ACAN2517 Arduino library
For the MCP2517FD, MCP2518FD and MCP251863 CANFD Controllers
in CAN 2.0B mode
Version 1.1.13

Pierre Molinaro

August 14, 2023

Contents

1 Versions

Version	Date	Comment
1.1.13	August 14, 2023	• Fixed maximum SPI clock frequency to 80 % of master clock frequency. • Added handling of GPIO0, GPIO1 and XSTBY, see section 19 page 40.
1.1.12	October 1, 2021	Added data_s64, data_s32, data_s16 and data_s8 to CANMessage class union members, see section 6 page 11 (thanks to tomtom0707).
1.1.11	April 21, 2021	Added Arduino Uno – MCP2518FDClick wiring scheme (thanks to soso49).
1.1.10	January 27, 2021	Fixed retransmission attempts setting bug. Added NoRetransmissionAttemptsDemoTeensy3x.ino sketch.
1.1.9	January 14, 2021	**Release 1.1.8 is broken, does not compile** (thanks to W.J. Loor).

		Improved method to read also the BDIAG0_REGISTER diagnostic register (thanks to turmary), see section 20.4 page 42.
1.1.8	May 31, 2020	Fix retransmission attempts settings (thanks to Flole)
1.1.7	April 27, 2020	Added dataFloat to CANMessage (thanks to Koryphon)
1.1.6	Sept. 19, 2019	Bug fixes. Added ACAN2517::currentOperationMode method, see section 20.1 page 41. Added ACAN2517::recoverFromRestrictedOperationMode method, see section 20.2 page 42. Added ACAN2517::errorCounters method, see section 20.3 page 42.
1.1.5	June 2, 2019	Fixed a race condition on ESP32 (thanks to Nick Kirkby).
1.1.4	March 22, 2019	Several speed enhancements (thanks to thomasfla). Fixed TxQ enable bug (thanks to danielhenz for having fixed this in ACAN2517FD). Added demo sketch LoopBackIntensiveTestTeensy3xUsingTxQ.
1.1.3	February 8, 2019	Compatibility for Arduino Uno. Added demo sketch LoopBackDemoArduinoUno.
1.1.2	February 3, 2019	Added setting mINTIsOpenDrain (section 18.7.2 page 38). Remove useless mutex (ESP32).
1.1.1	January 31, 2019	New option: no interrupt pin (section 7.5 page 16).
1.1.0	January 27, 2019	First release running on ESP32 (section 7.4 page 15).
1.0.4	January 14, 2019	Fixed mask and acceptance filters for extended messages. New LoopBackDemoTeensy3xStandardFilterTest.ino sample code for checking standard reception filters. New LoopBackDemoTeensy3xExtendedFilterTest.ino sample code for checking extended reception filters.
1.0.3	January 6, 2019	Fixed identifiers for extended messages. Updated TestWithACAN.ino sample code for checking extended message identifiers. Changed mode names. MCP2517Filters -> ACAN2517Filters
1.0.2	Nov. 3, 2018	Changed mode names.
1.0.1	October 24, 2018	Corrected typos.
1.0.0	October 23, 2018	Initial release

2 Features

The ACAN2517 library is a MCP2517FD CAN ("Controller Area Network") Controller driver for any board running Arduino.

This driver configures the MCP2517FD in CAN 2.0B mode. It does not handle the CANFD capabilities.

This library is compatible with:

- the ACAN 1.0.6 and above library (https://github.com/pierremolinaro/acan), CAN driver

for FlexCan module embedded in Teensy 3.1 / 3.2, 3.5, 3.6 microcontrollers;

- the ACAN2515 1.0.1 and above library (https://github.com/pierremolinaro/acan2515), CAN driver for MCP2515 CAN controller;
- the ACAN2517FD library (https://github.com/pierremolinaro/acan2517FD), CAN driver for the MCP2517FD and MCP2518FD CAN controllers, in CANFD mode.

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 32 reception filter registers are easily defined;
- reception filters accept call back functions;
- driver and controller transmit buffer sizes are customisable;
- driver and controller receive buffer size is customisable;
- overflow of the driver receive buffer is detectable;
- MCP2517FD internal RAM allocation is customizable and the driver checks no overflow occurs;
- *loop back*, *self reception*, *listing only* MCP2517FD controller modes are selectable.

3 MCP2517FD or MCP2518FD?

In short: I recommend using a MCP2518FD. My opinion is that the MCP2517FD has hardware bugs.

3.1 Reset

An originality of the MCP2517FD is that it has no reset pin. Resetting the MCP2517FD can only be done by software, by sending a RESET command through the SPI. But sometimes, for reasons I don't know, the reset is not done correctly.We can see this because the value returned by the ACAN2517FD::begin function is not zero (see section 17.3 page 29). Some possible errors are 0x1 (kRequestedConfigurationModeTimeOut, the MCP2517FD cannot reach the *configuration* mode), 0x40000 (kReadBackErrorWithFullSpeedSPIClock, the MCP2517FD RAM cannot be written and read back). Typically, this can happen when uploading and starting a new version of the firmware into the microcontroller. **So I recommend to always check the value returned by the ACAN2517FD::begin function is zero.** In such case, you should power off and the power on.

With a MCP2518FD, uploading and starting a new version of the firmware into the microcontroller always succeeds, but if the previous sketch has provided invalid clock setting, as enabling PLL with a 40MHz clock.

Note you should also add a pullup resistor on the nCS pin (section 7.1 page 13) with a MCP2517FD, I don't think this resistance is necessary with a MCP2518FD.

3.2 Clock

In short: I recommend using an external clock, as an integrated oscillator. Do not use a crystal oscillator.

Using a crystal oscillator may be tricky: just take a look to section 3.1.1 page 13 of the DS20005678D document, that gives few guidelines for selecting the correct crystal oscillator or ceramic resonator. This section gives very precise references for crystal oscillator and associated capacitors. Note also an *Optional Feedback Resistor* has been added in the C revision of this document, and the section 3.1.1 has been updated in the C and D revisions.

4MHz crystal oscillator. I have tried a 4MHz crystal oscillator (HC49US-FF3F18-4.0000), with two 22pF capacitors, so the clock setting is ACAN2517FDSettings::OSC_4MHz10xPLL. I noticed that a MCP2517FD worked well for a data bit rate up to 1Mbps; above 1Mbps, the MCP2517FD often enters in *Restricted Operation Mode*, but maybe it's due to internal bugs (see section 3.3 page 6). A MCP2518FD works prefectly with this oscillator.

40MHz crystal oscillator. I have also tried a 40MHz crystal oscillator (YIC-HC49US), with the same two 22pF capacitors, and the ACAN2517FD- Settings::OSC_40MHz setting. Surprisingly, the observed frequency on the OSC2 pin was… 13.3MHz! Exactly one third of 40MHz. Probably the 22pF capacitors are not appropriate. The OSC2 pin signal, observed at the oscilloscope, had a very small amplitude: 300mV.

Morality: if you choose a crystal oscillator, always observe the frequency obtained with an oscilloscope.

4MHz integrated oscillator. I use a 4MHz integrated oscillator (LFSPXO024978BULK, the supply voltage of my MCP2517FD is 3.3V), connected to OSC1. OSC2 is left unconnected. The clock setting is ACAN2517FDSettings::OSC_4MHz10xPLL. I have observed with oscilloscope the OSC1 pin signal, it has the correct frequency, and the amplitude I expected: 3.3V.

Same behaviour as with the 4MHz crystal oscillator: buggy with a MCP2517FD above 1Mbs, sucess with a MCP2518FD.

40MHz integrated oscillator. I use a 40MHz integrated oscillator (LFSPXO026068BULK. The clock setting is ACAN2517FDSettings::OSC_40MHz. I have also observed with oscilloscope the OSC1 pin signal, it has the correct frequency, and the amplitude I expected: 3.3V.

Same behaviour as with the 4MHz integrated oscillator: buggy with a MCP2517FD above 1Mbs, sucess with a MCP2518FD.

3.3 Restricted Operation Mode

In CANFD mode (not handled by this library, but the ACAN2517FD CANFD library), and for data bit rates higher than 1Mbps with a MCP2517FD, I have noticed the error counters may have not zero values (error counters can be read by the errorCounters method, see section 20.3 page 42), and the MCP2517FD enters sometimes in *Restricted Operation Mode*. The modes operation is described in DS20005678D, figure 2.1 page 9. *Restricted Operation Mode* is reached from *Normal Modes* on *System Error*, as the driver lets the SERR2LOM bit equal to 0.

System Error is described in section 10.5.6, page 63. The MCP2517FD Data Sheet Errata (DS80000792B) gives an explanation: *The SPI Interface can block the CANFD Controller module from accessing RAM*

in between SPI bytes and between the last byte and the rising edge of the nCS line during an SPI READ or SPI READ CRC instruction while accessing RAM. If the CANFD Controller module is blocked for more than TSPIMAXDLY, a TX MAB underflow or an RX MAB overflow can occur. Within the CANFD Control Field, TSPIMAXDLY is 3 NBT + 5 DBT, that is for an `1Mbps` arbitration bit rate and a data bit factor `x8` (`8Mbps`) : $3 \cdot 1\mu s + 5 \cdot 125ns = 3.625\mu s$. The challenge is to write a driver that checks these constraints. This is not easy, as transfers are made through `transfer` and `transfer16` SPI Arduino routines, and their implementation may vary from one platform to another. In the `ACAN2517` code, I have masked interruptions during transfers to minimize the delay between bytes, and to ensure that the `nCS` signal becomes inactive (high) as quickly as possible at the end of the transfer. In CAN2.0B, as bit rate is at most 1 Mbps, TSPIMAXDLY is $8\mu s$. A slow CPU may exceed this limit.

