ACAN2515 library for Arduino
Version 1.0.4

Pierre Molinaro

November 23, 2018

Contents

1 Versions

Version	Date	Comment
1.0.0	October 12, 2018	Initial release
1.0.1	October 23, 2018	Workaround external interrupt masking for Teensy 3.5 / 3.6 Use of a lambda function for interrupt service routine
1.0.3	November 3, 2018	Correct setting of `rtr` and `ext` properties on message receive (thanks to Arjan-Woltjer for having fixed this bug, `https://github.com/pierremolinaro/acan2515/pull/1`)
1.0.4	November 23, 2018	BugFix: transmit buffer #2 size setting Transmit and send buffers properties are now `uint16_t` (instead of `uint32_t`), for saving memory `ACAN2515::begin` now returns an `uint16_t` (instead of `uint32_t`) New `ACAN2515Settings` constructor with explicit bit rate settings (see section 12.2 page 28 and `LoopBackDemoBitRateSettings` demo sketch)

2 Features

The `ACAN2515` library is a MCP2515 CAN ("Controller Area Network") Controller driver for any board running Arduino. It has been designed to make it easy to start and to be easily configurable:

- default configuration sends and receives any frame – no default filter to provide;
- efficient built-in CAN bit settings computation from user bit rate;
- user can fully define its own CAN bit setting values;
- all reception filter registers are easily defined (2 mask registers, 6 acceptance registers);
- reception filters accept call back functions;
- driver transmit buffer sizes are customisable;
- driver receive buffer size is customisable;
- overflow of the driver receive buffer is detectable;
- *loop back*, *self reception*, *listing only* MCP2515 controller modes are selectable.

3 Data flow

The figure 1 illustrates message flow for sending and receiving CAN messages.

Sending messages. A message is defined by an instance of `CANMessage` class. For sending a message, user code calls the `tryToSend` method – see section 7 page 12, and the `idx` property of the sent message

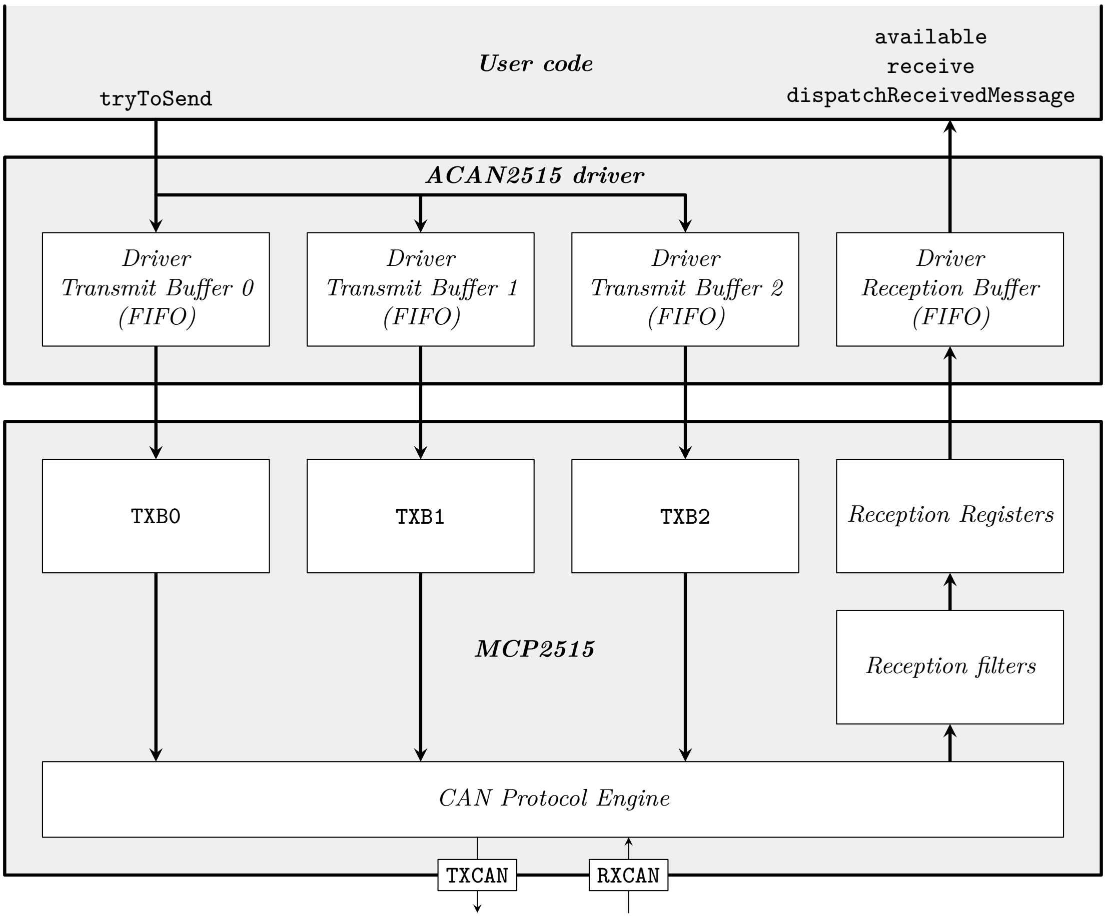

Figure 1 – Message flow in `ACAN2515` driver and `MCP2515` CAN Controller

specifies a transmit buffer. The ACAN2515 driver defines 3 transmit buffers, each of them corresponding to the one of the 3 `MCP2515` transmit buffers (`TXB0`, `TXB1`, `TXB2`). These buffers can contain at most one message. The message is transfered in a driver transmit before to be moved by the interrupt service routine into the corresponding MCP2515 transmit buffer. The size of the *Driver Transmit Buffer 0* is 16 by default, the size of the *Driver Transmit Buffer 1* and *Driver Transmit Buffer 1* are zero by default – see section 7.2 page 13 for changing the default values.

Receiving messages. The MCP2515 *CAN Protocol Engine* transmits all correct frames to the *reception filters*. By default, they are configured as pass-all, see section 9 page 16 for configuring them. Messages that pass the filters are stored in the *Reception Registers* (`RXB0` and `RXB1`). The interrupt service routine transfers the messages from these registers to the *Driver Receive Buffer*. The size of the *Driver Receive Buffer* is 32 by default – see section 8.1 page 15 for changing the default value. Three user methods are available:

- the `available` method returns `false` if the *Driver Receive Buffer* is empty, and `true` otherwise;
- the `receive` method retrieves messages from the *Driver Receive Buffer* – see section 8 page 14;
- the `dispatchReceivedMessage` method if you have defined the reception filters that name a call-back function – see section 10 page 21.

Sequentiality. The `ACAN2515` driver and the configuration of the `MCP2515` controller can ensure sequentiality of data messages[1], under some conditions. The driver ensures the sequentiality of the emissions, provided that you use only one transmit buffer: if an user program calls `tryToSend` first for a message M_1 specifying the B_i buffer and then for a message M_2 specifying the same buffer, the driver ensures that M_1 will be sent on the CAN bus before M_2. However, if M_2 specifies an other buffer, there is no guarantee that M_1 will appear on the bus before M_2. In reception, the driver ensures sequentiality based on the reception filters: if a received message M_1 passes a given filter, and then a received message M_2 passes the same filter, then the messages are retrieved in this order by the `receive` or the `dispatchReceivedMessage` methods.

4 A simple example: LoopBackDemo

The following code is a sample code for introducing the `ACAN2515` library, extracted from the `LoopBackDemo` sample code included in the library distribution. It runs natively on any Arduino compatible board, and is easily adaptable to any microcontroller supporting `SPI`. It demonstrates how to configure the driver, to send a CAN message, and to receive a CAN message.

Note: this code runs without any CAN transceiver (the `TXCAN` and `RXCAN` pins of the `MCP2515` are left open), the `MCP2515` is configured with the *loop back* setting on.

```
#include <ACAN2515.h>
```

This line includes the `ACAN2515` library.

```
static const byte MCP2515_SCK = 27 ; // SCK input of MCP2515
static const byte MCP2515_SI  = 28 ; // SI input of MCP2515
static const byte MCP2515_SO  = 39 ; // SO output of MCP2515
```

Define the `SPI` alternate pins. This is actually required if you uses `SPI` alternate pins.

```
static const byte MCP2515_CS  = 20 ; // CS input of MCP2515
static const byte MCP2515_INT = 37 ; // INT output of MCP2515
```

Define the pins connected to $\overline{\texttt{CS}}$ and $\overline{\texttt{INT}}$ pins.

```
ACAN2515 can (MCP2515_CS, SPI, MCP2515_INT) ;
```

Instanciation of the `ACAN2515` library, declaration and initialization of the `can` object that implements the driver. The constructor names: the number of the pin connected to the $\overline{\texttt{CS}}$ pin, the `SPI` object (you can use `SPI1`, `SPI2`, …), the number of the pin connected to the $\overline{\texttt{INT}}$ pin.

