ACAN2517FD Arduino library
For `MCP2517FD`, in CAN FD mode
Version 1.0.3

Pierre Molinaro

January 6, 2019

Contents

1 Versions

Version	**Date**	**Comment**
1.0.0	October 28, 2018	Initial release
1.0.1	October 29, 2018	Conformity with Arduino library
1.0.2	November 2, 2018	added `mISOCRCEnabled` setting
1.0.3	January 6, 2019	Corrected identifiers for extended messages.

2 Features

The `ACAN2517FD` library is a MCP2517FD CAN ("Controller Area Network") Controller driver for any board running Arduino. It handles CAN FD frames.

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 ACAN2517 library (`https://github.com/pierremolinaro/acan2517`), CAN driver for `MCP2517FD` CAN controller, in CAN 2.0B 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;
- ISO CRC enabled by default;
- efficient built-in CAN bit settings computation from arbitration and data bit rates;
- 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 Data flow

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

3.1 Data flow in default configuration

The figure 1 illustrates message flow in the default configuration.

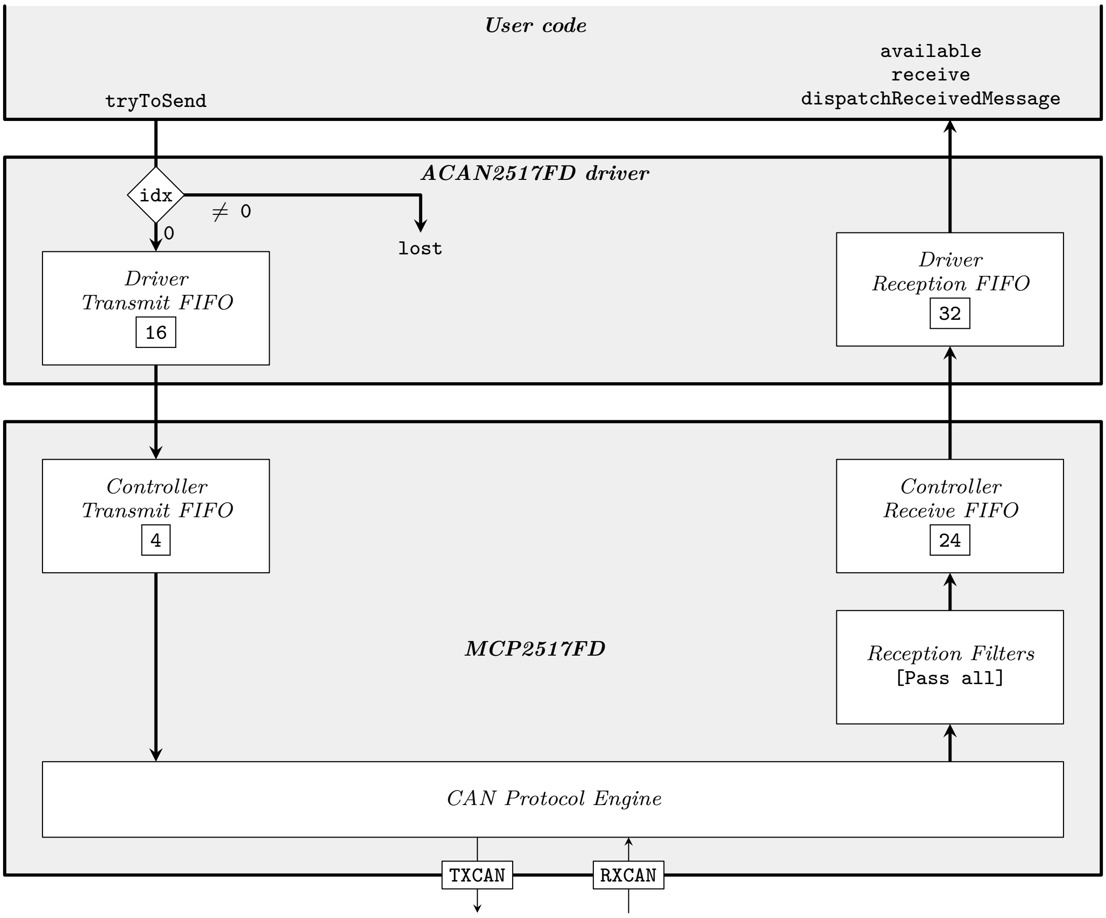

Figure 1 – Message flow in `ACAN2517FD` driver and `MCP2517FD` CAN Controller, default configuration

