The typical peak current consumption is about 30mA.

2.5. Connector

The connector used on the module is a universal DB9 connector. This connector allows the user to interconnect some devices or connect with your own vehicle, it depends on the cable.

The CAN Bus module comes with a male-female standard DB9 cable. This cable is useful for connecting the module to other CAN Bus devices which have a DB9 male connector.

Figure : Male-female DB9 cable

If user wants to connect the CAN Bus module to a vehicle, Libelium offers the option to purchase an optional OBD-II to DB9 cable. The OBD-II standard has been mandatory for all cars and light trucks sold in the United States since 1996, and the EOBD standard has been mandatory for all petrol vehicles sold in the European Union since 2001 and all diesel vehicles since 2004. The car manufactures usually protect the access to the CAN Bus of the vehicle, so sometimes is not possible to get data directly by connecting the CAN Bus module to the OBD-II connector of the car. The user must consult how to connect the CAN Bus module to the device or car.

Signal Ground	1	5
Chassis Ground	2	4
CAN High (J-2284)	3	6
ISO 9141-2 K Line	4	7
CAN Low J-2284	5	14
J1850 Bus -	6	10
J1850 Bus +	7	2
ISO 9141-2 L Line	8	15
Battery Power	9	16

Figure : OBD-II to DB9 cable pin out

This cable allows the user to access the pins on a car's OBD-II connector. It has an OBD-II connector on one end and a DB9 male serial connector on the other.

Figure : OBD-II to DB9 cable

The OBD-II specification provides for a standardized hardware interface—the female 16-pin (2x8) J1962 connector. Unlike others connectors, which was sometimes found under the hood of the vehicle, the OBD-II connector is required to be within 2 feet (0.61m) of the steering wheel (unless an exemption is applied for by the manufacturer, in which case it is still somewhere within reach of the driver).

3. Dual radio with Expansion Board

3.1. Expansion Radio Board

The RS-232 module can use the Expansion Radio Board. The Expansion Board allows to connect two communication modules at the same time in the Waspmote sensor platform. This means a lot of different combinations are possible using any of the wireless radios available for Waspmote: 802.15.4, ZigBee, DigiMesh, 868 MHz, 900 MHz, LoRa, WiFi, GPRS, GPRS+GPS, 3G, 4G, Sigfox, LoRaWAN, Bluetooth Pro, Bluetooth Low Energy and RFID/NFC. Besides, the following Industrial Protocols modules are available: RS-485/Modbus, RS-232 Serial/Modbus and CAN Bus.

Some of the possible combinations are:

- LoRaWAN - GPRS
- 802.15.4 - Sigfox
- 868 MHz - RS-485
- RS-232 - WiFi
- DigiMesh - 4G
- RS-232 - RFID/NFC
- WiFi - 3G
- CAN Bus - Bluetooth
- etc.

Remark: *GPRS, GPRS+GPS, 3G and 4G modules do not need the Expansion Board to be connected to Waspmote. They can be plugged directly in the socket1.*