You can check current `MCP2517FD` operation mode by calling the `ACAN2517FD::currentOperationMode` function (section 20.1 page 41. It returns `7` for the *Restricted Operation Mode*. You can recover from *Restricted Operation Mode* by calling the `ACAN2517FD::recoverFromRestrictedOperationMode` function (section 20.2 page 42); however, some send or receive data has been lost.

I have never observed that a `MCP2518FD` enters the `Restricted Operation Mode`.

4 Data flow

Two figures illustrate message flow for sending and receiving CAN messages: figure 1 is the default configuration, figure 2 is the customized configuration.

4.1 Data flow in default configuration

The figure 1 illustrates message flow in the default configuration.

Sending messages. The ACAN2517 driver defines a *driver transmit FIFO* (default size: 16 messages), and configures the `MCP2517FD` with a *controller transmit FIFO* with a size of 32 messages.

A message is defined by an instance of `CANMessage` class. For sending a message, user code calls the `tryToSend` method – see section 13 page 22, and the `idx` property of the sent message should be equal to `0` (default value).

Receiving messages. The MCP2517FD *CAN Protocol Engine* transmits all correct frames to the *reception filters*. By default, they are configured as pass-all, see section 15 page 25 for configuring them. Messages that pass the filters are stored in the *Controller Reception FIFO*; its size is 32 message by default. The interrupt service routine transfers the messages from this FIFO to the *Driver Receive FIFO*. The size of the *Driver Receive Buffer* is 32 by default – see section 14.1 page 24 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 14 page 23;
- the `dispatchReceivedMessage` method if you have defined the reception filters that name a call-back function – see section 16 page 28.

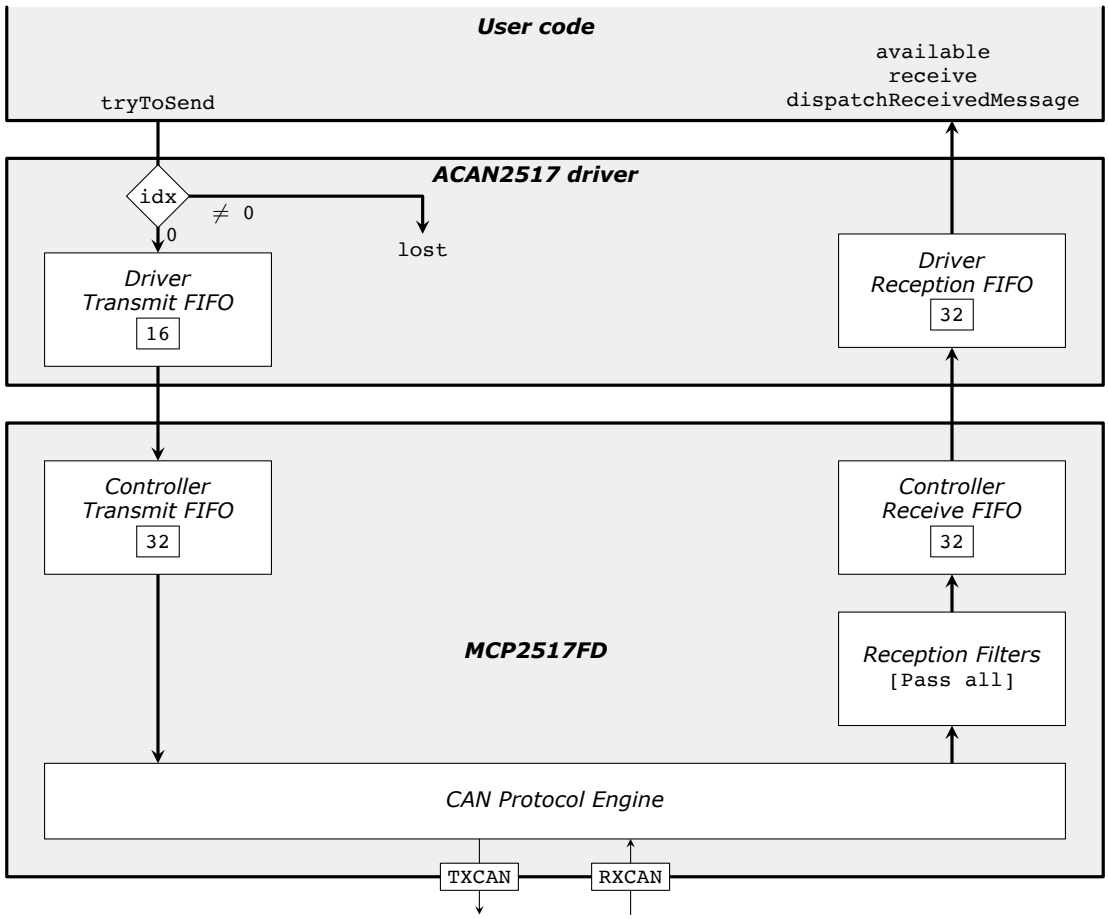

Figure 1 – Message flow in ACAN2517 driver and MCP2517FD CAN Controller, default configuration

4.2 Data flow, custom configuration

The figure 2 illustrates message flow in a custom configuration.

Note. The *transmit Event FIFO* and the `transmitEvent` function are not currently implemented.

You can allocate the *Controller transmit Queue*: send order is defined by frame priority (see section 10 page 20). You can also define up to 32 receive filters (see section 15 page 25). Sizes of MCP2517FD internal buffer are easily customizable.

5 A simple example: LoopBackDemo

The following code is a sample code for introducing the ACAN2517 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 MCP2517FD are left open), the MCP2517FD is configured in the *loop back* mode.

```
#include <ACAN2517.h>
```

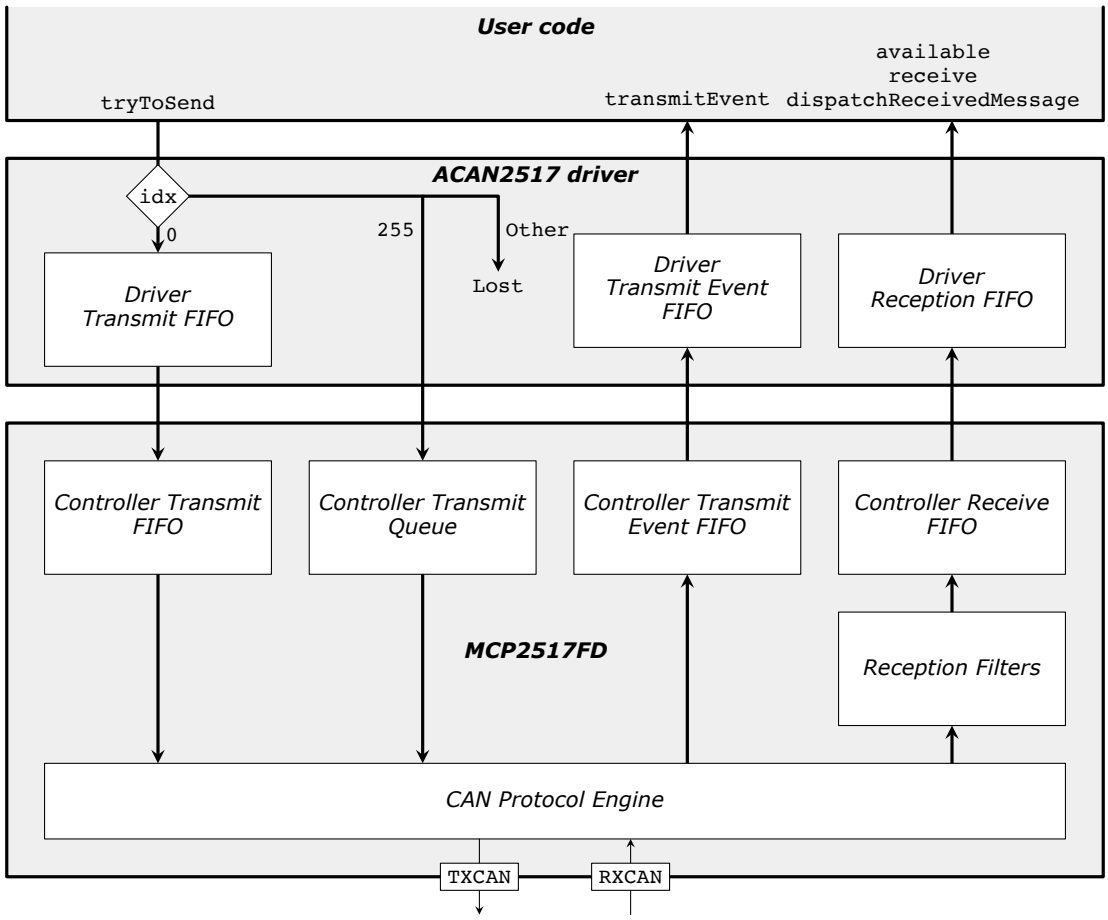

Figure 2 – Message flow in ACAN2517 driver and MCP2517FD CAN Controller, custom configuration

This line includes the ACAN2517 library.

```
static const byte MCP2517_CS  = 20 ; // CS input of MCP2517FD
static const byte MCP2517_INT = 37 ; // INT output of MCP2517FD
```

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

```
ACAN2517 can (MCP2517_CS, SPI, MCP2517_INT) ;
```

Instanciation of the ACAN2517 library, declaration and initialization of the can object that implements the driver. The constructor names: the number of the pin connected to the $\overline{\text{CS}}$ pin, the SPI object (you can use SPI1, SPI2, ...), the number of the pin connected to the $\overline{\text{INT}}$ pin.

```
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 7.2 page 13), selecting alternate pins should be done before calling `SPI.begin`, on Adafruit Feather M0 (section 7.3 page 14), this should be done after. Calling `SPI.begin` explicitly allows you to fully handle alternate pins.