[1] Sequentiality means that if an user program calls `tryToSend` first for a message M_1 and then for a message M_2, the message M_1 will be allways retrieved by `receive` or `dispatchReceivedMessage` before the message M_2.

```
static const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
```

Specifies the frequency of the MCP2515 quartz.

```
void setup () {
//--- Switch on builtin led
  pinMode (LED_BUILTIN, OUTPUT) ;
  digitalWrite (LED_BUILTIN, HIGH) ;
//--- Start serial
  Serial.begin (38400) ;
//--- Wait for serial (blink led at 10 Hz during waiting)
  while (!Serial) {
    delay (50) ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
  }
```

Builtin led is used for signaling. It blinks led at 10 Hz during until serial monitor is ready.

```
  SPI.begin () ;
```

You should call `SPI.begin`. Many platforms define alternate pins for SPI. On Teensy 3.x (section 6.1 page 9), selecting alternate pins should be done before calling `SPI.begin`, on Adafruit Feather M0 (section 6.2 page 11), this should be done after. Calling `SPI.begin` explicitly allows you to fully handle alternate pins.

```
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 125 * 1000) ;
```

Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACAN2515Settings` class. The constructor has two parameters: the `MCP2515` quartz frequency, and the desired CAN bit rate (here, 125 kb/s). It returns a `settings` object fully initialized with CAN bit settings for the desired bit rate, and default values for other configuration properties.

```
  settings.mRequestedMode = ACAN2515RequestedMode::LoopBackMode ;
```

This is the second step. You can override the values of the properties of `settings` object. Here, the `mRequestedMode` property is set to `LoopBackMode` – its value is `NormalMode` by default. Setting this property enables *loop back*, that is you can run this demo sketch even it you have no connection to a physical CAN network. The section 12.8 page 33 lists all properties you can override.

```
  const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
```

This is the third step, configuration of the `can` driver with `settings` values. The driver is configured for being able to send any (standard / extended, data / remote) frame, and to receive all (standard / extended, data / remote) frames. If you want to define reception filters, see section 9 page 16. The second argument is the *interrupt service routine*, and is defined by a C++ lambda expression[2]. See section 11.2 page 22 for using a function instead.

```
  if (errorCode != 0) {
    Serial.print ("Configuration error 0x") ;
    Serial.println (errorCode, HEX) ;
  }
}
```

Last step: the configuration of the `can` driver returns an error code, stored in the `errorCode` constant. It has the value 0 if all is ok – see section 11.3 page 23.

```
static unsigned gBlinkLedDate = 0 ;
static unsigned gReceivedFrameCount = 0 ;
static unsigned gSentFrameCount = 0 ;
```

The `gSendDate` global variable is used for sending a CAN message every 2 s. The `gSentCount` global variable counts the number of sent messages. The `gReceivedCount` global variable counts the number of received messages.

```
void loop() {
  CANMessage frame ;
```

The `message` object is fully initialized by the default constructor, it represents a standard data frame, with an identifier equal to 0, and without any data – see section 5 page 8.

```
  if (gBlinkLedDate < millis ()) {
    gBlinkLedDate += 2000 ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
    const bool ok = can.tryToSend (frame) ;
    if (ok) {
      gSentFrameCount += 1 ;
      Serial.print ("Sent: ") ;
      Serial.println (gSentFrameCount) ;
    }else{
      Serial.println ("Send failure") ;
    }
  }
```

We try to send the data message. Actually, we try to transfer it into the *Driver transmit buffer*. The transfer succeeds if the buffer is not full. The `tryToSend` method returns `false` if the buffer is full, and `true` otherwise. Note the returned value only tells if the transfer into the *Driver transmit buffer* is successful or not: we have no way to know if the frame is actually sent on the the CAN network.

[2]https://en.cppreference.com/w/cpp/language/lambda

Then, we act the successfull transfer by setting `gSendDate` to the next send date and incrementing the `gSentCount` variable. Note if the transfer did fail, the send date is not changed, so the `tryToSend` method will be called on the execution of the `loop` function.

```
  if (can.available ()) {
    can.receive (frame) ;
    gReceivedFrameCount ++ ;
    Serial.print ("Received:␣") ;
    Serial.println (gReceivedFrameCount) ;
  }
}
```

As the `MCP2515` controller is configured in *loop back* mode, all sent messages are received. The `receive` method returns `false` if no message is available from the *driver reception buffer*. It returns `true` if a message has been successfully removed from the *driver reception buffer*. This message is assigned to the `message` object. If a message has been received, the `gReceivedCount` is incremented ans displayed.

5 The `CANMessage` class

Note. The `CANMessage` class is declared in the `CANMessage.h` header file. The class declaration is protected by an include guard that causes the macro `GENERIC_CAN_MESSAGE_DEFINED` to be defined. The ACAN[3] (version 1.0.3 and above) driver, the ACAN2517[4] driver contain an identical `CANMessage.h` file header, enabling using ACAN driver, ACAN2515 driver and ACAN2517 driver in a same sketch.

A *CAN message* is an object that contains all CAN frame user informations. All properties are initialized by default, and represent a standard data frame, with an identifier equal to 0, and without any data.

```
class CANMessage {
  public : uint32_t id = 0 ;  // Frame identifier
  public : bool ext = false ; // false -> standard frame, true -> extended frame
  public : bool rtr = false ; // false -> data frame, true -> remote frame
  public : uint8_t idx = 0 ;  // Used by the driver
  public : uint8_t len = 0 ;  // Length of data (0 ... 8)
  public : union {
    uint64_t data64     ; // Caution: subject to endianness
    uint32_t data32 [2] ; // Caution: subject to endianness
    uint16_t data16 [4] ; // Caution: subject to endianness
    uint8_t  data   [8] = {0, 0, 0, 0, 0, 0, 0, 0} ;
  } ;
} ;
```

[3]The ACAN driver is a CAN driver for FlexCAN modules integrated in the Teensy 3.x microcontrollers, https://github.com/pierremolinaro/acan.

[4]The ACAN2517 driver is a CAN driver for the `MCP2517` CAN controller, https://github.com/pierremolinaro/acan2517.

Note the message datas are defined by an **`union`**. So message datas can be seen as height bytes, four 16-bit unsigned integers, two 32-bit, or one 64-bit. Be aware that multi-byte integers are subject to endianness (Cortex M4 processors of Teensy 3.x are little-endian).

The `idx` property is not used in CAN frames, but:

- for a received message, it contains the acceptance filter index (see section 10 page 21);
- on sending messages, it is used for selecting the transmit buffer (see section 7.1 page 12).

6 Connecting a MCP2515 to your microcontroller

Connecting a `MCP2515` requires 5 pins (figure 2):

- hardware SPI requires you use dedicaced pins of your microcontroller. You can use alternate pins (see below), and if your microcontroller supports several hardware SPIs, you can select any of them;
- connecting the $\overline{\texttt{CS}}$ signal requires one digital pin, that the driver configures as an `OUTPUT` ;
- connecting the $\overline{\texttt{INT}}$ signal requires one other digital pin, that the driver configures as an external interrupt input; so this pin should have interrupt capability (checked by the `begin` method of the driver object).

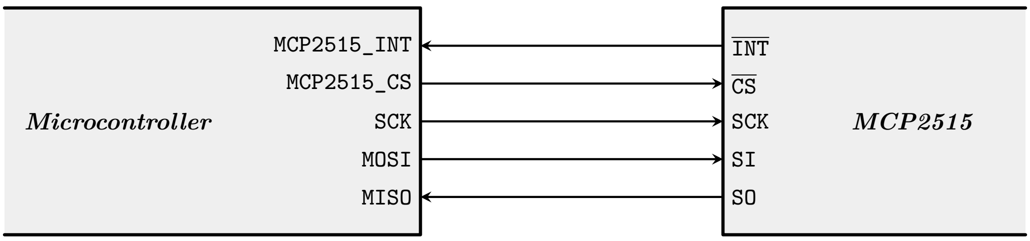

Figure 2 – `MCP2515` connection to a microcontroller

The `begin` function of `ACAN2515` library configures the selected SPI with a frequency of 10 Mbit/s (the maximum frequency supported by the `MCP2515`). More precisely, the SPI library of your microcontroller may adopt a frequency lower than 10 Mbit/s; for example, the maximum frequency of the Arduino Uno SPI is 8 Mbit/s.