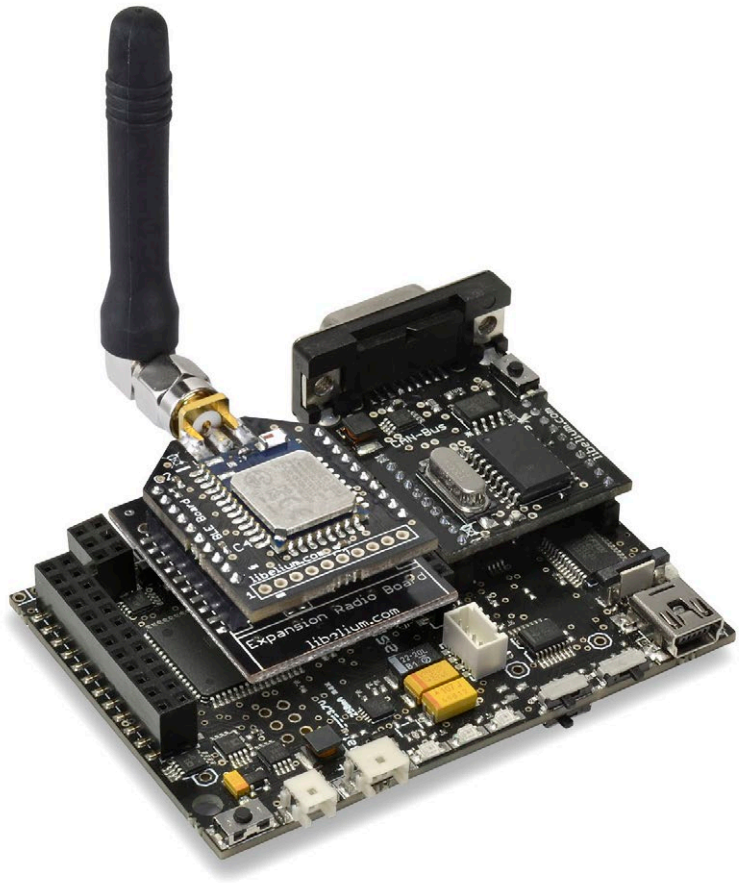


Figure : Waspmote with CAN Bus module on socket 0 and Bluetooth module on socket 1

This API provides a function in order to initialize the CAN Bus module. The CAN Bus module can be used only in the socket 0. The rest of functions are used the same way as they are used with older API versions. In order to understand them we recommend to read this guide.

Note: *CAN Bus, RS-485 and LoRa modules can only be plugged on the socket 0 of Waspmotes. That means, for example, that one Waspmote cannot have RS-485 and LoRa at the same time..*

Warnings:

- Avoid to use DIGITAL7 pin when working with the Expansion Board. This pin is used for setting the XBee into sleep mode.
- Avoid to use DIGITAL6 pin when working with the Expansion Board. This pin is used as power supply for the Expansion Board.
- Incompatibility with Sensor Boards:
  - Agriculture v30 and Agriculture PRO v30: Incompatible with Watermark and solar radiation sensors
  - Events v30: Incompatible with interruption shift register
  - Gases v30: DIGITAL6 is incompatible with CO2 (SOCKET_2) and DIGITAL7 is incompatible with NO2 (SOCKET_3)
  - Smart Water v30: DIGITAL7 incompatible with conductivity sensor
  - Smart Water Ions v30: Incompatible with ADC conversion (sensors cannot be read if the Expansion Board is in use)
  - Gases PRO v30: Incompatible with SOCKET_2 and SOCKET_3
  - Cities PRO v30: Incompatible with SOCKET_3. I2C bus can be used. No gas sensor can be used.

4. CAN Bus on Plug & Sense!

The CAN Bus protocol is available for Plug & Sense! as a secondary communication module. This is an optional feature. The CAN Bus module is placed on socket 0 by default, being accessible through an additional and dedicated socket on the antenna side of the enclosure. On the other hand, the main radio interface of the Plug & Sense! device is placed on socket 1.

Figure : Industrial Protocols available on Plug & Sense!

The user can choose between 2 probes to connect the CAN Bus protocol: A standard DB9 connector and a waterproof terminal block junction box. These options make the connections on industrial environments or outdoor applications easier.

Figure : DB9 probe connected to Plug & Sense!

Figure : Terminal box probe connected to Plug & Sense!

The CAN Bus signals are wired on the female DB9 connector and on the Terminal box according to the next table:

CAN Bus		
Terminal Box	**DB9**	
-	-	1
-	-	2
CAN_H	CAN_H	3
-	-	4
CAN_L	CAN_L	5
-	-	6
-	-	7
-	-	8
-	-	9

Figure : Wiring of CAN Bus signals on Plug & Sense!

5. Applications

This module allows the user to interface the Waspmote ecosystem with CAN Bus systems. Waspmote allows to perform three main applications:

1º- Connect any sensor to an existing CAN Bus device/network

Waspmote can be configured to work as a node in the network, inserting sensor data into the CAN Bus already present. Waspmote can obtain information from more than 70 sensors which are currently integrated in the platform by using specific sensor boards (e.g: CO, CO_2, temperature, humidity, acceleration, pH, IR, luminosity, etc). This way, the sensor information can be read from any CAN Bus device connected to the bus.