```
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL, 125 * 1000) ;
```

Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACAN2517Settings` class. The constructor has two parameters: the `MCP2517FD` quartz specification, 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 = ACAN2517Settings::InternalLoopBack ;
```

This is the second step. You can override the values of the properties of `settings` object. Here, the `mRequestedMode` property is set to `InternalLoopBack` – its value is `Normal20B` 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 18.7 page 38 lists all properties you can override.

```
  const uint32_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 15 page 25. The second argument is the *interrupt service routine*, and is defined by a C++ lambda expression[1]. See section 17.2 page 29 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 17.3 page 29.

```
static uint32_t gBlinkLedDate = 0 ;
static uint32_t gReceivedFrameCount = 0 ;
static uint32_t 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 6 page 11.

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

```
  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. 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 `MCP2517FD` 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.

6 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[2] (version 1.0.3 and above) driver, the ACAN2515[3] 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 ;  // This field is used by the driver
```

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

[3]The ACAN2515 driver is a CAN driver for the MCP2515 CAN controller, https://github.com/pierremolinaro/acan2515.

```
  public : uint8_t len = 0 ;  // Length of data (0 ... 8)
  public : union {
    uint64_t data64        ; // Caution: subject to endianness
    int64_t  data_s64      ; // Caution: subject to endianness
    uint32_t data32    [2] ; // Caution: subject to endianness
    int32_t  data_s32  [2] ; // Caution: subject to endianness
    float    dataFloat [2] ; // Caution: subject to endianness
    uint16_t data16    [4] ; // Caution: subject to endianness
    int16_t  data_s16  [4] ; // Caution: subject to endianness
    int8_t   data_s8   [8] ;
    uint8_t  data      [8] = {0, 0, 0, 0, 0, 0, 0, 0} ;
  } ;
} ;
```

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, one 64-bit or two 32-bit floats. Be aware that multi-byte integers and floats 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 16 page 28);
- on sending messages, it is used for selecting the transmit buffer (see section 13 page 22).

7 Connecting a MCP2517FD to your microcontroller

Connecting a MCP2517FD requires 5 pins (figure 3):

- 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 $\overline{\text{CS}}$ signal requires one digital pin, that the driver configures as an OUTPUT ;
- connecting the $\overline{\text{INT}}$ signal requires one other digital pin, that the driver configures with INPUT_PULLUP and uses as an external interrupt input; so this pin should have interrupt capability (checked by the `begin` method of the driver object);
- the $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ signals are not used by driver and are left not connected.

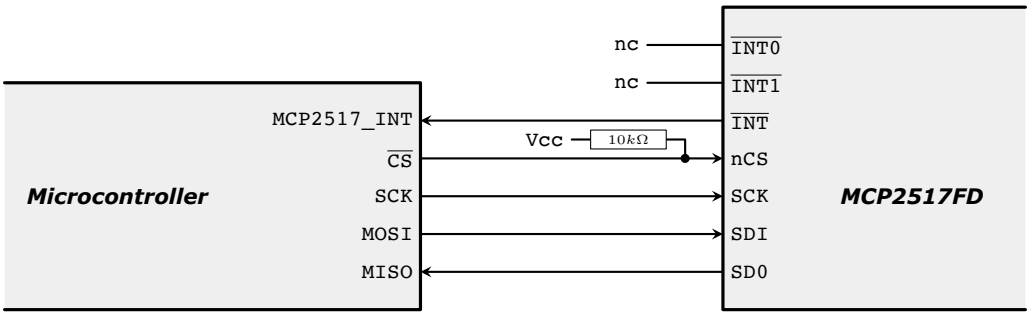

Figure 3 – MCP2517FD connection to a microcontroller

7.1 Pullup resistor on nCS pin

Note the $10\ k\Omega$ resistor between nCS and Vcc. I have experienced that this resistor is useful in the following case: a sketch using the MCP2517FD is running, and I upload a new sketch. During this process, the microcontroller is reset, leaving its $\overline{CS}$ pin floating. Without the $10\ k\Omega$ resistor, the nCS level is unpredictable, and if it becomes low, initiates transactions. I think this can crash the MCP2517FD firmware, and the following reset command sent by the driver not handled. With the resistor, the nCS level remains high until the driver sets the nCS as output.

7.2 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."*[4]

For example, the LoopBackDemoTeensy3x sketch uses SPI1 on a Teensy 3.5 with these alternate pins[5]:

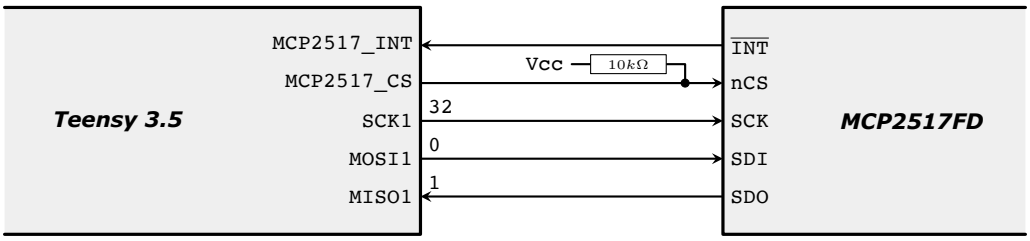

Figure 4 – Using SPI alternate pins on a Teensy 3.5

You call the SPI1.setMOSI, SPI1.setMISO, and SPI1.setSCK functions **before** calling the begin function of your ACAN2517 instance:

```
ACAN2517 can (MCP2517_CS, SPI1, MCP2517_INT) ;
...
static const byte MCP2517_SCK = 32 ; // SCK input of MCP2517
static const byte MCP2517_SDI =  0 ; // SDI input of MCP2517
static const byte MCP2517_SDO =  1 ; // SDO output of MCP2517
...
void setup () {
  ...
  SPI1.setMOSI (MCP2517_SDI) ;
  SPI1.setMISO (MCP2517_SDO) ;
  SPI1.setSCK  (MCP2517_SCK) ;
  SPI1.begin () ;
  ...
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
  ...
```

Note you can use the SPI1.pinIsMOSI, SPI1.pinIsMISO, and SPI1.pinIsSCK functions to check if the alternate pins you select are valid:

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

```
void setup () {
  ...
  Serial.print ("Using␣pin␣#") ;
  Serial.print (MCP2517_SDI) ;
  Serial.print ("␣for␣MOSI:␣") ;
  Serial.println (SPI1.pinIsMOSI (MCP2517_SDI) ? "yes" : "NO!!!") ;
  Serial.print ("Using␣pin␣#") ;
  Serial.print (MCP2517_SDO) ;
  Serial.print ("␣for␣MISO:␣") ;
  Serial.println (SPI1.pinIsMISO (MCP2517_SDO) ? "yes" : "NO!!!") ;
  Serial.print ("Using␣pin␣#") ;
  Serial.print (MCP2517_SCK) ;
  Serial.print ("␣for␣SCK:␣") ;
  Serial.println (SPI1.pinIsSCK (MCP2517_SCK) ? "yes" : "NO!!!") ;
  SPI1.setMOSI (MCP2517_SDI) ;
  SPI1.setMISO (MCP2517_SDO) ;
  SPI1.setSCK  (MCP2517_SCK) ;
  SPI1.begin () ;
  ...
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
  ...
```

7.3 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 alternate SPI pins on Adafruit Feather M0.

For example, the `LoopBackDemoAdafruitFeatherM0` sketch uses `SERCOM1` on an Adafruit Feather M0 as illustrated in figure 5.