6.1 Using alternate pins on Teensy 3.x

Demo sketch: `LoopBackDemoTeensy3x`.

On Teensy 3.x, "*the main SPI pins are enabled by default. SPI pins can be moved to their alternate position with `SPI.setMOSI(pin)`, `SPI.setMISO(pin)`, and `SPI.setSCK(pin)`. You can move all of them, or just the ones that conflict, as you prefer.*"[5]

For example, the `LoopBackDemoTeensy3x` sketch uses `SPI0` on a Teensy 3.5 with these alternate pins[6]:

[5] See `https://www.pjrc.com/teensy/td_libs_SPI.html`
[6] See `https://www.pjrc.com/teensy/pinout.html`

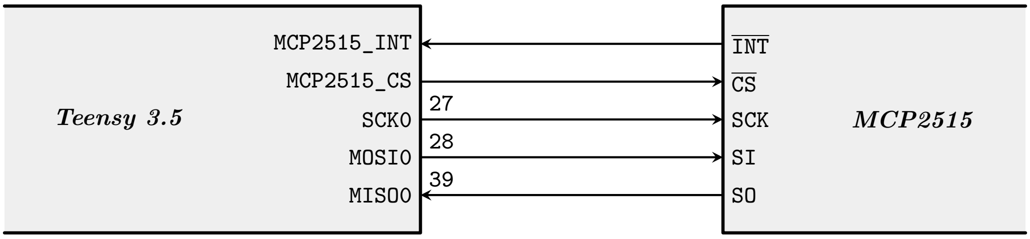

Figure 3 – Using SPI alternate pins on a Teensy 3.5

You call the `SPI.setMOSI`, `SPI.setMISO`, and `SPI.setSCK` functions **before** calling the `begin` function of your `ACAN2515` instance (generally done in the `setup` function):

```
ACAN2515 can (MCP2515_CS, SPI, MCP2515_INT) ;
...
static const byte MCP2515_SCK = 27 ; // SCK input of MCP2515
static const byte MCP2515_SI  = 28 ; // SI input of MCP2515
static const byte MCP2515_SO  = 39 ; // SO output of MCP2515
...
void setup () {
  ...
  SPI.setMOSI (MCP2515_SI) ;
  SPI.setMISO (MCP2515_SO) ;
  SPI.setSCK (MCP2515_SCK) ;
  SPI.begin () ;
  ...
  const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
  ...
```

Note you can use the `SPI.pinIsMOSI`, `SPI.pinIsMISO`, and `SPI.pinIsSCK` functions to check if the alternate pins you select are valid:

```
void setup () {
  ...
  Serial.print ("Using pin #") ;
  Serial.print (MCP2515_SI) ;
  Serial.print (" for MOSI: ") ;
  Serial.println (SPI.pinIsMOSI (MCP2515_SI) ? "yes" : "NO!!!") ;
  Serial.print ("Using pin #") ;
  Serial.print (MCP2515_SO) ;
  Serial.print (" for MISO: ") ;
  Serial.println (SPI.pinIsMISO (MCP2515_SO) ? "yes" : "NO!!!") ;
  Serial.print ("Using pin #") ;
  Serial.print (MCP2515_SCK) ;
  Serial.print (" for SCK: ") ;
  Serial.println (SPI.pinIsSCK (MCP2515_SCK) ? "yes" : "NO!!!") ;
  SPI.setMOSI (MCP2515_SI) ;
  SPI.setMISO (MCP2515_SO) ;
  SPI.setSCK (MCP2515_SCK) ;
  SPI.begin () ;
```

```
...
const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
...
```

6.2 Using alternate pins on an Adafruit Feather M0

Demo sketch: `LoopBackDemoAdafruitFeatherM0`.

See https://learn.adafruit.com/using-atsamd21-sercom-to-add-more-spi-i2c-serial-ports/overview document that explains in details how configure and set an alternate SPI on Adafruit Feather M0.

For example, the `LoopBackDemoAdafruitFeatherM0` sketch uses `SERCOM1` on an Adafruit Feather M0 as illustrated in figure 4.

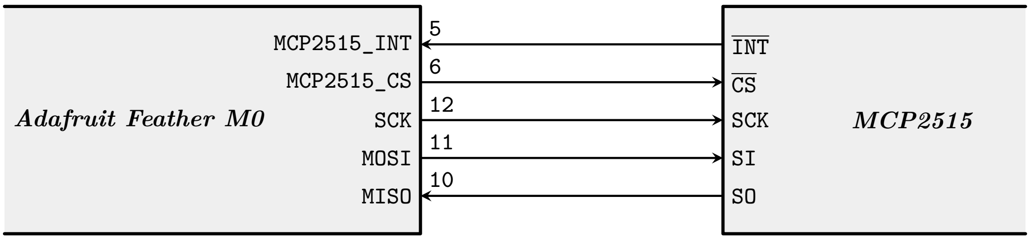

Figure 4 – Using SPI alternate pins on an Adafruit Feather M0

The configuration code is the following. Note you should call the `pinPeripheral` function **after** calling the `mySPI.begin` function.

```
#include <wiring_private.h>
...
static const byte MCP2515_SCK = 12 ; // SCK pin, SCK input of MCP2515
static const byte MCP2515_SI  = 11 ; // MOSI pin, SI input of MCP2515
static const byte MCP2515_SO  = 10 ; // MISO pin, SO output of MCP2515
...
SPIClass mySPI (&sercom1,
                MCP2515_SO, MCP2515_SI, MCP2515_SCK,
                SPI_PAD_0_SCK_3, SERCOM_RX_PAD_2);
...
static const byte MCP2515_CS  =  6 ; // CS input of MCP2515
static const byte MCP2515_INT =  5 ; // INT output of MCP2515
...
ACAN2515 can (MCP2515_CS, mySPI, MCP2515_INT) ;
...
void setup () {
  ...
  mySPI.begin () ;
  pinPeripheral (MCP2515_SI, PIO_SERCOM);
  pinPeripheral (MCP2515_SCK, PIO_SERCOM);
  pinPeripheral (MCP2515_SO, PIO_SERCOM);
```

```
...
const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
...
```

7 Sending frames

The ACAN2515 driver define three transmit buffers, each of them corresponding to a MCP2515 hardware buffer.

7.1 The tryToSend method

```
...
CANMessage message ;
// Setup message
const bool ok = can.tryToSend (message) ;
...
```

You call the tryToSend method for sending a message in the CAN network. Note this function returns before the message is actually sent; this function only appends the message to a transmit buffer.

The idx field of the message specifies the transmit buffer (0 → transmit buffer 0, 1 → transmit buffer 1, 2 → transmit buffer 2, any other value → transmit buffer 0). The default value of the idx field is zero: the message is sent throught TXB0.

The method tryToSend returns:

- true if the message has been successfully transmitted to driver transmit buffer; note that does not mean that the CAN frame has been actually sent;
- false if the message has not been successfully transmitted to driver transmit buffer, it was full.

So it is wise to systematically test the returned value.

A way is to use a global variable to note if the message has been successfully transmitted to driver transmit buffer. For example, for sending a message every 2 seconds:

```
static unsigned gSendDate = 0 ;

void loop () {
  if (gSendDate < millis ()) {
    CANMessage message ;
    // Initialize message properties
    const bool ok = can.tryToSend (message) ;
    if (ok) {
      gSendDate += 2000 ;
    }
```

```
  }
}
```

An other hint to use a global boolean variable as a flag that remains `true` while the message has not been sent.

```
static bool gSendMessage = false ;

void loop () {
  ...
  if (frame_should_be_sent) {
    gSendMessage = true ;
  }
  ...
  if (gSendMessage) {
    CANMessage message ;
    // Initialize message properties
    const bool ok = can.tryToSend (message) ;
    if (ok) {
      gSendMessage = false ;
    }
  }
  ...
}
```

7.2 Driver transmit buffer sizes

By default:

- driver transmit buffer 0 size is 16;
- driver transmit buffer 1 and 2 sizes are 0.

You can change the default values by setting the `mTransmitBuffer0Size`, `mTransmitBuffer1Size`, `mTransmitBuffer2Size` properties of `settings` variable; for example:

```
ACAN2515Settings settings (QUARTZ_FREQUENCY, 125 * 1000) ;
settings.mTransmitBuffer0Size = 30 ;
const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
...
```

A zero size is valid: calling the `tryToSend` method returns `true` if the corresponding `TXBi` register is empty, and `false` if it is full.