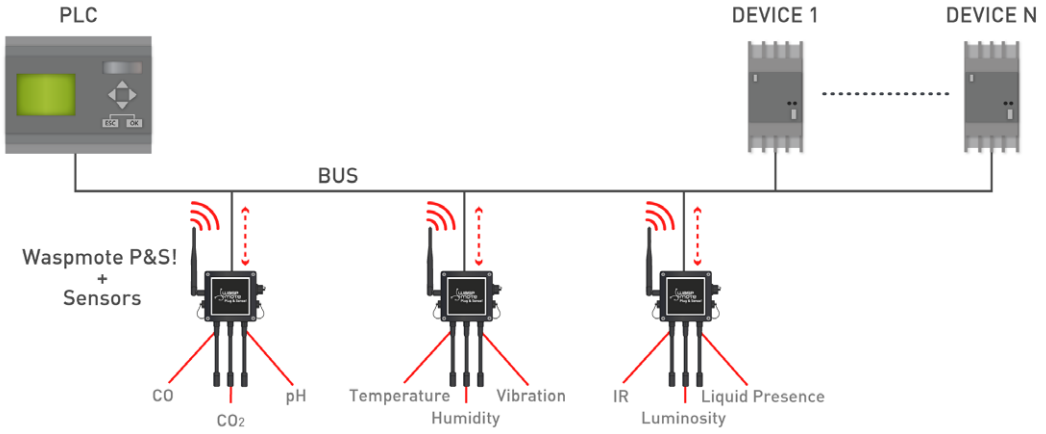


Figure : Waspmote integrated in a CAN Bus network

2º- Add wireless connectivity to CAN Bus devices

Waspmote can be configured to read the information from the bus and send it to the Libelium IoT Gateway using any of the wireless radio modules available: 802.15.4, 868 MHz, 900 MHz, WiFi, 4G, Sigfox and LoRaWAN, Bluetooth Pro, Bluetooth Low Energy and RFID/NFC. Remember that the CAN Bus module can only be plugged on the socket 0 because it is controlled via SPI bus; this makes impossible to plug another SPI-type module in the same Waspmote, like RS-485 or LoRa.

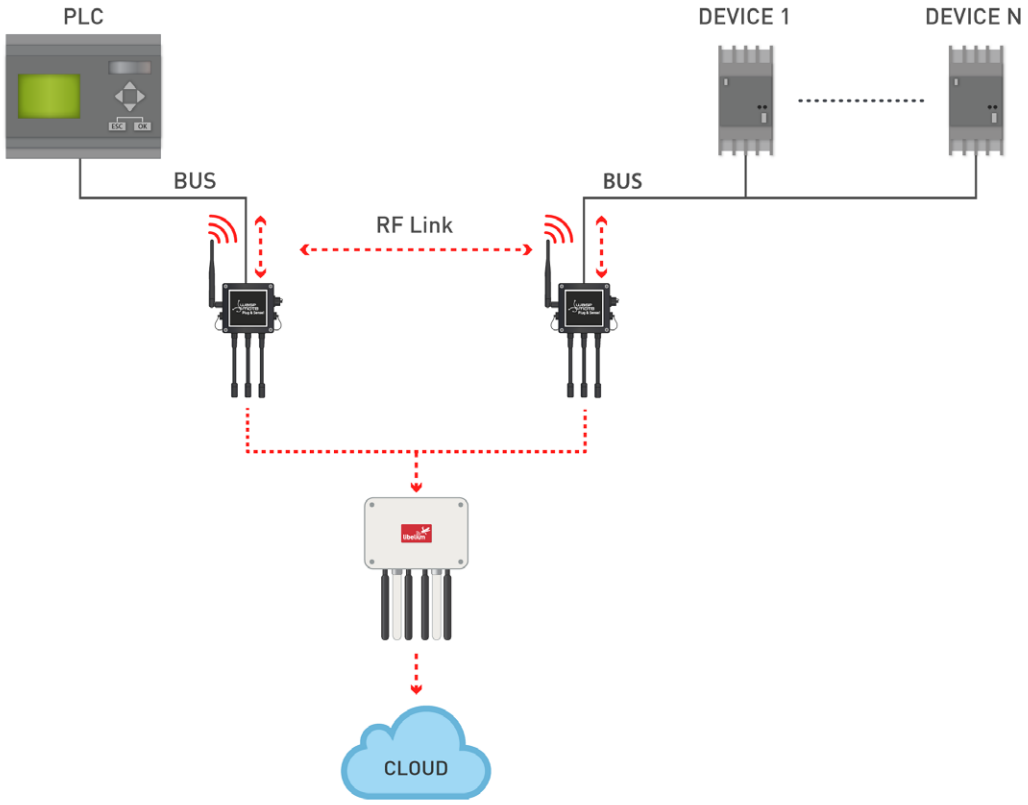


Figure : wireless connectivity

3º- Connect to the Cloud CAN Bus devices

Waspmote can be configured to read the information coming from the CAN Bus and send it wirelessly directly to the Cloud using WiFi, GPRS, GPRS+GPS, 3G and 4G radio interfaces.