Sending messages. The ACAN2517FD driver defines a *driver transmit FIFO* (default size: 16 messages), and configures the `MCP2517FD` with a *controller transmit FIFO* with a size of 4 messages. `MCP2517FD` RAM has a capacity of 2048 bytes, that limits the sizes of the *controller transmit FIFO* and *controller receive FIFO*. See section 13 page 17.

A message is defined by an instance of `CANFDMessage` class. For sending a message, user code calls the `tryToSend` method – see section 14 page 18, 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 16 page 21 for configuring them. Messages that pass the filters are stored in the *Controller Reception FIFO*; its size is 24 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 15.1 page 20 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 15 page 20;
- the `dispatchReceivedMessage` method if you have defined the reception filters that name a call-back function – see section 17 page 24.

3.2 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.

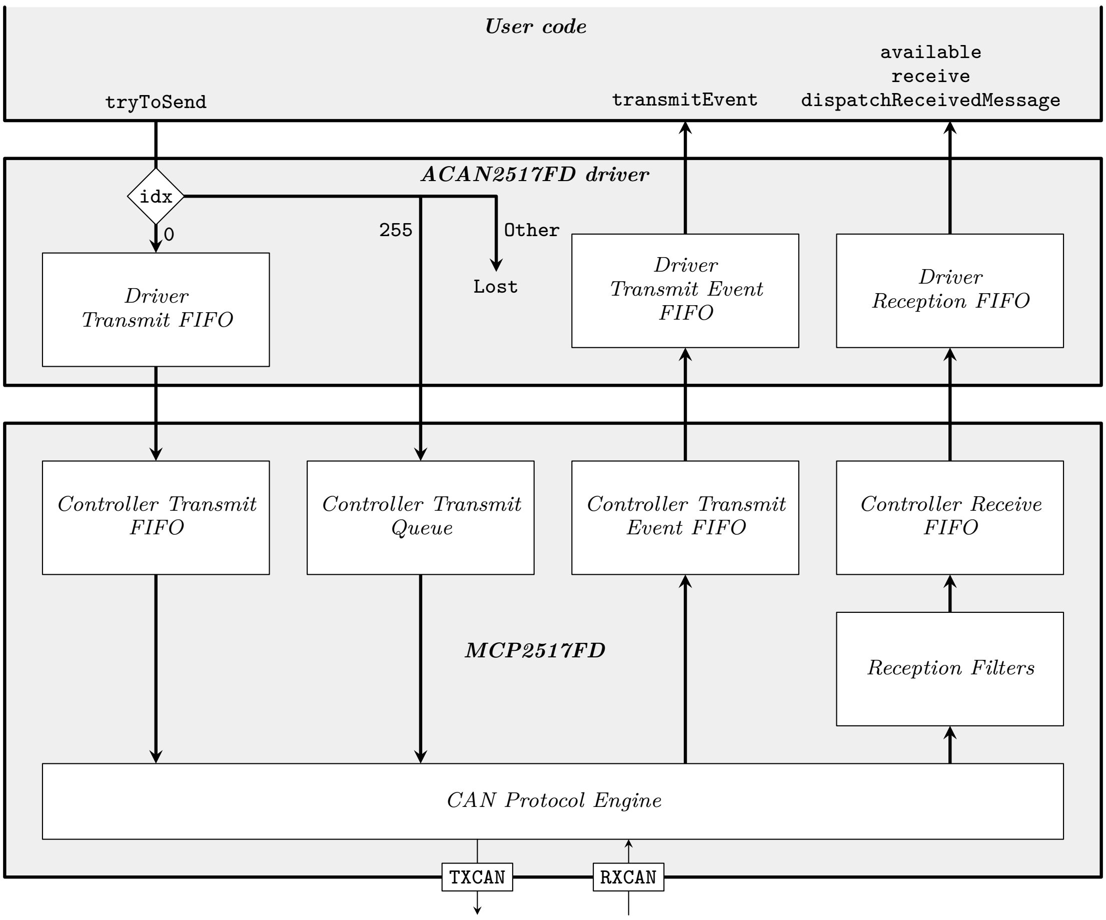

Figure 2 – Message flow in `ACAN2517FD` driver and `MCP2517FD` CAN Controller, custom configuration

You can allocate the *Controller transmit Queue*: send order is defined by frame priority (see section 10 page 15). You can also define up to 32 receive filters (see section 16 page 21). Sizes of `MCP2517FD` internal buffer are easily customizable.

4 A simple example: LoopBackDemo

The following code is a sample code for introducing the ACAN2517FD 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 <ACAN2517FD.h>
```

This line includes the ACAN2517FD 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 $\overline{\texttt{CS}}$ and $\overline{\texttt{INT}}$ pins (adapt to your design).

```
ACAN2517FD can (MCP2517_CS, SPI, MCP2517_INT) ;
```

Instanciation of the ACAN2517FD 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.

```
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 7.1 page 10), selecting alternate pins should be done before calling SPI.begin, on Adafruit Feather M0 (section 7.2 page 12), this should be done after. Calling SPI.begin explicitly allows you to fully handle alternate pins.

```
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               125 * 1000, ACAN2517FDSettings::DATA_BITRATE_x4) ;
```

Configuration is a four-step operation. This line is the first step. It instanciates the settings object of the ACAN2517FDSettings class. The constructor has three parameters: the MCP2517FD oscillator specification, the desired CAN arbitration bit rate (here, 125 kb/s), and the data bit rate, given by a multiplicative factor of the arbitration bit rate; here, the data bit rate is 125 kb/s * 4 = 500 kbit/s. It returns a `settings` object fully initialized with CAN bit settings for the desired arbitration and data bit rates, and default values for other configuration properties.