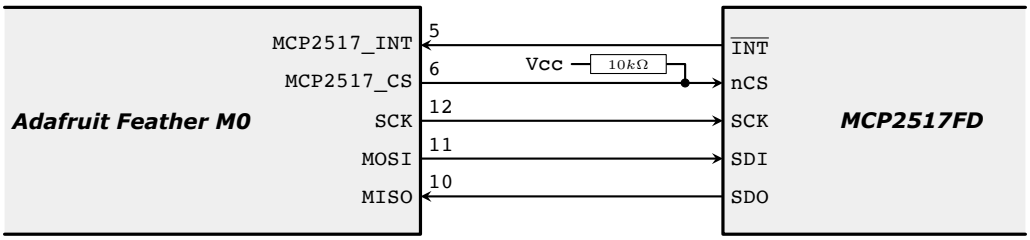

Figure 5 – Using SPI alternate pins on an Adafruit Feather M0

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 MCP2517_SCK = 12 ; // SCK pin, SCK input of MCP2517FD
static const byte MCP2517_SDI = 11 ; // MOSI pin, SDI input of MCP2517FD
static const byte MCP2517_SDO = 10 ; // MISO pin, SDO output of MCP2517FD

SPIClass mySPI (&sercom1,
                MCP2517_SDO, MCP2517_SDI, MCP2517_SCK,
                SPI_PAD_0_SCK_3, SERCOM_RX_PAD_2);
```

```
static const byte MCP2517_CS  =  6 ; // CS input of MCP2517FD
static const byte MCP2517_INT =  5 ; // INT output of MCP2517FD
...
ACAN2517 can (MCP2517_CS, mySPI, MCP2517_INT) ;
...
void setup () {
  ...
  mySPI.begin () ;
  pinPeripheral (MCP2517_SDI, PIO_SERCOM);
  pinPeripheral (MCP2517_SCK, PIO_SERCOM);
  pinPeripheral (MCP2517_SDO, PIO_SERCOM);
  ...
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
  ...
```

7.4 Connecting to an ESP32

Demo sketches: `LoopBackDemoESP32` and `LoopBackESP32-intensive`. See also the ESP32 demo sketch `SPI_Multiple_Busses`.

Link: https://randomnerdtutorials.com/esp32-pinout-reference-gpios/

Two ESP32 SPI busses are available in Arduino, `HSPI` and `VSPI`. By default, Arduino SPI is `VSPI`. The ESP32 default pins are given in table 2.

Port	SCK	MOSI	MISO
VSPI	IO18	IO23	IO19
HSPI	IO14	IO13	IO12

Table 2 – ESP32 SPI default pins

7.4.1 Connecting MCP2517_CS and MCP2517_INT

For `MCP2517_CS`, you can use any port that can be configured as digital output. `ACAN2517` does not support hardware chip select. For `MCP2517_INT`, you can use any port that can be configured as digital input, as ESP32 provides interrupt capability on any input pin.

Note. `IO34` to `IO39` are input only pins, without internal pullup or pulldown. So you cannot use theses pins for `MCP2517_CS`. If you use one of theses pins for `MCP2517_INT`, you should add an external pullup resistor if you configure the $\overline{\text{INT}}$ pin as Open Drain (section 18.7.2 page 38).

7.4.2 Using SPI

Default `SPI` (i.e. `VSPI`) pins are: SCK=18, MISO=19, MOSI=23 (figure 6).

You can change the default pins with additional arguments (up to three) for `SPI.begin` :

```
  SPI.begin (SCK_PIN) ; // Uses MISO and MOSI default pins
```

or

```
  SPI.begin (SCK_PIN, MISO_PIN) ; // Uses MOSI default pin
```

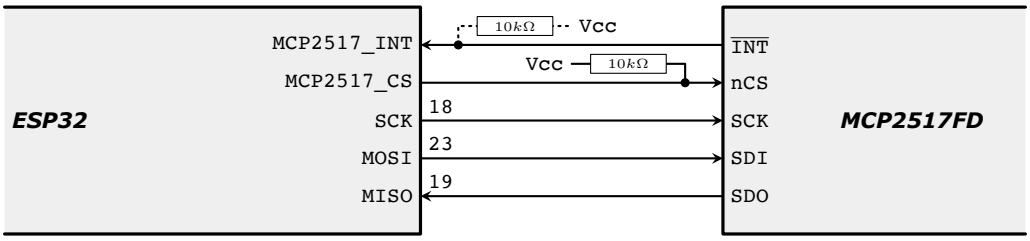

Figure 6 – Using VSPI default pins on an ESP32

or

```
SPI.begin (SCK_PIN, MISO_PIN, MOSI_PIN) ;
```

Note that `SPI.begin` accepts a fourth argument, for CS pin. Do not use this feature with `ACAN2517`.

7.4.3 Using HSPI

The ESP32 demo sketch `SPI_Multiple_Busses` shows how to use both `HSPI` and `VSPI`. However for `ACAN2517`, we proceed in a slightly different way:

```
#include <SPI.h>
....
SPIClass hspi (HSPI) ;
ACAN2517 can (MCP2517_CS, hspi, MCP2517_INT) ;
....
void setup () {
  ....
  hspi.begin () ; // You can also add parameters for not using default pins
  ....
}
```

You declare the `hspi` object before declaring the `can` object. You can change the `hspi` name, the important point is the `HSPI` argument that specifies the HSPI bus. Then, instead of using the `SPI` name, you use the `hspi` name in:

- `can` object declaration;
- in begin SPI instruction.

See the `LoopBackESP32-intensive` sketch for an example with `VSPI`.

7.5 Connection with no interrupt pin

See the `LoopBackDemoTeensy3xNoInt` and `LoopBackDemoESP32NoInt` sketches.

Note that not using an interruption is only valid if the message throughput is not too high. Received messages are recovered by polling, so the risk of MCP2517FD internal buffers overflowing is greater.

For not using the interrupt signal, you should adapt your sketch as following:

1. the last argument of `can` constructor should be `255`, meaning no interrupt pin;

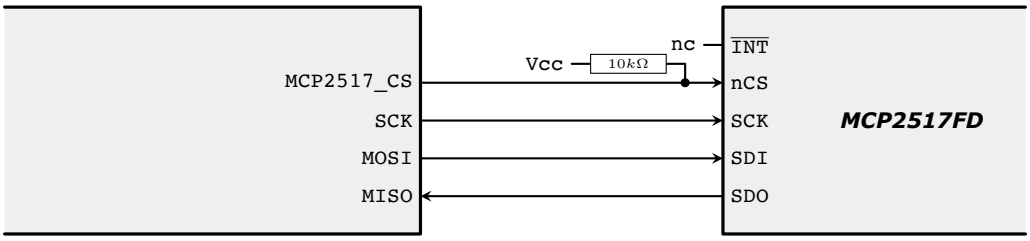

Figure 7 – Connection with no interrupt pin

2. the second argument of `can.begin` should be `NULL` (no interrupt service routine);
3. in the `loop` function, you should call `can.poll` as often as possible.

```
ACAN2517 can (MCP2517_CS, SPI, 255) ; // Last argument is 255 -> no interrupt pin

void setup () {
  ...
  const uint32_t errorCode = can.begin (settings, NULL) ; // ISR is null
  ...
}

void loop () {
  can.poll () ;
  ...
}
```

7.6 Wiring schemes

Here I list wiring schemes sent by users. If you want to see your wiring scheme here, send it to me. I will publish it in the next release of the library.

7.6.1 Arduino Uno - MCP2518FDClick

Thanks to `soso49` for this wiring scheme (figure 8).

8 Clock configuration

The `MCP251xFD` Oscillator Block Diagram is given in figure 9. Microchip recommends using a 4, 40 or 20 MHz CLKIN, Crystal or Ceramic Resonator. *A PLL can be enabled to multiply a 4 MHz clock by 10 by setting the PLLEN bit. Setting the SCLKDIV bit divides the SYSCLK by 2.*[6]

The `ACAN2517Settings` class defines an enumerated type for specifying your settings:

```
class ACAN2517Settings {
  public: typedef enum {
    OSC_4MHz,
    OSC_4MHz_DIVIDED_BY_2,
    OSC_4MHz10xPLL,
    OSC_4MHz10xPLL_DIVIDED_BY_2,
```

[6] DS20005678B, page 13.

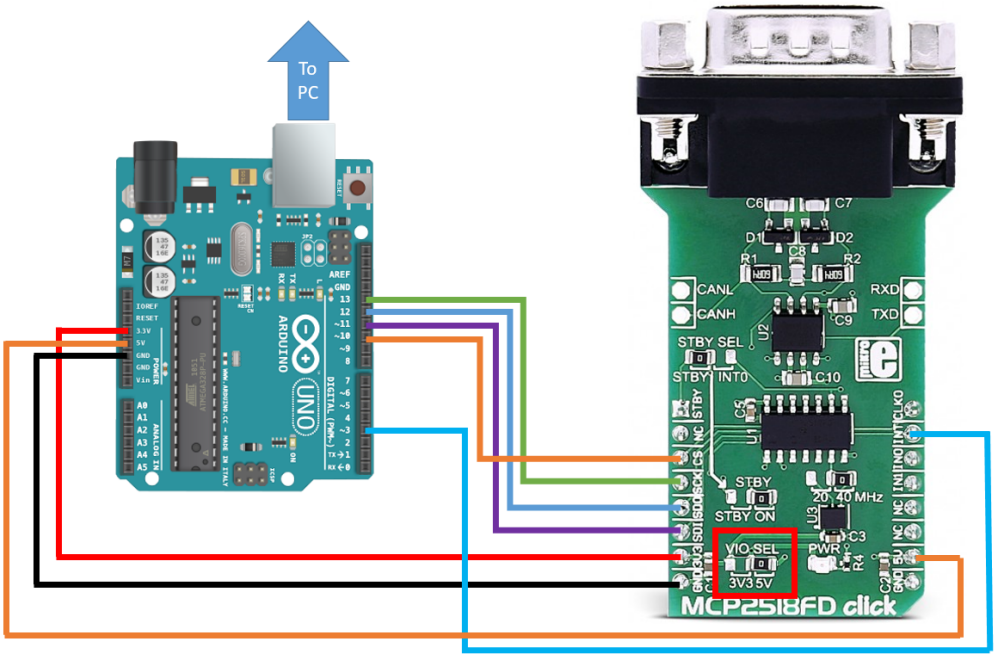

Figure 8 – Connecting an Arduino Uno with a MCP2518FDClick board

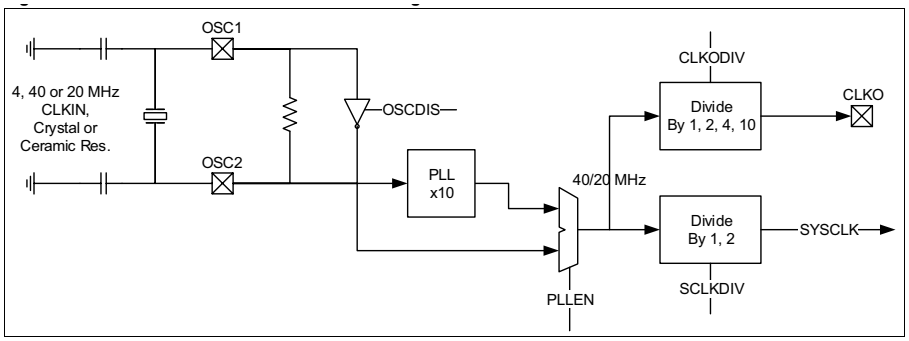

Figure 9 – `MCP251xFD` Oscillator Block Diagram (`DS20005678B`, figure 3.1 page 13)