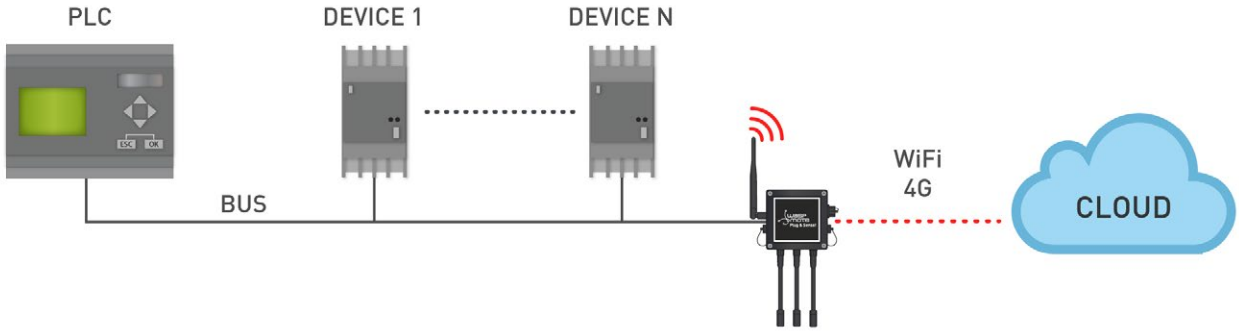


Figure : Cloud connection

6. Libelium's API

It is mandatory to include the CAN Bus library when using this module. The following line must be introduced at the beginning of the code:

```
#include <WaspCAN.h>
```

Waspmote's API CAN Bus files:

- WaspCAN.cpp
- WaspCAN.h

APIs functions

- Private functions:

The following functions are executed inside the API functions. In normal conditions, the user must NOT manage or use them.

`void writeRegister( char direction, char data)`	Writes a MCP2515 register
`char readRegister(char direction);`	Reads a MCP2515 register
`void bitModify(char direction, char mask, char data);`	Modifies a bit of the MCP2515 registers
`char readStatus(char type);`	Checks the status of the MCP2515 registers
`bool checkFreeBuffer(void);`	Checks if the buffers are empty
`void reset(void);`	Resets module

Figure : Table of private functions

- Data structure:

This is a structure created inside the CAN Bus library. The structure fits with the standard CAN Bus frame, so it is used to create the message to send, or to receive messages from other devices.

```
typedef struct{
  unsigned int id;
        struct {
              char rtr: 1;
              char length: 4;
        }header;

  uint8_t data[8];
}messageCAN;

messageCAN messageRx;
messageCAN messageTx;
```

-Public functions:

`WaspCAN(void)`	Empty constructor
`bool ON(uint16_t speed);`	Powers the CAN Bus module and opens the SPI
`uint8_t messageAvailable(void);`	Checks if there is any message
`char getMessage(messageCAN *msje);`	Takes the CAN message
`char sendMessage(messageCAN *msje);`	Sends the CAN message
`void printMessage(messageCAN *msje);`	Prints a CAN message in the serial monitor
`void setMode(uint8_t mode);`	Configures the MCP2515 work mode

Figure : Table of public functions

- CAN in a Automation functions:

The OBD-II standard sets some common PIDs for all cars, but it is not always respected. This is a list of useful functions for retreiving data from cars. Consult the specific PIDs in each car.

`unsigned int getEngineLoad();`	Calculated engine load value in %
`unsigned int getEngineCoolantTemp();`	Engine coolant temperature in ºC
`unsigned int getFuelPressure();`	Fuel pressure in KPa
`unsigned int getIntakeMAPressure();`	Intake manifold absolute pressure in KPa
`unsigned int getEngineRPM();`	Engine RPM
`unsigned int getVehicleSpeed();`	Vehicle speed in Km/h
`unsigned int getTimingAdvance();`	Timing advance (time in seconds from the car start)
`unsigned int getIntankeAirTemp();`	Intake air temperature in ºC
`unsigned int getMAFairFlowRate();`	MAF air flow rate in g/s
`unsigned int getThrottlePosition();`	Throttle position in %
`unsigned int getFuelLevel();`	Fuel level in %
`unsigned int getBarometricPressure();`	Barometric pressure in KPa
`unsigned int getEngineFuelRate();`	Engine fuel rate in litres/hour
`void CiARequest(uint8_t PID);`	General PID function. Request information through OBD.