7.3 The `transmitBufferSize` method

The `transmitBufferSize` method has one argument, the index i of a driver transmit buffer ($0 \leqslant i \leqslant 2$). It returns the allocated size of this driver transmit buffer, that is the value of `settings.mTransmitBuffer`i`Size` when the `begin` method is called.

```
const uint32_t s = can.transmitBufferSize (1) ; // Driver transmit buffer 1
```

7.4 The `transmitBufferCount` method

The `transmitBufferCount` method has one argument, the index i of a driver transmit buffer ($0 \leqslant i \leqslant 2$). It returns the current number of messages in the driver transmit buffer i.

```
const uint32_t n = can.transmitBufferCount (0) ; // Driver transmit buffer 0
```

7.5 The `transmitBufferPeakCount` method

The `transmitBufferPeakCount` method has one argument, the index i of a driver transmit buffer ($0 \leqslant i \leqslant 2$). It returns the peak value of message count in the driver transmit buffer i.

```
const uint32_t max = can.transmitBufferPeakCount (2) ; // Driver transmit buffer 0
```

If the transmit buffer is full when `tryToSend` is called, the return value of this call is `false`. In such case, the following calls of `transmitBufferPeakCount(`i`)` will return `transmitBufferSize (`i`)+1`.

So, when `transmitBufferPeakCount(`i`)` returns a value lower or equal to `transmitBufferSize (`i`)`, it means that calls to `tryToSend` have allways returned `true`, and no overflow occurs on transmit buffer i.

8 Retrieving received messages using the `receive` method

There are two ways for retrieving received messages :

- using the `receive` method, as explained in this section;
- using the `dispatchReceivedMessage` method (see section 10 page 21).

This is a basic example:

```
void loop () {
  CANMessage message ;
  if (can.receive (message)) {
    // Handle received message
  }
  ...
}
```

The `receive` method:

- returns `false` if the driver receive buffer is empty, `message` argument is not modified;
- returns `true` if a message has been has been removed from the driver receive buffer, and the `message` argument is assigned.

You need to manually dispatch the received messages. If you did not provide any receive filter, you should check the `rtr` bit (remote or data frame?), the `ext` bit (standard or extended frame), and the `id` (identifier value). The following snippet dispatches three messages:

```
void loop () {
  CANMessage message ;
  if (can.receive (message)) {
    if (!message.rtr && message.ext && (message.id == 0x123456)) {
      handle_myMessage_0 (message) ; // Extended data frame, id is 0x123456
    }else if (!message.rtr && !message.ext && (message.id == 0x234)) {
      handle_myMessage_1 (message) ;  // Standard data frame, id is 0x234
    }else if (message.rtr && !message.ext && (message.id == 0x542)) {
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
    }
  }
  ...
}
```

The `handle_myMessage_0` function has the following header:

```
void handle_myMessage_0 (const CANMessage & inMessage) {
  ...
}
```

So are the header of the `handle_myMessage_1` and the `handle_myMessage_2` functions.

8.1 Driver receive buffer size

By default, the driver receive buffer size is 32. You can change it by setting the `mReceiveBufferSize` property of `settings` variable before calling the `begin` method:

```
ACAN2515Settings settings (QUARTZ_FREQUENCY, 125 * 1000) ;
settings.mReceiveBufferSize = 100 ;
const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
...
```

As the size of `CANMessage` class is 16 bytes, the actual size of the driver receive buffer is the value of `settings.mReceiveBufferSize * 16`.

8.2 The `receiveBufferSize` method

The `receiveBufferSize` method returns the size of the driver receive buffer, that is the value of the `mReceiveBufferSize` property of `settings` variable when the the `begin` method is called.

```
const uint32_t s = can.receiveBufferSize () ;
```

8.3 The `receiveBufferCount` method

The `receiveBufferCount` method returns the current number of messages in the driver receive buffer.

```
const uint32_t n = can.receiveBufferCount () ;
```

8.4 The `receiveBufferPeakCount` method

The `receiveBufferPeakCount` method returns the peak value of message count in the driver receive buffer.

```
const uint32_t max = can.receiveBufferPeakCount () ;
```

Note the driver receive buffer can overflow, if messages are not retrieved (by calling the `receive` or the `dispatchReceivedMessage` methods). If an overflow occurs, further calls of `can.receiveBufferPeakCount ()` return `can.receiveBufferSize ()+1`.

9 Acceptance filters

It is recommended to read the Microchip documentation `DS20001801H`, section 4.5 page 33. The figure 5 shows the `MCP2515` acceptance filter registers.

9.1 Default behaviour

The `can.begin (settings, [] can.isr () ; )` method sets the `RXM0` and `RXM1` registers to 0, so, the `MCP2515` receives all CAN bus messages.

More precisely, as `RXM0` is zero, all messages are received in `RXB0`. If a new message is received as `RXB0` is full, the new message is lost.

You can set the `mRolloverEnable` property of your `ACAN2515Settings` object to `true` (it is `false` by default). Doing that, if a new message is received as `RXB0` is full, the new message is transferred to `RXB1`. If `RXB1` is full the new message is lost.

9.2 Defining filters

Sample sketch: the `loopbackUsingFilters` sketch shows how defining filters.

For defining filters, you should:

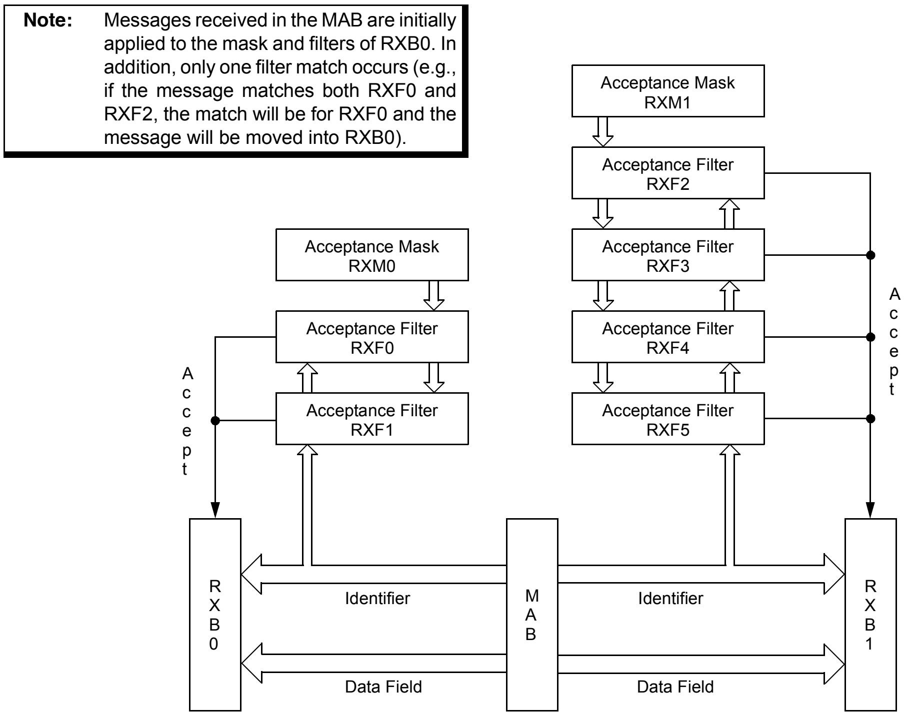

Figure 5 – `MCP2515` acceptance filters (`DS20001801H`, figure 4.2 page 25)

- define the values for the `RXM0` and `RXM1` acceptance masks;
- submitting an `ACAN2515AcceptanceFilter` array to the `ACAN2515::begin` method.

The `ACAN2515AcceptanceFilter` array defines the values that the `ACAN2515::begin` method sets to the `RXF`*i* acceptance filter registers.

Four functions are available for managing filters:

- `standard2515Mask` and `extended2515Mask` functions for defining `RXM`*i* value;
- `standard2515Filter` and `extended2515Filter` functions for defining `RXF`*i* value.

`RXM`*i* and `RXF`*i* values you handle are `ACAN2515Mask` class instances, that provides four `uint8_t` properties: `mSIDH`, `mSIDL`, `mEID8`, `mEID0`. They correspond to the `MCP2515` registers. If you want, you can set directly these properties, without using the above functions.

Filter remote and data frames. The `MCP2515` filters do not handle the `RTR` bit: for example, you cannot specify you want to accept data frames and discard remote frames. This should be done by your code.