```
settings.mRequestedMode = ACAN2517FDSettings::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 `NormalFD` 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 19.11 page 36 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 16 page 21. The second argument is the *interrupt service routine*, and is defined by a C++ lambda expression[1]. See section 18.2 page 25 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 18.3 page 26.

```
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() {
  CANFDMessage 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 ;
```

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

```
      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.

5 The `CANFDMessage` class

Note. The `CANFDMessage` class is declared in the `CANFDMessage.h` header file. The class declaration is protected by an include guard that causes the macro `GENERIC_CANFD_MESSAGE_DEFINED` to be defined. This allows an other library to freely include this file without any declaration conflict.

A CAN FD message is an object that contains all CAN FD 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 CANFDMessage {
  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 ... 64)
  public : union {
    uint64_t data64 [ 8] ; // Caution: subject to endianness
    uint32_t data32 [16] ; // Caution: subject to endianness
    uint16_t data16 [32] ; // Caution: subject to endianness
    uint8_t  data   [64] = {} ;
  } ;
  ...
} ;
```

Note the message datas are defined by an **`union`**. So message datas can be seen as 64 bytes, 32 x 16-bit unsigned integers, 16 x 32-bit, or 8 x 64-bit. Be aware that multi-byte integers are subject to endianness (Cortex M4 processors of Teensy 3.x are little-endian).

5.1 The `len` property

Note that `len` field contains the actual length, not its encoding in CAN FD frames. So valid values are: 0, 1, ..., 8, 12, 16, 20, 24, 32, 48, 64. Having other values is an error that prevents frame to be sent by the `ACAN2517FD::tryToSend` method. You can use the `pad` method (see below) for padding with `0x00` bytes to the next valid length

5.2 The `idx` property

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

- for a received message, it contains the acceptance filter index (see section 17 page 24);
- on sending messages, it is used for selecting the transmit buffer (see section 14 page 18).

5.3 The `pad` method

