

## CAN library

This library allows you to communicate with MCP2515 CAN controller through SPI making CAN communications possible through Arduino. For more information and details about MCP2515 see Datasheet at <http://ww1.microchip.com/downloads/en/DeviceDoc/21801G.pdf>

Note: Arduino SPI library is required (Installed by default when installing Arduino IDE).

## CAN Functions

All of these functions will need a class object creation. For these examples we will use CAN2 but it can be anything besides just plain CAN.

MCP CAN2 (SPI CS pin);

### Begin()

Begin function will initialize MCP2515 CAN controller and set it to 1 of 5 modes. It will also set the CAN bit rate.

### Function:

```
CAN2 .begin(mode, bitrate);
```

### Parameters:

- **mode**
  - NORMAL: Mode allows active monitoring of CAN activity, send and receive messages and make full use of all flags, filters, and errors.
  - LISTEN: Mode allows monitoring of CAN activity by receiving messages (valid or invalid)
  - SLEEP: Mode puts MCP2515 to sleep. SPI interface is still active to receive commands
  - LOOPBACK: Mode allows internal transmission of messages between TX and RX buffers without actually sending messages on the CAN bus
  - CONFIG: Default mode when MCP2515 is powered up, reset or via command. The following registers are only modifiable in this mode:
    - CNF1, CNF2, CNF3
    - TXRTSCTRL
    - Filter registers
    - Mask registers

- **bitrate**
  - 10 kBit/s
  - 20 kBit/s
  - 50 kBit/s
  - 100 kBit/s
  - 125 kBit/s
  - 250 kBit/s (typical J1939 bit rate)
  - 500 kBit/s (typical CAN bit rate or J1939 “High Speed”)
  - 1000 MBit/s

## **Send()**

Send function will load one of the 3 TX buffers with a valid message and will send when CAN bus is available.

## **Function:**

```
CAN2 .send(ID, frame type, length, data);
```

## **Parameters:**

- **ID:** This is the CAN message identifier. ID can be 11 bit (standard) or 29 bit (extended) frame. ID can be given an unsigned long variable for either frame. Example IDs:
  - **CAN:** 0x02F
  - **J1939:** 0x18FEF121
  - **CANopen:** 0x608
- **Frame type:** Parameter determines if ID is extended or standard.
  - **stdID** This sets message to be a standard 11 bit identifier
  - **extID** This sets message to be an extended 29 bit identifier
- **Length:** This provides the number of data byte message will contain. It can range from 1 to 8. (0 if no data).
- **Data:** Data is contained within a byte array declared before this function. Data can range from 1-8 byte length. Example:
  - **data[]** :{0x10, 0x15, 0xFF, 0x1C.....}

## Read()

Function will return a valid message received within one of the 2 RX buffers. Once a message is read the buffer will automatically clear and set the correct flags. Library contains 2 different functions to read a message depending on the user application.

### Function:

```
CAN2.read(&ID, &length, &data);
```

### Parameter:

- **ID:** This will return the CAN message identifier from message received. ID could be 11 bit (standard) or 29 bit (extended) frame.
- **Length:** This will return the number of data bytes expected to be in the message.
- **Data:** This will return the data within the data fields

### Function:

```
CAN2.read(*message);
```

### Parameter:

Message structure will need to be defined before using this function. Messages structures are as follows:

#### **CAN message;**

- This will return ID, rtr, data length and data

#### **J1939 message;**

- This will return ID, Priority, PGN, source address, destination address, data length, data

#### **CANopen message;**

- This will return COB\_ID, Function code, Node, rtr, data length and data.

### **loadMsg()**

Load message function can be used by user when a TX message needs to be loaded to a specific buffer but does not need to be sent yet.

#### **Function:**

```
CAN2.loadMsg(buffer, ID, frame type, length, data);
```

#### **Parameter:**

Parameters are the same as send() but with the addition of buffer

- **Buffer:**
  - **buffer1:** Loads message to buffer 1
  - **buffer2:** Loads message to buffer 2
  - **buffer3:** Loads message to buffer 3
  - **allBuffers:** Loads message to all buffers

### **sendTx()**

Send TX message function can be used by user when a TX message has been loaded to a specific buffer and needs to be sent. This function is also called requested to send (RTS) by MCP2515

#### **Function:**

```
CAN2.sendTx(buffer);
```

#### **Parameter:**

- **Buffer:**
  - **buffer1:** Sends message in buffer 1
  - **buffer2:** Sends message in buffer 2
  - **buffer3:** Sends message in buffer 3
  - **allBuffers:** Sends messages in all buffer 3

### **msgAvailable()**

Message available is a boolean function that will return true or false (1 or 0) if a valid message has been received in one of the two buffers. This function differs from Arduino available() as it does not return the number of bytes available to read.