Figure : Table of CiA functions

7. Library functions

7.1. Library constructor

To start using Waspmote CAN Bus library, an object from class 'WaspCAN' must be created. This object, called 'CAN', is created inside the CAN Bus library and it is public to all libraries. It is used through this guide to show how the CAN Bus library works. When creating this constructor, all the variables are defined with an initial value by default.

7.2. Switching the module on

This function powers the CAN Bus module and configures the SPI bus. The CAN Bus module can be used only in the socket 0. This function is necessary to configure the module, so the CAN Bus module must be plugged before. In this function must be configured the speed communication. There are four possibilities, 125 Kbs, 250 Kbps, 500 Kbps and 1 Mbs. The most frequently used is 500 Kbps. To operate with the CAN Bus module, we must work with all nodes at the same rate.

Example of use:

```
// Setting up our devices and I/Os
void setup()
{
  // Inits the USB
  USB.ON();
  delay(100);

  // Let's open the bus. Remember the input parameter:
  // 1000: 1Mbps
  // 500: 500Kbps <--- Most frequently used
  // 250: 250Kbp
  // 125: 125Kbps
  CAN.ON(1000);
}
```

You can see how to use this function in this example:
http://www.libelium.com/development/waspmote/examples/can-bus-01-basic-example

7.3. Switching the module off

Switches off the Can Bus module and stops sending data frames. This function will disconnect the power supply of the module so all data stored on the stack will be lost.

Example of use:

```
{
  // Switches off the module
  CAN.OFF();
  delay(100);
}
```

7.4. Receiving data

We can receive CAN messages, filling the predefined message (messageCAN) with the CAN controller message.

```
{
//**************************************
// 1. Receive data
//**************************************

  if (CAN.messageAvailable() == 1)
  {
        // Read the last message received
        CAN.getMessage(&CAN.messageRx);
        // Print in the serial monitor the received message
        CAN.printMessage(&CAN.messageRx);
  }
}
```

You can see how to use this function in this example:
http://www.libelium.com/development/waspmote/examples/can-bus-01-basic-example

7.5. Sending data

The CAN Bus module supports CAN to exchange data with other devices. This profile allows to create connections to another device using the same profile (CAN connection). Some functions have been developed to handle the communication between two or more devices. We can send CAN messages, sending the message that we have defined.

Example of use:

```
{
//**************************************
// 2. Send data
//**************************************

  // Insert the ID in the data structure
  CAN.messageTx.id = OWNID;
  // These fields include the data to send
  CAN.messageTx.data[0] = 0;
  CAN.messageTx.data[1] += 1;
  CAN.messageTx.data[2] = 2;
  CAN.messageTx.data[3] = 3;
  CAN.messageTx.data[4] = 4;
  CAN.messageTx.data[5] = 5;
  CAN.messageTx.data[6] = 6;
  CAN.messageTx.data[7] = 7;

  // The length of the data structure
  CAN.messageTx.header.length = 8;
  // Send data
  CAN.sendMessage(&CAN.messageTx);
}
```

You can see how to use this function in this example:
http://www.libelium.com/development/waspmote/examples/can-bus-01-basic-example

7.6. Printing data

We can print through the serial port the received message to view the data. This functions is very useful for debugging and code testing.

Example of use:

```
{
  CAN.printMessage(&CAN.messageRx);
  delay(10);
}
```

7.7. CAN in Automation

The Can Bus library defines the most important and used PIDs. The CAN Bus module can be used to request information from a vehicle.

Typically, an automotive technician will use PIDs with a proffesional scan tool connected to the vehicle's OBD-II connector. The scan tool sends PIDs to the vehicle's controller area network (CAN). A device on the bus recognizes the PID as one it is responsible for, and reports the value for that PID to the bus. The scan tool reads the response, and displays it. The user could use Waspmote to emulate this kind of scan tools.

Example of use:

```
{
  // Read the value of vehicle speed
  int vehicleSpeed = CAN.getVehicleSpeed();

  // Print received data in the serial monitor
  USB.print("Vehicle Speed: ");
  USB.print(vehicleSpeed);
  USB.println(" Km/h");
  delay(1000);
}
```

Please note that each car manufacturer implements a different CAN Bus configuration, and each car has a different PID list. This information is usually difficult to access for security reasons.