```
void CANFDMessage::pad (void) ;
```

The `CANFDMessage::pad` method appends zero bytes to datas for reaching the next valid length. Valid lengths are: 0, 1, ..., 8, 12, 16, 20, 24, 32, 48, 64. If the length is already valid, no padding is performed. For example:

```
  CANFDMessage frame ;
  frame.length = 21 ; // Not a valid value for sending
  frame.pad () ;
// frame.length is 24, frame.data [21] is 0, frame.data [22] is 0, frame.data [23] is 0
```

5.4 The `isValid` method

```
bool CANFDMessage::isValid (void) const ;
```

Not all settings of `CANFDMessage` instances represent a valid frame. For example, there is no CAN FD remote frame, so a remote frame should have its length lower than or equal to 8. There is no constraint on extended / standard identifier (`ext` property). The table 1 page 9 lists the constraints on `rtr` and `len` properties. An `CANFDMessage` instance that checks theses conditions is valid.

Frame	**rtr**	**len**
CAN 2.0B Remote Frame	`true`	$0 \leqslant len \leqslant 8$
CAN 2.0B Data Frame	`false`	$0 \leqslant len \leqslant 8$
CAN FD Data Frame	`false`	12, 16, 20, 24, 32, 48, 64

Table 1 – `CANFDMessage` validity constraints

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 ACAN2517FD driver in a same sketch.

A *CAN message* is an object that contains all CAN 2.0B 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. In the `ACAN2517FD` library, the `CANMessage` class is only used for sending CAN 2.0B messages by a `ACAN2517FD::tryToSend` method.

```
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} ;
  } ;
} ;
```

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 17 page 24);
- on sending messages, it is used for selecting the transmit buffer (see section 14 page 18).

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 $\overline{\texttt{CS}}$ 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);
- the $\overline{\texttt{INT0}}$ and $\overline{\texttt{INT1}}$ signals are not used by driver and are left not connected.

7.1 Using alternate pins on Teensy 3.x

Demo sketch: `LoopBackDemoTeensy3x`.

[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.

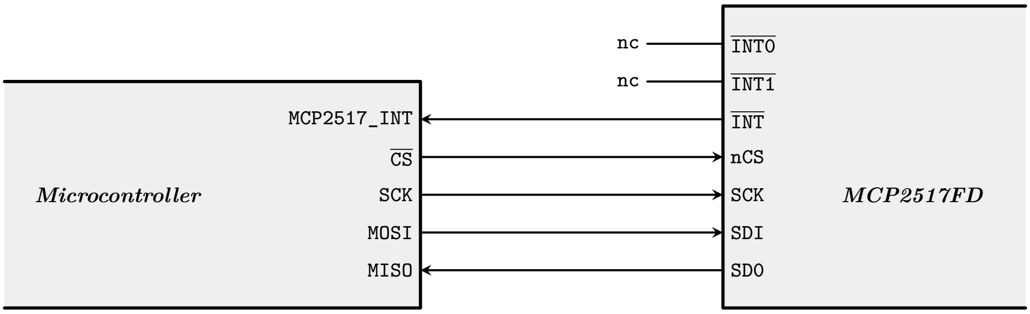

Figure 3 – `MCP2517FD` connection to a microcontroller

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]:

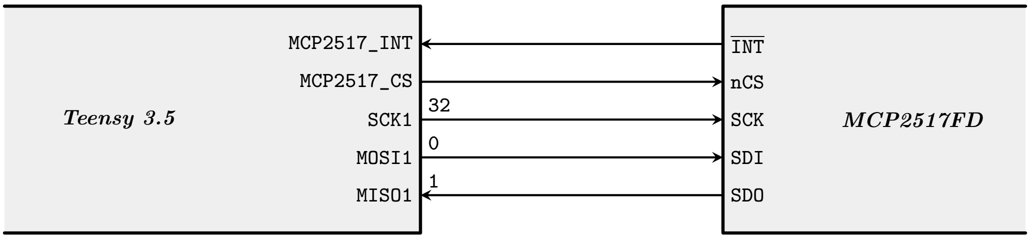

Figure 4 – Using SPI alternate pins on a Teensy 3.5

You call the `SPI1.setMOSI`, `SPI1.setMISO`, and `SPI1.setSCK` functions **before** calling the `begin` function of your `ACAN2517FD` instance:

```
ACAN2517FD 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:

```
void setup () {
  ...
  Serial.print ("Using pin #") ;
  Serial.print (MCP2517_SDI) ;
```

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

```
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 () ; }) ;
...
```

7.2 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 alternate SPI pins on Adafruit Feather M0.

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

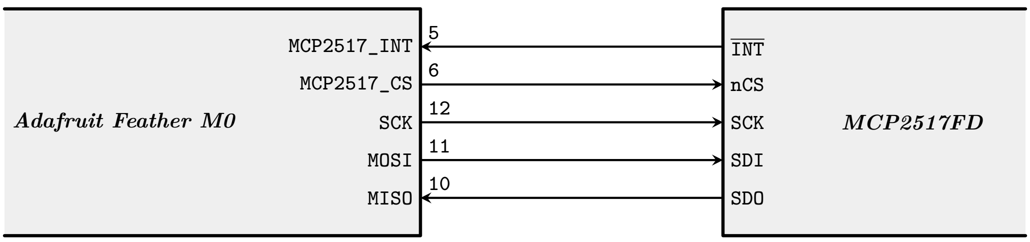

Figure 5 – Using SPI alternate pins on an Adafruit Feather M0

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
...
ACAN2517FD 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 () ; }) ;
  ...
```

8 Clock configuration

The `MCP251xFD` Oscillator Block Diagram is given in figure 6. 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]

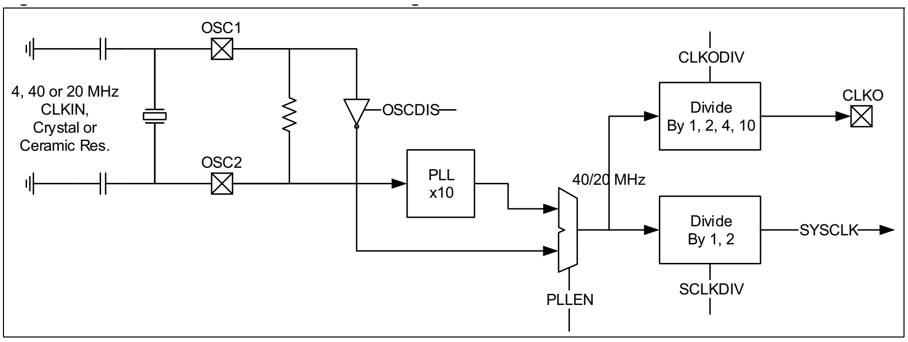

Figure 6 – `MCP251xFD` Oscillator Block Diagram (`DS20005678B`, figure 3.1 page 13)

The `ACAN2517FDSettings` class defines an enumerated type for specifying your settings:

```
class ACAN2517FDSettings {
  public: typedef enum {
    OSC_4MHz,
    OSC_4MHz_DIVIDED_BY_2,
    OSC_4MHz10xPLL,
    OSC_4MHz10xPLL_DIVIDED_BY_2,
    OSC_20MHz,
    OSC_20MHz_DIVIDED_BY_2,
    OSC_40MHz,
    OSC_40MHz_DIVIDED_BY_2
  } Oscillator ;
  ...
} ;
```

The first argument of the `ACAN2517FDSettings` constructor specifies the oscillator. For example, with a 4 MHz clock, the `ACAN2517FDSettings::OSC_4MHz10xPLL` setting leads to a 40 MHz `SYSCLK`.

The eight clock settings are given in the table 2. Note Microchip recommends a 40 MHz or 20 MHz `SYSCLK`. The `ACAN2517FDSettings` class has two accessors that return current settings: `oscillator()` and `sysClock()`.

[6] DS20005678B, page 13.

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 2 – The `ACAN2517FD` oscillator selection

The `begin` function of `ACAN2517FD` library first configures the selected SPI with a frequency of 1 Mbit/s, 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 4) 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 three other parameters you can override:

- `settings.mControllerTransmitFIFORetransmissionAttempts` is the number of retransmission attempts; by default, it is set to `UnlimitedNumber`; other values are `Disabled` and `ThreeAttempts`;
- `settings.mControllerTransmitFIFOPriority` is the priority of the transmit FIFO: between 0 (lowest priority) and 31 (highest priority); default value is 0;

- `settings.mControllerTransmitFIFOPayload` is the controller transmit FIFO object payload size; default value is `PAYLOAD_64`, enabled sending any CAN FD frame; see section 12 page 16.

The *controller transmit FIFO* is located in the `MCP2517FD` RAM. It requires 16 bytes for each message (see section 13 page 17).

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

There are three 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`;

[7] DS20005678B, section 4.5, page 28.

- `inSettings.mControllerTXQBufferPriority` is the priority of the TXQ buffer: between 0 (lowest priority) and 31 (highest priority); default value is 31;
- `inSettings.mControllerTXQBufferPayload` is the controller TXQ buffer object payload size; default value is `PAYLOAD_64`, enabled sending any CAN FD frame; see section 12 page 16.

The *transmit queue* is located in the `MCP2517FD` RAM. It requires 16 bytes for each message (see section 13 page 17).

11 Receive FIFO

The receive FIFO (see figure 1 page 4) 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.

You can override the `mControllerReceiveFIFOPayload` value, which represents the controller receive FIFO object payload size; default value is `PAYLOAD_64`, enabled receiving any CAN FD frame. See section 12 page 16.

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 `ACAN2517FD::available`, `ACAN2517FD::receive` and `ACAN2517FD::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 Payload size

Controller transmit FIFO, controller TXQ buffer and controller receive FIFO objects are stored in the internal `MCP2517FD` RAM. The size of each object depends on the setting applied to the corresponding FIFO or buffer.

By default, all FIFOs and buffer accept objects up to 64 data bytes. The size of each object is 72 bytes. As the internal `MCP2517FD` RAM has a capacity of 2048 bytes, only 28 objects are available, and they are allocated as follows:

- controller transmit FIFO (`mControllerTransmitFIFOSize` property): 4 objects;
- controller TXQ buffer (`mControllerTXQSize` property): no object;

- controller receive FIFO (mControllerReceiveFIFOSize property): 24 objects.

The details of RAM usage computation are presented in section 13 page 17.

Note the ACAN2517 library[8] handles an MCP2517FD in CAN 2.0B mode. As CAN 2.0B frames contains at most 8 bytes, the size of each object is 16 bytes, allowing using up to 128 objects.

With the mControllerTransmitFIFOPayload, the mControllerTXQBufferPayload and the mControllerReceiveFIFOPayload properties, you can adjust the object size following your application requirements. The table 3 shows the possible values of these properties and the corresponding payload and object size.

By example, suppose your application allways send data frames with no more than 24 bytes. You can set the mControllerTransmitFIFOPayload and mControllerReceiveFIFOPayload properties to ACAN2517FDSettings::PAYLOAD_24, leading to an object size equal to 32 bytes. If your application also receives data frames with no more than 24 bytes, you can also set the mControllerReceiveFIFOPayload property to ACAN2517FDSettings::PAYLOAD_24. All your objects require 32 bytes, allowing 64 objects in the MCP2517FD RAM. The benefit is you can now increase controller buffer sizes, for example:

- controller transmit FIFO (mControllerTransmitFIFOSize property): 16 objects;
- controller TXQ buffer (mControllerTXQSize property): 16 objects;
- controller receive FIFO (mControllerReceiveFIFOSize property): 32 objects.

Object Size specification	Payload	Object Size
ACAN2517FDSettings::PAYLOAD_8	Up to 8 bytes	16 bytes
ACAN2517FDSettings::PAYLOAD_12	Up to 12 bytes	20 bytes
ACAN2517FDSettings::PAYLOAD_16	Up to 16 bytes	24 bytes
ACAN2517FDSettings::PAYLOAD_20	Up to 20 bytes	28 bytes
ACAN2517FDSettings::PAYLOAD_24	Up to 24 bytes	32 bytes
ACAN2517FDSettings::PAYLOAD_32	Up to 32 bytes	40 bytes
ACAN2517FDSettings::PAYLOAD_48	Up to 48 bytes	56 bytes
ACAN2517FDSettings::PAYLOAD_64	Up to 64 bytes	72 bytes

Table 3 – ACAN2517FD object size from payload size specification

12.1 The ACAN2517FDSettings::objectSizeForPayload static method