```
    OSC_20MHz,
    OSC_20MHz_DIVIDED_BY_2,
    OSC_40MHz,
    OSC_40MHz_DIVIDED_BY_2
  } Oscillator ;
  ...
} ;
```

The first argument of the `ACAN2517Settings` constructor specifies the oscillator. For example, with a 4 MHz clock, the following settings lead to a 40 MHz `SYSCLK`, and a 1 MHz bit rate:

```
ACAN2517Settings settings2517 (ACAN2517Settings::OSC_4MHz10xPLL, 1000 * 1000) ;
```

The eight clock settings are given in the table 3. Note Microchip recommends a 40 MHz or 20 MHz `SYSCLK`. The `ACAN2517Settings` class has two accessors that return current settings: `oscillator()` and `sysClock()`.

The `begin` function of `ACAN2517` library first configures the selected SPI with a frequency of 1 Mbit/s,

Quartz	Oscillator parameter	SYSCLK
4 MHz	OSC_4MHz	4 MHz
4 MHz	OSC_4MHz_DIVIDE_BY_2	2 MHz
4 MHz	OSC_4MHz10xPLL	40 MHz
4 MHz	OSC_4MHz10xPLL_DIVIDE_BY_2	20 MHz
20 MHz	OSC_20MHz	20 MHz
20 MHz	OSC_20MHz_DIVIDE_BY_2	10 MHz
40 MHz	OSC_40MHz	40 MHz
40 MHz	OSC_40MHz_DIVIDE_BY_2	20 MHz

Table 3 – The ACAN2517 oscillator selection

for resetting the MCP2517FD and programming the PLLEN and SCLKDIV bits. Then SPI clock is set to a frequency equal to SYSCLK / 2, the maximum allowed frequency. More precisely, the SPI library of your microcontroller may adopt a lower frequency; for example, the maximum frequency of the Arduino Uno SPI is 8 Mbit/s.

Note that an incorrect setting may crash the MCP2517FD firmware (for example, enabling the PLL with a 20 MHz or 40 MHz quartz). In such case, no SPI communication can then be established, and in particular, the MCP2517FD cannot be reset by software. As the MCP2517FD has no RESET pin, the only way is to power off and power on the MCP2517FD.

9 Transmit FIFO

The transmit FIFO (see figure 1 page 8) is composed by:

- the *driver transmit FIFO*, whose size is positive or zero (default 16); you can change the default size by setting the `mDriverTransmitFIFOSize` property of your `settings` object;
- the *controller transmit FIFO*, whose size is between 1 and 32 (default 32); you can change the default size by setting the `mControllerTransmitFIFOSize` property of your `settings` object.

Having a *driver transmit FIFO* of zero size is valid; in this case, the FIFO must be considered both empty and full.

For sending a message throught the *Transmit FIFO*, call the `tryToSend` method with a message whose `idx` property is zero:

- if the *controller transmit FIFO* is not full, the message is appended to it, and `tryToSend` returns `true`;
- otherwise, if the *driver transmit FIFO* is not full, the message is appended to it, and `tryToSend` returns `true`; the interrupt service routine will transfer messages from *driver transmit FIFO* to the *controller transmit FIFO* when it becomes not full;
- otherwise, both FIFOs are full, the message is not stored and `tryToSend` returns `false`.

The transmit FIFO ensures sequentiality of emissions.

There are two other parameters you can override:

- `inSettings.mControllerTransmitFIFORetransmissionAttempts` is the number of retransmission attempts; by default, it is set to `UnlimitedNumber`; other values are `Disabled` and `ThreeAttempts`;

- inSettings.mControllerTransmitFIFOPriority is the priority of the transmit FIFO: between 0 (lowest priority) and 31 (highest priority); default value is 0.

The *controller transmit FIFO* is located in the MCP2517FD RAM. It requires 16 bytes for each message (see section 12 page 21).

9.1 The driverTransmitBufferSize method

The driverTransmitBufferSize method returns the allocated size of this driver transmit buffer, that is the value of settings.mDriverTransmitBufferSize when the begin method is called.

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

9.2 The driverTransmitBufferCount method

The driverTransmitBufferCount method returns the current number of messages in the driver transmit buffer.

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

9.3 The driverTransmitBufferPeakCount method

The driverTransmitBufferPeakCount method returns the peak value of message count in the driver transmit buffer

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

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 driverTransmitBufferPeakCount() will return driverTransmitBufferSize ()+1.

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

10 Transmit Queue (TXQ)

The *Transmit Queue* is handled by the MCP2517FD, its contents is located in its RAM. **It is not a FIFO.** *Messages inside the TXQ will be transmitted based on their ID. The message with the highest priority ID, lowest ID value will be transmitted first*[7].

By default, the *transmit queue* is disabled (its default size is 0); you can change the default size by setting the mControllerTXQSize property of your settings object. The maximum valid size is 32.

For sending a message throught the *transmit queue*, call the tryToSend method with a message whose idx property is 255:

- if the *transmit queue* size is not zero and if it is not full, the message is appended to it, and tryToSend returns true;

[7] DS20005678B, section 4.5, page 28.

- otherwise, the message is not stored and `tryToSend` returns `false`.

There are two other parameters you can override:

- `inSettings.mControllerTXQBufferRetransmissionAttempts` is the number of retransmission attempts; by default, it is set to `UnlimitedNumber`; other values are `Disabled` and `ThreeAttempts`;
- `inSettings.mControllerTXQBufferPriority` is the priority of the TXQ buffer: between 0 (lowest priority) and 31 (highest priority); default value is 31.

The *transmit queue* is located in the `MCP2517FD` RAM. It requires 16 bytes for each message (see section 12 page 21).

11 Receive FIFO

The receive FIFO (see figure 1 page 8) is composed by:

- the *driver receive FIFO*, whose size is positive (default 32); you can change the default size by setting the `mDriverReceiveFIFOSize` property of your `settings` object;
- the *controller receive FIFO*, whose size is between 1 and 32 (default 32); you can change the default size by setting the `mControllerReceiveFIFOSize` property of your `settings` object.

When an incoming message is accepted by a receive filter:

- if the *controller receive FIFO* is full, the message is lost;
- otherwise, it is stored in the *controller receive FIFO*.

Then, if the *driver receive FIFO* is not full, the message is transferred by the *interrupt service routine* from *controller receive FIFO* to the *driver receive FIFO*. So the *driver receive FIFO* never overflows, but *controller receive FIFO* may.

The `ACAN2517::available`, `ACAN2517::receive` and `ACAN2517::dispatchReceivedMessage` methods work only with the *driver receive FIFO*. As soon as it becomes not full, messages from *controller receive FIFO* are transferred by the *interrupt service routine*.

The receive FIFO ensures sequentiality of reception.

The *controller receive FIFO* is located in the `MCP2517FD` RAM. It requires 16 bytes for each message (see next section).

12 RAM usage

The `MCP2517FD` contains a 2048 bytes RAM that is used to store message objects[8]. There are three different kinds of message objects:

- Transmit Message Objects used by the TXQ buffer;
- Transmit Message Objects used by the transmit FIFO;

[8] DS20005688B, section 3.3, page 63.

- Receive Message Objects used by the receive FIFO.

Every message object is 16 bytes[9], so you can use up to 128 message objects.

By default, the transmit FIFO is 32 message deep (512 bytes), the TXQ buffer is disabled (0 byte), and the receive FIFO is 32 message deep (512 bytes), given a total amount of 1024 bytes.

The `ACAN2517Settings::ramUsage` method computes the required memory amount:

```
uint32_t ACAN2517Settings::ramUsage (void) const {
  uint32_t result = 0 ;
//--- TXQ
  result += 16 * mControllerTXQSize ;
//--- Receive FIFO (FIFO #1)
  result += 16 * mControllerReceiveFIFOSize ;
//--- Send FIFO (FIFO #2)
  result += 16 * mControllerTransmitFIFOSize ;
//---
  return result ;
}
```

The `ACAN2517:begin` method checks the required memory amount is lower or equal than 2048 bytes. Otherwise, it raises the error code `kControllerRamUsageGreaterThan2048`.

You can also use the *MCP2517FD RAM Usage Calculations* Excel sheet from Microchip[10].

13 Sending frames: 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 for the transmit FIFO (section 9 page 19) ;
- 255 for the transmit Queue (section 10 page 20).

The method `tryToSend` returns:

- `true` if the message has been successfully transmitted to the 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 the 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:

[9] 16 bytes because the MCP2517FD is in the CAN 2.0B mode, otherwise a CANFD message object can require up to 72 bytes.
[10] http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD%20RAM%20Usage%20Calculations%20-%20UG.xlsx

```
static uint32_t 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 ;
    }
  }
  ...
}
```

14 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 16 page 28).

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.

14.1 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:

```
ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL, 125 * 1000) ;
settings.mReceiveBufferSize = 100 ;
const uint32_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`.