You can see how to use this function in this example:
http://www.libelium.com/development/waspmote/examples/can-bus-02-get-engine-rpm

The user can also request additional PIDs that he can define in the library. On the other hand, the user can request PIDs manually with the use of the function `CiARequest()`.

Example of use:

```
{
  int data;

  // Read the value of RPM if the engine
  CAN.CiARequest(myPID);

  if (CAN.messageRx.id == ID_RESPONSE)
  {
        // Insert the formula here
        data = uint16_t(CAN.messageRx.data[3]);
        CAN.printMessage(&CAN.messageRx);
  }

  // Print received data in the serial monitor
  USB.print("Returned data: ");
  USB.print(data);
  USB.print("\n");
  delay(1000);
}
```

You can see how to use this function in this example:
http://www.libelium.com/development/waspmote/examples/can-bus-05-general-pids

8. Certifications

Libelium offers 2 types of IoT sensor platforms, Waspmote OEM and Plug & Sense!:

- **Waspmote OEM** is intended to be used for research purposes or as part of a major product so it needs final certification on the client side. More info at: www.libelium.com/products/waspmote
- **Plug & Sense!** is the line ready to be used out-of-the-box. It includes market certifications. See below the specific list of regulations passed. More info at: www.libelium.com/products/plug-sense

Besides, Meshlium, our multiprotocol router for the IoT, is also certified with the certifications below. Get more info at:

www.libelium.com/products/meshlium

List of certifications for Plug & Sense! and Meshlium:

- CE (Europe)
- FCC (US)
- IC (Canada)
- ANATEL (Brazil)
- RCM (Australia)
- PTCRB (cellular certification for the US)
- AT&T (cellular certification for the US)

Figure : Certifications of the Plug & Sense! product line

You can find all the certification documents at:

www.libelium.com/certifications

9. Code examples and extended information

For more information about the Waspmote hardware platform go to:
http://www.libelium.com/waspmote
http://www.libelium.com/development/waspmote

In the Waspmote Development section you can find complete examples:
http://www.libelium.com/development/waspmote/examples

Example:

```
/*
 *  ------ [CAN_Bus_04] CAN Bus Dash Board --------
 *
 *  This sketch shows how to get the most important parameters from
 *  a vehicle using the standard OBD-II PIDs codes. This codes are
 *  used to request data from a vehicle, used as a diagnostic tool.
 *
 *  Copyright (C) 2014 Libelium Comunicaciones Distribuidas S.L.
 *  http://www.libelium.com
 *
 *  This program is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

// Include always these libraries before using the CAN Bus functions
#include <WaspCAN.h>

void setup()
{
  // Turn on the USB
  USB.ON();
  delay(100);

  // Print initial message
  USB.println("Initializing CAN Bus...");

  // Configuring the Bus at 500 Kbit/s
  CAN.ON(500);

  USB.println("CAN Bus initialized at 500 KBits/s");
  USB.println();
}
```

```
void loop()
{
  // Read the value of the Vehicle Speed
  int vehicleSpeed = CAN.getVehicleSpeed();

  // Read the value of RPM of the engine
  int engineRPM = CAN.getEngineRPM();

  // Read the engine fuel rate
  int engineFuelRate = CAN.getEngineFuelRate();

  // Get the fuel level
  int fuelLevel = CAN.getFuelLevel();

  // Get the throttle position
  int throttlePosition = CAN.getThrottlePosition();

  //Get the fuel pressure value
  int fuelPressure = CAN.getFuelPressure();

  USB.println(F("<=============================================>"));
  USB.print(F("\tVehicle Speed => "));
  USB.print(vehicleSpeed);
  USB.println(" Km/h");

  USB.print(F("\tEngine RPM => "));
  USB.print(engineRPM);
  USB.println(" RPM");

  USB.print(F("\tEngine Fuel Rate => "));
  USB.print(engineFuelRate);
  USB.println(" L/h");

  USB.print(F("\tFuel Level => "));
  USB.print(fuelLevel);
  USB.println(" %");

  USB.print(F("\tThrottle Position => "));
  USB.print(throttlePosition);
  USB.println(" %");

  USB.print(F("\tFuel Pressure => "));
  USB.print(fuelPressure);
  USB.println(" KPa");

  USB.println(F("<=============================================>"));
  USB.println();
  delay(1000);
}
```

10. Documentation changelog

From v7.0 to v7.1:

- Added references to the integration of Industrial Protocols for Plug & Sense!

11. API changelog

Keep track of the software changes on this link:

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