Multiple filter matches. From `DS20001801H`, section 4.5.4 page 34: *If more than one acceptance filter matches, the FILHITn bits will encode the binary value of the lowest numbered filter that matched. For example, if filters, RXF2 and RXF4, match, the FILHITn bits will be loaded with the value for RXF2. This essentially prioritizes the acceptance filters with a lower numbered filter having higher priority. Messages are compared to filters in ascending order of filter number. This also ensures that the message will only be received into one buffer. This implies that RXB0 has a higher priority than RXB1.*

The MCP2515 filters cannot be disabled, so all mask registers can be taken into account during the acceptance of a message. For example, if MCP2515 filters are defined with the RXM0, RXF0, RXF1 registers, leaving RXM1 equal to 0 provides the transfer to RXB1 of all messages discarded by RXF0 and RXF1.

For dealing with all situations, the ACAN2515::begin method accepts three prototypes.

No filter.

```
ACAN2515Settings settings (QUARTZ_FREQUENCY, 125 * 1000) ;
const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
```

No filter is provided, RXM0 and RXM1 are set to 0, enabling the acceptance of all messages by RXB0.

One filter. For example:

```
ACAN2515Settings settings (QUARTZ_FREQUENCY, 125 * 1000) ;
const ACAN2515Mask rxm0 = extended2515Mask (0x1FFFFFFF) ;
const ACAN2515AcceptanceFilter filter [] = {
  {extended2515Filter (0x12345678), receive0} // RXF0
} ;
const uint16_t errorCode = can.begin (settings,
                                      [] { can.isr () ; },
                                      rxm0, // Value set to RXM0 register
                                      filter, // The filter array
                                      1) ; // Filter array size
```

Here, one type of message is accepted, extended (data or remote) frames with an identifier equal to 0x12345678. This defines explicitly RXM0 and RXF0; for disabling acceptance by RXF1, it is set with RXF0 value; RXM1 is set with RXM0 value, and the RXF2 to RXF5 registers are set with the RXF0 value. No message will be accepted by RXB1 filters.

The definition of a filter is associated with a call back function – here receive0. This function is called indirectly when the dispatchReceivedMessage method is called – see section 10 page 21.

Two filters. For example:

```
ACAN2515Settings settings (QUARTZ_FREQUENCY, 125 * 1000) ;
const ACAN2515Mask rxm0 = extended2515Mask (0x1FFFFFFF) ;
const ACAN2515AcceptanceFilter filters [] = {
  {extended2515Filter (0x12345678), receive0}, // RXF0
  {extended2515Filter (0x18765432), receive1}  // RXF1
} ;
const uint16_t errorCode = can.begin (settings,
                                      [] { can.isr () ; },
                                      rxm0, // Value set to RXM0 register
                                      filters, // The filter array
                                      2) ; // Filter array size
```

Here, two types of message are accepted, extended (data or remote) frames with an identifier equal to 0x12345678 or 0x18765432. This defines explicitly RXM0, RXF0 and RXF1; RXM1 is set with RXM0 value, and the RXF2 to RXF5 registers are set with the RXF1 value. No message will be accepted by RXB1 filters.

Three to five filters. For example, with four filters:

```
ACAN2515Settings settings (QUARTZ_FREQUENCY, 125 * 1000) ;
const ACAN2515Mask rxm0 = extended2515Mask (0x1FFFFFFF) ;
const ACAN2515Mask rxm1 = standard2515Mask (0x7FF, 0, 0) ;
const ACAN2515AcceptanceFilter filters [] = {
  {extended2515Filter (0x12345678), receive0}, // RXF0
  {extended2515Filter (0x18765432), receive1}, // RXF1
  {standard2515Filter (0x567, 0, 0), receive2},// RXF2
  {standard2515Filter (0x123, 0, 0), receive3} // RXF3
} ;
const uint16_t errorCode = can.begin (settings,
                                      [] { can.isr () ; },
                                      rxm0, // Value set to RXM0 register
                                      rxm1, // Value set to RXM1 register
                                      filters, // The filter array
                                      4) ; // Filter array size
```

Four types of message are accepted, extended (data or remote) frames with an identifier equal to `0x12345678` or `0x18765432`, and standard (data or remote) frames with an identifier equal to `0x567` or `0x123`. The `RXF4` and `RXF5` registers are set with the `RXF3` value.

Six filters.

```
ACAN2515Settings settings (QUARTZ_FREQUENCY, 125 * 1000) ;
const ACAN2515Mask rxm0 = extended2515Mask (0x1FFFFFFF) ;
const ACAN2515Mask rxm1 = standard2515Mask (0x7FF, 0, 0) ;
const ACAN2515AcceptanceFilter filters [] = {
  {extended2515Filter (0x12345678), receive0}, // RXF0
  {extended2515Filter (0x18765432), receive1}, // RXF1
  {standard2515Filter (0x567, 0, 0), receive2},// RXF2
  {standard2515Filter (0x123, 0, 0), receive3},// RXF3
  {standard2515Filter (0x777, 0, 0), receive4},// RXF4
  {standard2515Filter (0x3AB, 0, 0), receive5} // RXF5
} ;
const uint16_t errorCode = can.begin (settings,
                                      [] { can.isr () ; },
                                      rxm0, // Value set to RXM0 register
                                      rxm1, // Value set to RXM1 register
                                      filters, // The filter array
                                      6) ; // Filter array size
```

Six types of message are accepted, all filter registers are explicitly defined.

9.2.1 Extended frames acceptance

The `extended2515Mask` and `extended2515Filter` functions helps you to define extended frame filters. Extended frame filters test extended identifier value.

The acceptance criterion is[7]:

[7]See DS20001801H, section 4.5 *Message Acceptance Filters and Masks*, page 33.

$$\texttt{acceptance\_mask} \; \& \; (\texttt{received\_identifier} \; \texttt{nXOR} \; \texttt{acceptance\_filter}) == 0$$

where & is the bit-wise *and* operator, and `nXOR` is the *not xor* bit-wise operator.

Accepting all extended frames.

```
const ACAN2515Mask rxm0 = extended2515Mask (0) ;
```

No extended frame identifier bit is tested, all extended frames are accepted.

Accepting individual extended frames.

```
const ACAN2515Mask rxm0 = extended2515Mask (0x1FFFFFFF) ;
```

All extended frame identifier bits are tested, only extended frames whose identifiers match the filters are accepted.

Accepting several identifiers. The bits at 0 of the mask correspond to bits that are not tested for acceptance. For example:

```
const ACAN2515Mask rxm0 = extended2515Mask (0x1FFFFF0F) ;
```

If you define an acceptance filter by `extended2515Filter (0x12345608)`, any extended frame with an identifier equal to `0x123456`x`8` is accepted.

9.2.2 Standard frames acceptance

The `standard2515Mask` and `standard2515Filter` functions helps you to define extended frame filters. Standard frame filters test standard identifier value, first and second data byte.

The acceptance criterion is[8]:

$$\texttt{acceptance\_mask} \; \& \; ((\texttt{received\_identifier}, \; \texttt{data\_byte}_0, \; \texttt{data\_byte}_1) \; \texttt{nXOR} \; \texttt{acceptance\_filter}) == 0$$

where & is the bit-wise *and* operator, and `nXOR` is the *not xor* bit-wise operator.

Accepting all standard frames, without testing data bytes.

```
const ACAN2515Mask rxm0 = standard2515Mask (0, 0, 0) ;
```

Accepting individual standard frames, without testing data bytes.

```
const ACAN2515Mask rxm0 = standard2515Mask (0x7FF, 0, 0) ;
```

All standard frame identifier bits are tested, only standard frames whose identifiers match the filters are accepted.

Accepting several identifiers, without testing data bytes. The bits at 0 of the mask correspond to bits that are not tested for acceptance. For example:

```
const ACAN2515Mask rxm0 = standard2515Mask (0x70F, 0, 0) ;
```

[8] See DS20001801H, section 4.5 *Message Acceptance Filters and Masks*, page 33.

If you define an acceptance filter by `standard2515Filter (0x40A, 0, 0)`, any standard frame with an identifier equal to `0x4`x`A` is accepted.

Filtering from first data byte. The second argument of `standard2515Mask` specify first data byte filtering. For example:

```
const ACAN2515Mask rxm0 = standard2515Mask (0x70F, 0xFF, 0) ;
```

If you define an acceptance filter by `standard2515Filter (0x40A, 0x54, 0)`, any standard frame with an identifier equal to `0x4`x`A` and first byte equal to `0x54` is accepted.