#### **Function:**

```
CAN2.msgAvailable();
```

#### **Returns:**

**FALSE, NO or 0:** If RX buffer is empty

**TRUE, YES or 1:** If message is available in one of the two RX buffers.

### **clearRxBuffers()**

Clear receive buffers loads buffers with zeros. At power up, MCP2515 buffers are not truly empty. There is random data in the registers. Loading zeros prevents incorrect data to be read. Only needed at start up if user wants to be sure data is correct.

#### **Function:**

```
CAN2.clearRxBuffers();
```

#### **Returns:**

None

### **clearTxBuffers()**

Clear transmit buffers loads buffers with zeros. At power up, MCP2515 buffers are not truly empty. There is random data in the registers. Loading zeros prevents incorrect data to be sent. Only needed at start up if user wants to be sure data is correct.

#### **Function:**

```
CAN2.clearTxBuffers();
```

#### **Returns:**

None

#### **Note:**

If a sendTx() function is used on a buffer with zeros, it will send a message with zeros.

### clearFilters()

This will change receive buffers to accept all messages (standard or extended).

#### Function:

```
CAN2.clearFilters();
```

#### Returns:

None

### setInterrupts()

This will enable and set CAN interrupts.

#### Function:

```
CAN2.setInterrupts(mask, value);
```

#### Parameters:

**Mask:** This determines which bits in the register will be changed. A bit of 1 in the mask byte will allow a bit in the register to change, while a bit 0 will not.

**Value:** This determines what value the allowed modifiable bits will be set to.

#### Example:

	MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE	Address Value	
Mask	0	0	0	0	0	0	1	1	0x03	Mask to enable RX Interrupts
Value	x	x	x	x	x	x	1	1	0x03	Value to enable RX Interrupts
Previous Register	0	0	0	0	0	0	0	0	0x00	
Register	0	0	0	0	0	0	1	1	0x03	RX Interrupts enabled
Mask	0	0	0	1	1	0	0	0	0x18	Mask to enable TX Interrupts without changing previously set interrupts
Value	x	x	x	1	1	x	x	x	0x18	Value to enable RX Interrupts
Previous Register	0	0	0	0	0	0	1	1	0x00	
New register	0	0	0	1	1	0	1	1	0x1B	TX Interrupts enabled with previously enable RX interrupts

X = don't care

## Bit values

Bit 7: MERRE: Message Error Interrupt Enable bit

Bit 6: WAKIE: Wake-up Interrupt Enable bit

Bit 5: ERRIE: Error Interrupt Enable bit (multiple sources in EFLG register)

Bit 4: TX2IE: Transmit Buffer 2 Empty Interrupt Enable bit

Bit 3: TX1IE: Transmit Buffer 1 Empty Interrupt Enable bit

Bit 2: TX0IE: Transmit Buffer 0 Empty Interrupt Enable bit

Bit 1: RX1IE: Receive Buffer 1 Full Interrupt Enable bit

Bit 0: RX0IE: Receive Buffer 0 Full Interrupt Enable bit

## setMask() setFilter()

These two functions work together in order to set message acceptance filters. A mask is used to determine which bits will be checked against the filter. A zero will be accepted, a 1 will be rejected. The filter checks SIDH, SIDL, EID8 and EID0 buffer.

Note: A filter on one buffer will not stop messages to come into the second rx buffer

There are 2 masks and 6 filters and they are used as follow:

## Function:

```
CAN2.setMask(mask, data1, data2, data3, data4);
```

```
CAN2.setMask(filter, data1, data2, data3, data4);
```

## Parameters:

Mask & Filter

Buffer1 = mask0 used with filter0 and filter1.

Buffer 2 = mask1 used with filter2, filter3, filter4 and filter5.

## Example

### Extended ID

```
CAN2.setMask(mask0, 0x00, 0x00, 0xFF, 0xFF); // Mask set to check the last bytes of the message ID  
CAN2.setFilter(filter0, 0x00, 0x00, 0x10, 0xA2); // Only message with last four ID 10A2 will be accepted  
in buffer1
```