14.2 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 () ;
```

14.3 The `receiveBufferCount` method

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

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

14.4 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.

15 Acceptance filters

Note. The acceptance filters ACAN2517FD library, that handles a MCP2517FD CAN Controller in the CANFD mode[11], are almost identical, they differ only from the prototype of the callback routine.

If you invoke the ACAN2517.begin method with two arguments, it configures the MCP2517FD for receiving all messages.

```
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
```

If you want to define receive filters, you have to set up an ACAN2517Filters instance object, and pass it as third argument of the ACAN2517.begin method:

```
  ACAN2517Filters filters ;
  ... // Append filters
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }, filters) ;
  ...
```

15.1 An example

Sample sketch: the LoopBackDemoTeensy3xWithFilters sketch is an example of filter definition.

```
  ACAN2517Filters filters ;
```

First, you instanciate an ACAN2517Filters object. It represents an empty list of filters. So, if you do not append any filter, can.begin (settings, [] { can.isr () ; }, filters) configures the controller in such a way that no messages can be received.

```
// Filter #0: receive standard frame with identifier 0x123
  filters.appendFrameFilter (kStandard, 0x123, receiveFromFilter0) ;
// Filter #1: receive extended frame with identifier 0x12345678
  filters.appendFrameFilter (kExtended, 0x12345678, receiveFromFilter1) ;
```

You define the filters sequentially, with the four methods: appendPassAllFilter, appendFormatFilter, appendFrameFilter, appendFilter. Theses methods have as last argument an optional callback routine, that is called by the dispatchReceivedMessage method (see section 16 page 28).

The appendFrameFilter defines a filter that matches for an extended or standard identifier of a given value.

You can define up to 32 filters. Filter definition registers are outside the MCP2517FD RAM, so defining filter does not restrict the receive and transmit buffer sizes. Note that MCP2517FD filter does not allow to establish a filter based on the data / remote information.

[11] https://github.com/pierremolinaro/acan2517FD

```
// Filter #2: receive standard frame with identifier 0x3n4 (0 <= n <= 15)
  filters.appendFilter (kStandard, 0x70F, 0x304, receiveFromFilter2) ;
```

The `appendFilter` defines a filter that matches for an identifier that matches the condition:

$$\texttt{identifier} \ \& \ \texttt{0x70F} == \texttt{0x304}$$

The `kStandard` argument constraints to accept only standard frames. So the accepted standard identifiers are `0x304`, `0x314`, `0x324`, ..., `0x3E4`, `0x3F4`.

```
//----------------------------------- Filters ok ?
  if (filters.filterStatus () != ACAN2517Filters::kFiltersOk) {
    Serial.print ("Error␣filter␣") ;
    Serial.print (filters.filterErrorIndex ()) ;
    Serial.print (":␣") ;
    Serial.println (filters.filterStatus ()) ;
  }
```

Filter definitions can have error(s), you can check error kind with the `filterStatus` method. If it returns a value different than `ACAN2517Filters::kFiltersOk`, there is at least one error: only the last one is reported, and the `filterErrorIndex` returns the corresponding filter index. Note this does not check the number of filters is lower or equal than 32.

```
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }, filters) ;
```

The `begin` method checks the filter definition:

- it raises the `kMoreThan32Filters` error if more than 32 filters are defined;
- it raises the `kFilterDefinitionError` error if one or more filter definitions are erroneous (that is if `filterStatus` returns a value different than `ACAN2517Filters::kFiltersOk`).

15.2 The `appendPassAllFilter` method

```
void ACAN2517Filters::appendPassAllFilter (const ACANCallBackRoutine inCallBackRoutine) ;
```

This defines a filter that accepts all (standard / extended, remote / data) frames.

If used, this filter must be the last one: as the `MCP2517FD` tests the filters sequentially, the following filters will never match.

15.3 The `appendFormatFilter` method

```
void ACAN2517Filters::appendFormatFilter (const tFrameFormat inFormat,
                                          const ACANCallBackRoutine inCallBackRoutine) ;
```

This defines a filter that accepts:

- if `inFormat` is equal to `kStandard`, all standard remote frames and all standard data frames;
- if `inFormat` is equal to `kExtended`, all extended remote frames and all extended data frames.

15.4 The `appendFrameFilter` method

```
void ACAN2517Filters::appendFrameFilter (const tFrameFormat inFormat,
                                         const uint32_t inIdentifier,
                                         const ACANCallBackRoutine inCallBackRoutine) ;
```

This defines a filter that accepts:

- if `inFormat` is equal to `kStandard`, all standard remote frames and all standard data frames with a given identifier;
- if `inFormat` is equal to `kExtended`, all extended remote frames and all extended data frames with a given identifier.

If `inFormat` is equal to `kStandard`, the `inIdentifier` should be lower or equal to `0x7FF`. Otherwise, `settings.filterStatus ()` returns the `kStandardIdentifierTooLarge` error.

If `inFormat` is equal to `kExtended`, the `inIdentifier` should be lower or equal to `0x1FFFFFFF`. Otherwise, `settings.filterStatus ()` returns the `kExtendedIdentifierTooLarge` error.

15.5 The `appendFilter` method

```
void ACAN2517Filters::appendFilter (const tFrameFormat inFormat,
                                    const uint32_t inMask,
                                    const uint32_t inAcceptance,
                                    const ACANCallBackRoutine inCallBackRoutine) ;
```

The `inMask` and `inAcceptance` arguments defines a filter that accepts frame whose identifier verifies:

$$\texttt{identifier} \;\&\; \texttt{inMask} == \texttt{inAcceptance}$$

The `inFormat` filters standard (if `inFormat` is equal to `kStandard`) frames, or extended ones (if `inFormat` is equal to `kExtended`).

Note that `inMask` and `inAcceptance` arguments should verify:

$$\texttt{inAcceptance} \;\&\; \texttt{inMask} == \texttt{inAcceptance}$$

Otherwise, `settings.filterStatus ()` returns the `kInconsistencyBetweenMaskAndAcceptance` error.

If `inFormat` is equal to `kStandard`:

- the `inAcceptance` should be lower or equal to `0x7FF`; Otherwise, `settings.filterStatus ()` returns the `kStandardAcceptanceTooLarge` error;
- the `inMask` should be lower or equal to `0x7FF`; Otherwise, `settings.filterStatus ()` returns the `kStandardMaskTooLarge` error.

If `inFormat` is equal to `kExtended`:

- the `inAcceptance` should be lower or equal to `0x1FFFFFFF`; Otherwise, `settings.filterStatus ()` returns the `kExtendedAcceptanceTooLarge` error;

- the `inMask` should be lower or equal to `0x1FFFFFFF`; Otherwise, `settings.filterStatus ()` returns the `kExtendedMaskTooLarge` error.

16 The `dispatchReceivedMessage` method

Sample sketch: the `LoopBackDemoTeensy3xWithFilters` 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.

17 The `ACAN2517::begin` method reference

17.1 The prototypes

```
uint32_t ACAN2517::begin (const ACAN2517Settings & inSettings,
                          void (* inInterruptServiceRoutine) (void)) ;
```

This prototype has two arguments, a `ACAN2517Settings` instance that defines the settings, and the interrupt service routine, that can be specified by a lambda expression or a function (see section 17.2 page 29). It configures the controller in such a way that all messages are received (*pass-all* filter).

```
uint32_t ACAN2517::begin (const ACAN2517Settings & inSettings,
                          void (* inInterruptServiceRoutine) (void),
                          const ACAN2517Filters & inFilters) ;
```

The second prototype has a third argument, an instance of `ACAN2517Filters` class that defines the receive filters.

17.2 Defining explicitly the interrupt service routine

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

```
  const uint32_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 uint32_t errorCode = can.begin (settings, canISR) ;
```

17.3 The error code

The `ACAN2517::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 4. An error code could report several errors. The `ACAN2517` class defines static constants for naming errors.

17.3.1 kRequestedConfigurationModeTimeOut

The `ACAN2517::begin` method first configures SPI with a 1 Mbit/s clock, and then requests the configuration mode. This error is raised when the `MCP2517FD` does not reach the configuration mode in 2ms. It means that the `MCP2517FD` cannot be accessed via SPI.

17.3.2 kReadBackErrorWith1MHzSPIClock

Then, the `ACAN2517::begin` method checks accessibility by writing and reading back 32-bit values at the first `MCP2517FD` RAM address (`0x400`). The values are $1 << n$, with $0 \leqslant n \leqslant 31$. This error is raised when the read value is different from the written one. It means that the `MCP2517FD` cannot be accessed via SPI.

Bit	Static constant Name	Link
0	kRequestedConfigurationModeTimeOut	section 17.3.1 page 29
1	kReadBackErrorWith1MHzSPIClock	section 17.3.2 page 29
2	kTooFarFromDesiredBitRate	section 17.3.3 page 30
3	kInconsistentBitRateSettings	section 17.3.4 page 30
4	kINTPinIsNotAnInterrupt	section 17.3.5 page 30
5	kISRIsNull	section 17.3.6 page 30
6	kFilterDefinitionError	section 17.3.7 page 31
7	kMoreThan32Filters	section 17.3.8 page 31
8	kControllerReceiveFIFOSizeIsZero	section 17.3.9 page 31
9	kControllerReceiveFIFOSizeGreaterThan32	section 17.3.10 page 31
10	kControllerTransmitFIFOSizeIsZero	section 17.3.11 page 31
11	kControllerTransmitFIFOSizeGreaterThan32	section 17.3.12 page 31
12	kControllerRamUsageGreaterThan2048	section 17.3.13 page 31
13	kControllerTXQPriorityGreaterThan31	section 17.3.14 page 31
14	kControllerTransmitFIFOPriorityGreaterThan31	section 17.3.15 page 31
15	kControllerTXQSizeGreaterThan32	section 17.3.16 page 32
16	kRequestedModeTimeOut	section 17.3.17 page 32
17	kX10PLLNotReadyWithin1MS	section 17.3.18 page 32
18	kReadBackErrorWithFullSpeedSPIClock	section 17.3.19 page 32
19	kISRNotNullAndNoIntPin	section 17.3.20 page 32

Table 4 – The ACAN2517::begin method error code bits

17.3.3 kTooFarFromDesiredBitRate

This error occurs when the mBitRateClosedToDesiredRate property of the settings object is false. This means that the ACAN2517Settings constructor cannot compute a CAN bit configuration close enough to the desired bit rate. For example:

```
void setup () {
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL, 1) ; // 1 bit/s !!!
  // Here, settings.mBitRateClosedToDesiredRate is false
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
  // Here, errorCode contains ACAN2517::kCANBitConfigurationTooFarFromDesiredBitRate
}
```

17.3.4 kInconsistentBitRateSettings

The ACAN2517Settings constructor always 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 18.2 page 36.

17.3.5 kINTPinIsNotAnInterrupt

The pin you provide for handling the MCP2517FD interrupt has no interrupt capability.

17.3.6 kISRIsNull

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

17.3.7 kFilterDefinitionError

`settings.filterStatus()` returns a value different than `ACAN2517Filters::kFiltersOk`, meaning that one or more filters are erroneous. See section 15.1 page 25.

17.3.8 kMoreThan32Filters

You have defined more than 32 filters. `MCP2517FD` cannot handle more than 32 filters.

17.3.9 kControllerReceiveFIFOSizeIsZero

You have assigned 0 to `settings.mControllerReceiveFIFOSize`. The *controller receive FIFO size* should be greater than 0.

17.3.10 kControllerReceiveFIFOSizeGreaterThan32

You have assigned a value greater than 32 to `settings.mControllerReceiveFIFOSize`. The *controller receive FIFO size* should be lower or equal than 32.

17.3.11 kControllerTransmitFIFOSizeIsZero

You have assigned 0 to `settings.mControllerTransmitFIFOSize`. The *controller transmit FIFO size* should be greater than 0.

17.3.12 kControllerTransmitFIFOSizeGreaterThan32

You have assigned a value greater than 32 to `settings.mControllerTransmitFIFOSize`. The *controller transmit FIFO size* should be lower or equal than 32.

17.3.13 kControllerRamUsageGreaterThan2048

The configuration you have defined requires more than 2048 bytes of `MCP2517FD` internal RAM. See section 12 page 21.

17.3.14 kControllerTXQPriorityGreaterThan31

You have assigned a value greater than 31 to `settings.mControllerTXQBufferPriority`. The *controller transmit FIFO size* should be lower or equal than 31.

17.3.15 kControllerTransmitFIFOPriorityGreaterThan31

You have assigned a value greater than 31 to `settings.mControllerTransmitFIFOPriority`. The *controller transmit FIFO size* should be lower or equal than 31.

17.3.16 kControllerTXQSizeGreaterThan32

You have assigned a value greater than 32 to `settings.mControllerTXQSize`. The *controller transmit FIFO size* should be lower than 32.

17.3.17 kRequestedModeTimeOut

During configuration by the `ACAN2517::begin` method, the `MCP2517FD` 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.

17.3.18 kX10PLLNotReadyWithin1MS

You have requested the `QUARTZ_4MHz10xPLL` oscillator mode, enabling the 10x PLL. The `ACAN2517::begin` method waits during 2ms the PLL to be locked. This error is raised when the PLL is not locked within 2 ms.

17.3.19 kReadBackErrorWithFullSpeedSPIClock

After the oscillator configuration has been established, the `ACAN2517::begin` method configures the SPI at its full speed (`SYSCLK/2`, and checks accessibility by writing and reading back 32-bit values at the first `MCP2517FD` RAM address (`0x400`). The values are $1 << n$, with $0 \leqslant n \leqslant 31$. This error is raised when the read value is different from the written one.

17.3.20 kISRNotNullAndNoIntPin

This error occurs when you have no `INT` pin, and a not-null interrupt service routine:

```
ACAN2517 can (MCP2517_CS, SPI, 255) ; // Last argument is 255 -> no interrupt pin

void setup () {
  ...
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ; // ISR is not null
  ...
}
```

Interrupt service routine should be `NULL` if no `INT` pin is defined:

```
ACAN2517 can (MCP2517_CS, SPI, 255) ; // Last argument is 255 -> no interrupt pin

void setup () {
  ...
  const uint32_t errorCode = can.begin (settings, NULL) ; // Ok, ISR is null
  ...
}
```

See the `LoopBackDemoTeensy3xNoInt` and `LoopBackDemoESP32NoInt` sketches.

18 ACAN2517Settings class reference

Note. The ACAN2517Settings class is not Arduino specific. You can compile it on your desktop computer with your favorite C++ compiler. In the https://github.com/pierremolinaro/ACAN2517-dev GitHub repository, a command line tool is defined for exploring all CAN bit rates from 1 bit/s and 20 Mbit/s. It also checks that computed CAN bit decompositions are all consistent, even if they are too far from the desired baud rate.

18.1 The ACAN2517Settings constructor: computation of the CAN bit settings

The constructor of the ACAN2517Settings 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:

```
void setup () {
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             1 * 1000 * 1000) ; // 1 Mbit/s
  // Here, settings.mBitRateClosedToDesiredRate is true
  ...
}
```

Of course, with a 40 MHz or 20 MHz SYSCLK, CAN bit computation always 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:

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             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 $1\ \text{ppm} = 10^{-6}$. In other words, $10,000\ \text{ppm} = 1\%$.

By default, a desired bit rate is accepted if the distance from the computed actual bit rate is lower or equal to $1,000\ \text{ppm} = 0.1\ \%$. You can change this default value by adding your own value as third argument of ACAN2517Settings constructor:

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             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:

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             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 always gives a consistent result, resulting an actual bit rate closest from the desired bit rate. For example:

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             423 * 1000) ; // 423 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  421052 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 4603 ppm
  ...
}
```

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

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             423 * 1000) ; // 423 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  421052 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 4603 ppm
  Serial.print ("Bit rate prescaler: ") ;
  Serial.println (settings.mBitRatePrescaler) ; // BRP = 1
  Serial.print ("Phase segment 1: ") ;
  Serial.println (settings.mPhaseSegment1) ; // PS1 = 75
  Serial.print ("Phase segment 2: ") ;
  Serial.println (settings.mPhaseSegment2) ; // PS2 = 19
  Serial.print ("Resynchronization Jump Width: ") ;
  Serial.println (settings.mSJW) ; // SJW = 19
  Serial.print ("Triple Sampling: ") ;
  Serial.println (settings.mTripleSampling) ; // 0, meaning single sampling
```

```
  Serial.print ("Sample␣Point:␣") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 80, meaning 80%
  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 always 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.

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  settings.mPhaseSegment1 -= 8 ; // 63 -> 55: safe, 1 <= PS1 <= 256
  settings.mPhaseSegment2 += 8 ; // 16 -> 24: safe, 1 <= PS2 <= 128
  settings.mSJW += 8           ; // 16 -> 24: safe, 1 <= SJW <= PS2
  Serial.print ("Sample␣Point:␣") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 68, meaning 68%
  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 always respect CAN bit timing consistency! The `MCP2517FD` constraints are:

$$1 \leqslant \texttt{mBitRatePrescaler} \leqslant 256$$
$$2 \leqslant \texttt{mPhaseSegment1} \leqslant 256$$
$$1 \leqslant \texttt{mPhaseSegment2} \leqslant 128$$
$$1 \leqslant \texttt{mSJW} \leqslant \texttt{mPhaseSegment2}$$

Resulting actual bit rate is given by:

$$\text{Actual bit rate} = \frac{\texttt{SYSCLK}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2})}$$

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

$$\text{Sampling point} = 100 \cdot \frac{1 + \texttt{mPhaseSegment1}}{1 + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2}}$$

18.2 The CANBitSettingConsistency method

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

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             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 5.

The ACAN2517Settings 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	kBitRatePrescalerIsGreaterThan256	mBitRatePrescaler > 256
2	kPhaseSegment1IsLowerThan2	mPhaseSegment1 < 2
3	kPhaseSegment1IsGreaterThan256	mPhaseSegment1 > 256
4	kPhaseSegment2IsZero	mPhaseSegment2 == 0
5	kPhaseSegment2IsGreaterThan128	mPhaseSegment2 > 128
6	kSJWIsZero	mSJW == 0
7	kSJWIsGreaterThan128	mSJW > 128
8	kSJWIsGreaterThanPhaseSegment1	mSJW > mPhaseSegment1
9	kSJWIsGreaterThanPhaseSegment2	mSJW > mPhaseSegment2

Table 5 – The ACAN2517Settings::CANBitSettingConsistency method error codes

18.3 The actualBitRate method

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

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             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 18.2 page 36), the returned value is irrelevant.