Empty standard frame. An empty standard frame (without any data byte) is accepted, the filtering condition on the first data byte is ignored (see `loopbackFilterDataByte` sample sketch).

10 The `dispatchReceivedMessage` method

Sample sketch: the `loopbackUsingFilters` shows how using the `dispatchReceivedMessage` method.

Instead of calling the `receive` method, call the `dispatchReceivedMessage` method in your `loop` function. It calls the call back function associated with the matching filter.

If you have not defined any filter, do not use this function, call the `receive` method.

```
void loop () {
  can.dispatchReceivedMessage () ; // Do not use can.receive any more
  ...
}
```

The `dispatchReceivedMessage` method handles one message at a time. More precisely:

- if it returns `false`, the driver receive buffer was empty;
- if it returns `true`, the driver receive buffer was not empty, one message has been removed and dispatched.

So, the return value can used for emptying and dispatching all received messages:

```
void loop () {
  while (can.dispatchReceivedMessage ()) {
  }
  ...
}
```

If a filter definition does not name a call back function, the corresponding messages are lost.

The `dispatchReceivedMessage` method has an optional argument – `NULL` by default: a function name. This function is called for every message that pass the receive filters, with an argument equal to the matching filter index:

```
void filterMatchFunction (const uint32_t inFilterIndex) {
  ...
}

void loop () {
  can.dispatchReceivedMessage (filterMatchFunction) ;
  ...
}
```

You can use this function for maintaining statistics about receiver filter matches.

11 The `ACAN2515::begin` method reference

11.1 The `ACAN2515::begin` method prototypes

There are three `begin` method prototypes:

```
uint32_t ACAN2515::begin (const ACAN2515Settings & inSettings,
                          void (* inInterruptServiceRoutine) (void)) ;
```

```
uint32_t ACAN2515::begin (const ACAN2515Settings & inSettings,
                          void (* inInterruptServiceRoutine) (void),
                          const ACAN2515Mask inRXM0,
                          const ACAN2515AcceptanceFilter inAcceptanceFilters [],
                          const uint32_t inAcceptanceFilterCount) ;
```

```
uint32_t ACAN2515::begin (const ACAN2515Settings & inSettings,
                          void (* inInterruptServiceRoutine) (void),
                          const ACAN2515Mask inRXM0,
                          const ACAN2515Mask inRXM1,
                          const ACAN2515AcceptanceFilter inAcceptanceFilters [],
                          const uint32_t inAcceptanceFilterCount) ;
```

11.2 Defining explicitly the interrupt service routine

In this document, the *interrupt service routine* is defined by a lambda expression:

```
  const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
```

Instead of a lambda expression, you are free to define the *interrupt service routine* as a function:

```
void canISR () {
  can.isr () ;
}
```

And you pass `canISR` as argument to the `begin` method:

```
const uint16_t errorCode = can.begin (settings, canISR) ;
```

11.3 The error code

The `ACAN2515::begin` method returns an error code. The value `0` denotes no error. Otherwise, you consider every bit as an error flag, as described in table 1. An error code could report several errors. The `ACAN2515` class defines static constants for naming errors.

Bit	Static constant Name	Link
0	`kNoMCP2515`	section 11.3.1 page 23
1	`kTooFarFromDesiredBitRate`	section 11.3.2 page 23
2	`kInconsistentBitRateSettings`	section 11.3.3 page 23
3	`kINTPinIsNotAnInterrupt`	section 11.3.4 page 24
4	`kISRIsNull`	section 11.3.5 page 24
5	`kRequestedModeTimeOut`	section 11.3.6 page 24
6	`kAcceptanceFilterArrayIsNULL`	section 11.3.7 page 24
7	`kOneFilterMaskRequiresOneOrTwoAcceptanceFilters`	section 11.3.8 page 24
8	`kTwoFilterMasksRequireThreeToSixAcceptanceFilters`	section 11.3.9 page 24

Table 1 – The `ACAN2515::begin` method error code bits

11.3.1 `kNoMCP2515`

The `ACAN2515::begin` method checks accessibility by writing and reading back the `CNF1_REGISTER` first with the `0x55` value, then with the `0xAA` value. This error is raised when the read value is different from the written one. It means that the `MCP2515` cannot be accessed via SPI.

11.3.2 `kTooFarFromDesiredBitRate`

This error occurs when the `mBitRateClosedToDesiredRate` property of the `settings` object is `false`. This means that the `ACAN2515Settings` constructor cannot compute a CAN bit configuration close enough to the desired bit rate. For example:

```
void setup () {
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 1) ; // 1 bit/s !!!
  // Here, settings.mBitRateClosedToDesiredRate is false
  const uint16_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
  // Here, errorCode contains ACAN2515::kCANBitConfigurationTooFarFromDesiredBitRate
}
```

11.3.3 `kInconsistentBitRateSettings`

The `ACAN2515Settings` constructor allways returns consistent bit rate settings – even if the settings provide a bit rate too far away the desired bit rate. So this error occurs only when you have changed the

CAN bit properties (`mBitRatePrescaler`, `mPropagationSegment`, `mPhaseSegment1`, `mPhaseSegment2`, `mSJW`), and one or more resulting values are inconsistent. See section 12.3 page 30.

11.3.4 `kINTPinIsNotAnInterrupt`

The pin you provide for handling the `MCP2515` interrupt has no interrupt capability.

11.3.5 `kISRIsNull`

The interrupt service routine argument is `NULL`, you should provide a valid function.

11.3.6 `kRequestedModeTimeOut`

During configuration by the `ACAN2515::begin` method, the `MCP2515` is in the *configuration* mode. At this end of this process, the mode specified by the `inSettings.mRequestedMode` value is requested. The switch to this mode is not immediate, a register is repetitively read for checking the switch is done. This error is raised if the switch is not completed within a delay between 1 ms and 2 ms.

11.3.7 `kAcceptanceFilterArrayIsNULL`

The `ACAN2515::begin` method you have called names the `inAcceptanceFilters` argument, but it is `NULL`.

11.3.8 `kOneFilterMaskRequiresOneOrTwoAcceptanceFilters`

The `ACAN2515::begin` method you have called names the `inRXM0` argument (but not `inRXM1`), you should provide the value 1 or 2 to the `inAcceptanceFilterCount` argument.

11.3.9 `kTwoFilterMasksRequireThreeToSixAcceptanceFilters`

The `ACAN2515::begin` method you have called names the `inRXM0` and the the `inRXM1` arguments, you should provide the value 3 to 6 to the `inAcceptanceFilterCount` argument.

12 `ACAN2515Settings` class reference

Note. The `ACAN2515Settings` class is not Arduino specific. You can compile it on your desktop computer with your favorite C++ compiler. In the `https://github.com/pierremolinaro/acan2515-dev` GitHub repository, a command line tool is defined for exploring all CAN bit rates from 1 bit/s and 20 Mbit/s for a 16 MHz quartz: 63810 bit rates are valid, and 29 are exact. It also checks that computed CAN bit decompositions are all consistent, even if they are too far from the desired baud rate.

12.1 First ACAN2515Settings constructor: computation of the CAN bit settings

The constructor of the `ACAN2515Settings` has two mandatory arguments: the quartz frequency, and the desired bit rate. It tries to compute the CAN bit settings for this bit rate. If it succeeds, the constructed object has its `mBitRateClosedToDesiredRate` property set to `true`, otherwise it is set to `false`. For example:

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 1 * 1000 * 1000) ; // 1 Mbit/s
  // Here, settings.mBitRateClosedToDesiredRate is true
  ...
}
```

Of course, with a 16 MHz quartz, CAN bit computation allways succeeds for classical bit rates: 1 Mbit/s, 500 kbit/s, 250 kbit/s, 125 kbit/s. But CAN bit computation can also succeed for some unusual bit rates, as 727 kbit/s. You can check the result by computing actual bit rate, and the distance from the desired bit rate:

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 727 * 1000) ; // 727 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("actual␣bit␣rate:␣") ;
  Serial.println (settings.actualBitRate ()) ; //  727272 bit/s
  Serial.print ("distance:␣") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 375 ppm
  ...
}
```

The actual bit rate is 727,272 bit/s, and its distance from desired bit rate is 375 ppm. ”ppm” stands for ”part-per-million”, and 1 $\mathtt{ppm} = 10^{-6}$. In other words, $10,000$ $\mathtt{ppm} = 1\%$.