18.4 The exactBitRate method

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

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             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 18.2 page 36), the returned value is irrelevant.

With a 40 MHz SYSCLK, the 46 exact bit rates are : 500 bit/s, 625 bit/s, 640 bit/s, 800 bit/s, 1 kbit/s, 1250 bit/s, 1280 bit/s, 1600 bit/s, 2 kbit/s, 2500 bit/s, 2560 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, 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, 78125 bit/s, 80 kbit/s, 100 kbit/s, 125 kbit/s, 156250 bit/s, 160 kbit/s, 200 kbit/s, 250 kbit/s, 312500 bit/s, 320 kbit/s, 400 kbit/s, 500 kbit/s, 625 kbit/s, 800 kbit/s, 1 Mbit/s.

18.5 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): $1\ \text{ppm} = 10^{-6}$. In other words, $10,000\ \text{ppm} = 1\%$.

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             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 18.2 page 36), the returned value is irrelevant.

18.6 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 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%. The bit rate calculator tries to set the sample point at 80%.

```
void setup () {
  ...
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL,
                             500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("Sample␣point:␣") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 80 --> 80%
  ...
}
```

Note. If CAN bit settings are not consistent (see section 18.2 page 36), the returned value is irrelevant.

18.7 Properties of the ACAN2517Settings class

All properties of the ACAN2517Settings class are declared public and are initialized (table 6). 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.

18.7.1 The mTXCANIsOpenDrain property

This property defines the outpiut mode of the TXCAN pin:

- if false (default value), the TXCAN pin is a push/pull output;
- if true, the TXCAN pin is an open drain output.

18.7.2 The mINTIsOpenDrain property

This property defines the outpiut mode of the MCP2517FD INT pin:

- if false (default value), the INT pin is a push/pull output;
- if true, the INT pin is an open drain output.

18.7.3 The CLKO/SOF pin

The CLKO/SOF pin of the MCP2517FD controller is an output pin has five functions[12]:

- output *internally generated clock*;

[12] *Internally generated clock* is not SYSCLK, see figure 9 page 18.

Property	Type	Initial value	Comment
mOscillator	Oscillator	Constructor argument	
mSysClock	uint32_t	Constructor argument	
mDesiredBitRate	uint32_t	Constructor argument	
mBitRatePrescaler	uint16_t	0	See section 18.1 page 33
mPhaseSegment1	uint16_t	0	See section 18.1 page 33
mPhaseSegment2	uint8_t	0	See section 18.1 page 33
mSJW	uint8_t	0	See section 18.1 page 33
mBitRateClosedToDesiredRate	bool	false	See section 18.1 page 33
mTXCANIsOpenDrain	bool	false	See section 18.7.1 page 38
mINTIsOpenDrain	bool	false	See section 18.7.2 page 38
mCLKOPin	CLKOpin	CLKO_DIVIDED_BY_10	See section 18.7.3 page 38
mRequestedMode	RequestedMode	Normal20B	See section 18.7.4 page 40
mDriverTransmitFIFOSize	uint16_t	16	See section 9 page 19
mControllerTransmitFIFOSize	uint8_t	32	See section 9 page 19
mControllerTransmitFIFOPriority	uint8_t	0	See section 9 page 19
mControllerTransmitFIFO-RetransmissionAttempts	RetransmissionAttempts	UnlimitedNumber	See section 9 page 19
mControllerTXQSize	uint8_t	0	See section 10 page 20
mControllerTXQBufferPriority	uint8_t	31	See section 10 page 20
mControllerTXQBuffer-RetransmissionAttempts	RetransmissionAttempts	UnlimitedNumber	See section 10 page 20
mDriverReceiveFIFOSize	uint16_t	32	See section 11 page 21
mControllerReceiveFIFOSize	uint8_t	32	See section 11 page 21

Table 6 – Properties of the ACAN2517Settings class

- output *internally generated clock* divided by 2;
- output *internally generated clock* divided by 4;
- output *internally generated clock* divided by 10;
- output SOF ("Start Of Frame").

By default, after power on, CLKO/SOF pin outputs *internally generated clock* divided by 10.

The ACAN2517Settings class defines an enumerated type for specifying these settings:

```
class ACAN2517Settings {
  public: typedef enum {CLKO_DIVIDED_BY_1, CLKO_DIVIDED_BY_2,
                        CLKO_DIVIDED_BY_4, CLKO_DIVIDED_BY_10,
                        SOF} CLKOpin ;
  ...
} ;
```

The mCLKOPin property lets you select the CLKO/SOF pin function; by default, this property value is CLKO_DIVIDED_BY_10, that corresponds to MCP2517FD power on setting. For example:

```
  ACAN2517Settings settings (ACAN2517Settings::OSC_4MHz10xPLL, CAN_BIT_RATE) ;
  ...
  settings.mCLKOPin = ACAN2517Settings::SOF ;
  ...
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
```

18.7.4 The mRequestedMode property

This property defines the mode requested at this end of the configuration: Normal20B (default value), InternalLoopBack, ExternalLoopBack, ListenOnly.

19 Handling GPIO0, GPIO1 and XSTBY

The #8 pin is never used as $\overline{\text{INT1}}$ by the library. By default, it is an input pin. You can use it as a input or ouput digital pin. The #9 pin is never used as $\overline{\text{INT0}}$ by the library. By default, it is an input pin. You can use it as a input or ouput digital pin, or as XSTBY output pin.

19.1 The gpioSetMode method

```
void ACAN2517FD::gpioSetMode (const uint8_t inPin, const uint8_t inMode) ;
```

This method sets the mode of GPIO0 or GPIO1.

Following the inPin value:

- 0: following the inMode value:
  - INPUT: pin #9 is no more XSTBY output, configure pin #9 (GPIO0) as digital input;
  - OUTPUT: pin #9 is no more XSTBY output, configure pin #9 (GPIO0) as digital output;
  - *other value*: does nothing;
- 1: following the inMode value:
  - INPUT: configure pin #8 (GPIO1) as digital input;
  - OUTPUT: configure pin #8 (GPIO1) as digital output;
  - *other value*: does nothing;
- *other value*: does nothing.

Example :

```
can.gpioSetMode (1, OUTPUT) ; // Configures GPIO1 as digital output
```

19.2 The gpioWrite method

```
void ACAN2517FD::gpioWrite (const uint8_t inPin, const uint8_t inLevel) ;
```

This method outputs a logic value on GPIO0 or GPIO1.

Following the inPin value:

- 0: following the inLevel value:
  - 0 or LOW: output a low level on GPIO0;
  - HIGH or any value > 0: output a high level on GPIO0;

- 1: following the inLevel value:
  - 0 or LOW: output a low level on GPIO1;
  - HIGH or any value > 0: output a high level on GPIO1;
- *other value*: does nothing.

Example :

```
can.gpioWrite (1, HIGH) ; // If GPIO1 is a digital output, outputs a high level
```

19.3 The gpioRead method

```
bool ACAN2517FD::gpioRead (const uint8_t inPin) ;
```

This method gets the logic value of GPIO0 or GPIO1.

Following the inPin value:

- 0: the function returns the level of GPIO0 pin;
- 1: the function returns the level of GPIO1 pin;
- *other value*: returns false.

Example :

```
const bool b = can.gpioRead (1) ; // Get GPIO1 logical level
```

19.4 The configureGPIO0AsXSTBY method

```
void ACAN2517FD::configureGPIO0AsXSTBY (void) ;
```

This method configures the #9 pin as XSTBY, overriding any previous pin mode.

20 Other ACAN2517FD methods

20.1 The currentOperationMode method

```
ACAN2517FD::OperationMode ACAN2517FD::currentOperationMode (void) ;
```

This function returns the MCP2517FD current operation mode, as a value of the ACAN2517FD::currentOperationMode enumerated type. This type is defined in the ACAN2517FD.h header file.

```
class ACAN2517FD {
  ...
  public: typedef enum : uint8_t {
    NormalFD = 0,
    Sleep = 1,
    InternalLoopBack = 2,
    ListenOnly = 3,
    Configuration = 4,
```

```
    ExternalLoopBack = 5,
    Normal20B = 6,
    RestrictedOperation = 7
  } OperationMode ;
  ...
} ;
```

20.2 The recoverFromRestrictedOperationMode method

```
bool ACAN2517FD::recoverFromRestrictedOperationMode (void) ;
```

If the MCP2517FD is in *Restricted Operation Mode*, this method requests the operation mode defined by the mRequestedMode property of the ACAN2517FDSettings class instance. This method has no effect is the current mode is not the *Restricted Operation Mode*.

This method returns true if both conditions are met:

- the MCP2517FD is in *Restricted Operation Mode*;
- the operation mode has been successfully recovered.

It returns false otherwise.

20.3 The errorCounters method

```
uint32_t ACAN2517FD::errorCounters (void) ;
```

This method returns the transmit / receive error count register value, as described in DS20005688B, REGISTER 3-19 page 41. The CiTREC value is zero when there is no error.

20.4 The diagInfos method

```
uint32_t ACAN2517FD::diagInfos (const int inIndex = 1) ;
```

Thanks to Flole998 and turmary. This method returns:

- if inIndex is equal to 0, the C1BDIAG0 register value, as described in DS20005688B, REGISTER 3-20 page 42;
- if inIndex is not equal to 0, the C1BDIAG1 register value, as described in DS20005688B, REGISTER 3-21 page 43.