By default, a desired bit rate is accepted if the distance from the computed actual bit rate is lower or equal to $1,000$ $\mathtt{ppm} = 0.1$ %. You can change this default value by adding your own value as third argument of `ACAN2515Settings` constructor:

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 727 * 1000, 100) ;
  Serial.print ("mBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual␣bit␣rate:␣") ;
  Serial.println (settings.actualBitRate ()) ; //  727272 bit/s
  Serial.print ("distance:␣") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 375 ppm
```

```
  ...
}
```

The third argument does not change the CAN bit computation, it only changes the acceptance test for setting the `mBitRateClosedToDesiredRate` property. For example, you can specify that you want the computed actual bit to be exactly the desired bit rate:

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 500 * 1000, 0) ; // Max distance is 0 ppm
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  500,000 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 0 ppm
  ...
}
```

In any way, the bit rate computation allways gives a consistent result, resulting an actual bit rate closest from the desired bit rate. For example:

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 10,100 ppm
  ...
}
```

You can get the details of the CAN bit decomposition. For example:

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 10,100 ppm
```

```
  Serial.print ("Bit rate prescaler: ") ;
  Serial.println (settings.mBitRatePrescaler) ; // BRP = 1
  Serial.print ("Propagation segment: ") ;
  Serial.println (settings.mPropagationSegment) ; // PropSeg = 6
  Serial.print ("Phase segment 1: ") ;
  Serial.println (settings.mPhaseSegment1) ; // PS1 = 5
  Serial.print ("Phase segment 2: ") ;
  Serial.println (settings.mPhaseSegment2) ; // PS2 = 6
  Serial.print ("Resynchronization Jump Width: ") ;
  Serial.println (settings.mSJW) ; // SJW = 4
  Serial.print ("Triple Sampling: ") ;
  Serial.println (settings.mTripleSampling) ; // 0, meaning single sampling
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 68, meaning 68%
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

The `samplePointFromBitStart` method returns sample point, expressed in per-cent of the bit duration from the beginning of the bit.

Note the computation may calculate a bit decomposition too far from the desired bit rate, but it is allways consistent. You can check this by calling the `CANBitSettingConsistency` method.

You can change the property values for adapting to the particularities of your CAN network propagation time. By example, you can increment the `mPhaseSegment1` value, and decrement the `mPhaseSegment2` value in order to sample the `CAN Rx` pin later.

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  settings.mPhaseSegment1 ++ ; // 5 -> 6: safe, 1 <= PS1 <= 8
  settings.mPhaseSegment2 -- ; // 5 -> 4: safe, 2 <= PS2 <= 8 and SJW <= PS2
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 75, meaning 75%
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; // 500000: ok, bit rate did not change
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

Be aware to allways respect CAN bit timing consistency! The constraints are:

$$
\begin{aligned}
1 \leqslant \texttt{mBitRatePrescaler} \leqslant 64 \\
1 \leqslant \texttt{mSJW} \leqslant 4 \\
1 \leqslant \texttt{mPropagationSegment} \leqslant 8 \\
\text{Single sampling: } 1 \leqslant \texttt{mPhaseSegment1} \leqslant 8 \\
\text{Triple sampling: } 2 \leqslant \texttt{mPhaseSegment1} \leqslant 8 \\
2 \leqslant \texttt{mPhaseSegment2} \leqslant 8 \\
\texttt{mSJW} < \texttt{mPhaseSegment2} \\
\texttt{mPhaseSegment2} \leqslant \texttt{mPropagationSegment} + \texttt{mPhaseSegment1}
\end{aligned}
$$

Resulting actual bit rate is given by:

$$
\text{Actual bit rate} = \frac{QuartzFrequency \;/\; 2}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2})}
$$

And sampling points (in per-cent unit) are given by:

$$
\text{Sampling point } \textit{(single sampling)} = 100 \cdot \frac{1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1}}{1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2}}
$$

$$
\text{Sampling first point } \textit{(triple sampling)} = 100 \cdot \frac{\texttt{mPropagationSegment} + \texttt{mPhaseSegment1}}{1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2}}
$$

12.2 Second ACAN2515Settings constructor: explicit CAN bit settings

New in release 1.0.4. This ACAN2515Settings constructor defines explicitly CAN bit settings. For example, see the LoopBackDemoBitRateSettings sketch :

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ACAN2515Settings settings (QUARTZ_FREQUENCY, // For computing actual bit rate
                             4, // Bit rate prescaler, 1...64
                             5, // Propagation Segment, 1...8
                             5, // Phase Segment1, 1...8
                             5, // Phase Segment2, 2...8
                             4) ; // SJW, 1...4
  ...
}
```

This constructor requires six arguments :

1. inQuartzFrequency: the quartz frequency (uint32_t); note the quartz frequency is only used for computing actual bit rate;

2. `inBitRatePrescaler`: the bit rate prescaler (`uint8_t`);
3. `inPropagationSegment`: the propagation segment (`uint8_t`);
4. `inPhaseSegment1`: the phase segment 1 (`uint8_t`);
5. `inPhaseSegment2`: the phase segment 2 (`uint8_t`);
6. `inSJW`: the Synchronization Jump Width (`uint8_t`).

By default, *single sampling* is selected. Set `mTripleSampling` to `true` is you want *triple sampling*.

Respect the `MCP2515` constraints:

$$\begin{aligned}
1 &\leqslant \texttt{inBitRatePrescaler} \leqslant 64 \\
1 &\leqslant \texttt{inSJW} \leqslant 4 \\
1 &\leqslant \texttt{inPropagationSegment} \leqslant 8 \\
\text{Single sampling: } 1 &\leqslant \texttt{inPhaseSegment1} \leqslant 8 \\
\text{Triple sampling: } 2 &\leqslant \texttt{inPhaseSegment1} \leqslant 8 \\
2 &\leqslant \texttt{inPhaseSegment2} \leqslant 8 \\
\texttt{inSJW} &< \texttt{inPhaseSegment2} \\
\texttt{inPhaseSegment2} &\leqslant \texttt{inPropagationSegment} + \texttt{inPhaseSegment1}
\end{aligned}$$

Call the `CANBitSettingConsistency` method (section 12.3 page 30) for checking your bit setting is consistent. Note the `ACAN2515::begin` method does this.

You can use this constructor for several reasons:

- you need a specific bit setting that the algorithm of the previous constructor cannot provide;
- you want to save program memory.

The algorithm of the previous constructor requires 32-bit arithmetic, that is expensive for a 8-bit processor as the Arduino Uno's one. The table 2 lists the program sizes of the `LoopBackDemo` and `LoopBackDemoBitRateSettings` sketches, for several platforms. The Teensy 3.5 settings are: USB Serial, 120 MHz, Smallest code with LTO.

Platform	**Sketch LoopBackDemo**	**Sketch LoopBackDemoBitRateSettings**
Arduino Uno	7 600 bytes	6 410 bytes
Adafruit Feather M0	15 976 bytes	15 656 bytes
Teensy 3.5	14 004 bytes	13 524 bytes

Table 2 – Sketch program sizes

A starting point for obtaining the bit setting parameters is to execute the first constructor and note the values it provides. For example, run the `LoopBackDemo` sketch, it displays in the serial monitor the bit setting values that you can then use in the `LoopBackDemoBitRateSettings` sketch.

You can also write a program for your desktop computer: the `ACAN2515Settings` class is not Arduino specific.

12.3 The CANBitSettingConsistency method

This method checks the CAN bit decomposition (given by mBitRatePrescaler, mPropagationSegment, mPhaseSegment1, mPhaseSegment2, mSJW property values) is consistent.

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  settings.mPhaseSegment1 = 0 ; // Error, mPhaseSegment1 should be >= 1 (and <= 8)
  Serial.print ("Consistency: 0x") ;
  Serial.println (settings.CANBitSettingConsistency (), HEX) ; // 0x10, meaning error
  ...
}
```

The CANBitSettingConsistency method returns 0 if CAN bit decomposition is consistent. Otherwise, the returned value is a bit field that can report several errors – see table 3.

The ACAN2515Settings class defines static constant properties that can be used as mask error. For example:

```
public: static const uint32_t kBitRatePrescalerIsZero = 1 << 0 ;
```

Bit	Error Name	Error
0	kBitRatePrescalerIsZero	mBitRatePrescaler == 0
1	kBitRatePrescalerIsGreaterThan64	mBitRatePrescaler > 64
2	kPropagationSegmentIsZero	mPropagationSegment == 0
3	kPropagationSegmentIsGreaterThan8	mPropagationSegment > 8
4	kPhaseSegment1IsZero	mPhaseSegment1 == 0
5	kPhaseSegment1IsGreaterThan8	mPhaseSegment1 > 8
6	kPhaseSegment2IsLowerThan2	mPhaseSegment2 < 2
7	kPhaseSegment2IsGreaterThan8	mPhaseSegment2 > 8
8	kPhaseSegment1Is1AndTripleSampling	(mPhaseSegment1 == 1) && mTripleSampling
9	kSJWIsZero	mSJW == 0
10	kSJWIsGreaterThan4	mSJW > 4
11	kSJWIsGreaterThanOrEqualToPhaseSegment2	mSJW >= mPhaseSegment2
12	kPhaseSegment2IsGreaterThanPSPlusPS1	mPhaseSegment2 > (mPropagationSegment + mPhaseSegment1)

Table 3 – The ACAN2515Settings::CANBitSettingConsistency method error codes

12.4 The actualBitRate method

The actualBitRate method returns the actual bit computed from mBitRatePrescaler, mPropagationSegment, mPhaseSegment1, mPhaseSegment2, mSJW property values.

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual␣bit␣rate:␣") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  ...
}
```

Note. If CAN bit settings are not consistent (see section 12.3 page 30), the returned value is irrelevant.

12.5 The exactBitRate method

The `exactBitRate` method returns `true` if the actual bit rate is equal to the desired bit rate, and `false` otherwise.

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 727 * 1000) ; // 727 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("actual␣bit␣rate:␣") ;
  Serial.println (settings.actualBitRate ()) ; //  727272 bit/s
  Serial.print ("distance:␣") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 375 ppm
  Serial.print ("Exact:␣") ;
  Serial.println (settings.exactBitRate ()) ; // 0 (---> false)
  ...
}
```

Note. If CAN bit settings are not consistent (see section 12.3 page 30), the returned value is irrelevant.

For a 16 MHz clock, the 28 exact bit rates are: 5 kbit/s, 6250 bit/s, 6400 bit/s, 8 kbit/s, 10 kbit/s, 12500 bit/s, 12800 bit/s, 15625 bit/s, 16 kbit/s, 20 kbit/s, 25 kbit/s, 31250 bit/s, 32 kbit/s, 40 kbit/s, 50 kbit/s, 62500 bit/s, 64 kbit/s, 80 kbit/s, 100 kbit/s, 125 kbit/s, 160 kbit/s, 200 kbit/s, 250 kbit/s, 320 kbit/s, 400 kbit/s, 500 kbit/s, 800 kbit/s, 1000 kbit/s.

For a 10 MHz clock, the 24 exact bit rates are: 3125 bit/s, 4 kbit/s, 5 kbit/s, 6250 bit/s, 8 kbit/s, 10 kbit/s, 12500 bit/s, 15625 bit/s, 20 kbit/s, 25 kbit/s, 31250 bit/s, 40 kbit/s, 50 kbit/s, 62500 bit/s, 78125 bit/s, 100 kbit/s, 125 kbit/s, 156250 bit/s, 200 kbit/s, 250 kbit/s, 312500 bit/s, 500 kbit/s, 625 kbit/s, 1000 kbit/s.

For a 8 MHz clock, the 28 exact bit rates are: 2500 bit/s, 3125 bit/s, 3200 bit/s, 4 kbit/s, 5 kbit/s, 6250 bit/s, 6400 bit/s, 8 kbit/s, 10 kbit/s, 12500 bit/s, 15625 bit/s, 16 kbit/s, 20 kbit/s, 25 kbit/s, 31250 bit/s, 32 kbit/s, 40 kbit/s, 50 kbit/s, 62500 bit/s, 80 kbit/s, 100 kbit/s, 125 kbit/s, 160 kbit/s, 200 kbit/s, 250 kbit/s, 400 kbit/s, 500 kbit/s, 800 kbit/s.

Note an 1 Mbit/s bit rate cannot be performed with a 8 MHz clock.

12.6 The ppmFromDesiredBitRate method

The `ppmFromDesiredBitRate` method returns the distance from the actual bit rate to the desired bit rate, expressed in part-per-million (ppm): $1\ \texttt{ppm} = 10^{-6}$. In other words, $10,000\ \texttt{ppm} = 1\%$.

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 727 * 1000) ; // 727 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("actual␣bit␣rate:␣") ;
  Serial.println (settings.actualBitRate ()) ; //  727272 bit/s
  Serial.print ("distance:␣") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 375 ppm
  ...
}
```

Note. If CAN bit settings are not consistent (see section 12.3 page 30), the returned value is irrelevant.

12.7 The samplePointFromBitStart method

The `samplePointFromBitStart` method returns the distance of sample point from the start of the CAN bit, expressed in part-per-cent (ppc): $1\ \texttt{ppc} = 1\% = 10^{-2}$. If triple sampling is selected, the returned value is the distance of the first sample point from the start of the CAN bit. It is a good practice to get sample point from 65% to 80%.

```
const uint32_t QUARTZ_FREQUENCY = 16 * 1000 * 1000 ; // 16 MHz
void setup () {
  ...
  ACAN2515Settings settings (QUARTZ_FREQUENCY, 500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("Sample␣point:␣") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 68 --> 68%
  ...
}
```

Note. If CAN bit settings are not consistent (see section 12.3 page 30), the returned value is irrelevant.

12.8 Properties of the `ACAN2515Settings` class

All properties of the `ACAN2515Settings` class are declared `public` and are initialized (table 4). The default values of properties from `mDesiredBitRate` until `mTripleSampling` corresponds to a CAN bit rate of `QUARTZ_FREQUENCY` / 64, that is 250,000 bit/s for a 16 MHz quartz.

Property	Type	Initial value	Comment
mQuartzFrequency	uint32_t	QUARTZ_FREQUENCY	
mDesiredBitRate	uint32_t	QUARTZ_FREQUENCY / 64	
mBitRatePrescaler	uint8_t	2	See section 12.1 page 25
mPropagationSegment	uint8_t	5	See section 12.1 page 25
mPhaseSegment1	uint8_t	5	See section 12.1 page 25
mPhaseSegment2	uint8_t	5	See section 12.1 page 25
mSJW	uint8_t	4	See section 12.1 page 25
mTripleSampling	bool	false	See section 12.1 page 25
mBitRateClosedToDesiredRate	bool	true	See section 12.1 page 25
mOneShotModeEnabled	bool	false	See section 12.8.1 page 33
mTXBPriority	uint8_t	0	See section 12.8.2 page 33
mRequestedMode	ACAN2515RequestedMode	NormalMode	See section 12.8.3 page 34
mCLKOUT_SOF_pin	ACAN2515CLKOUT_SOF	CLOCK	See section 12.8.4 page 34
mRolloverEnable	bool	true	See section 12.8.5 page 34
mReceiveBufferSize	uint16_t	32	See section 8.1 page 15
mTransmitBuffer0Size	uint16_t	16	See section 7.2 page 13
mTransmitBuffer1Size	uint16_t	0	See section 7.2 page 13
mTransmitBuffer2Size	uint16_t	0	See section 7.2 page 13

Table 4 – Properties of the `ACAN2515Settings` class

12.8.1 The `mOneShotModeEnabled` property

This boolean property corresponds to the `OSM` bit of the `CANCTRL` control register. It is false by default.

12.8.2 The `mTXBPriority` property

This property defines the transmit priority associated the `TXB`*i* registers:

- bits 1-0: priority of `TXB0`;
- bits 3-2: priority of `TXB1`;
- bits 5-4: priority of `TXB2`;
- bits 7-6: *unused*.

By default, its value is 0, all three `TXB`*i* registers get the same 0 priority.

12.8.3 The mRequestedMode property

This property defines the mode requested at this end of the configuration: `NormalMode` (default value), `ListenOnlyMode`, `LoopBackMode`.

12.8.4 The mCLKOUT property

This property defines signal output on the `CLKOUT/SOF` pin; possible values are: `CLOCK` (default value), `CLOCK2`, `CLOCK4`, `CLOCK8`, `SOF`, `HiZ`.

12.8.5 The mRolloverEnable property

This boolean property corresponds to the `BUKT` bit of the `RXB0CTRL` control register. If true (value by default), `RXB0` message will roll over and be written to `RXB1` if `RXB0` is full; if false, rollover is disabled.

13 CAN controller state

Two methods return the receive error counter and the transmit error counter.

13.1 The receiveErrorCounter method

```
public: uint8_t receiveErrorCounter (void) ;
```

13.2 The transmitErrorCounter method

```
public: uint8_t transmitErrorCounter (void) ;
```