```
uint32_t ACAN2517FDSettings::objectSizeForPayload (const PayloadSize inPayload) ;
```

This static method returns the object size for a given payload specification, following table 3.

13 RAM usage

The MCP2517FD contains a 2048 bytes RAM that is used to store message objects[9]. 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] https://github.com/pierremolinaro/acan2517

[9] DS20005688B, section 3.3, page 63.

- Receive Message Objects used by the receive FIFO.

There are six parameters that affect the required memory amount:

- the `mControllerTransmitFIFOSize` property sets the controller transmit FIFO object count;
- the `mControllerTransmitFIFOPayload` property defines the controller transmit FIFO object size;
- the `mControllerTXQSize` property sets the controller TXQ buffer object count;
- the `mControllerTXQBufferPayload` property defines the controller TXQ buffer object size;
- the `mControllerReceiveFIFOSize` property sets the controller receive FIFO object count;
- the `mControllerReceiveFIFOPayload` property defines the controller receive FIFO object size.

The `ACAN2517FDSettings::ramUsage` method computes the required memory amount as follows:

```
uint32_t ACAN2517FDSettings::ramUsage (void) const {
  uint32_t r = 0 ;
//--- TXQ
  r += objectSizeForPayload(mControllerTXQBufferPayload) * mControllerTXQSize;
//--- Receive FIFO (FIFO #1)
  r += objectSizeForPayload(mControllerReceiveFIFOPayload) * mControllerReceiveFIFOSize;
//--- Send FIFO (FIFO #2)
  r += objectSizeForPayload(mControllerTransmitFIFOPayload) * mControllerTransmitFIFOSize;
//---
  return r ;
}
```

The `ACAN2517FD: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].

14 Sending frames: the tryToSend methods

There are two `ACAN2517FD::tryToSend` methods. The first one handles sending CAN 2.0B and CAN FD frames:

```
bool ACAN2517FD::tryToSend (const CANFDMessage & inMessage) ;
```

For convience, a second prototype is provided, which sends only CAN 2.0B frames:

```
bool ACAN2517FD::tryToSend (const CANMessage & inMessage) ;
```

```
  ...
  CANFDMessage 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:

[10] http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD%20RAM%20Usage%20Calculations%20-%20UG.xlsx

- 0 for the transmit FIFO (section 9 page 14) ;
- 255 for the transmit Queue (section 10 page 15).

The `tryToSend` method returns:

- `false` if the message responds `false` to the `isValid` method (see section 5.4 page 9), or if its `len` property has a value greater than the corresponding buffer payload; an invalid message is never submitted to a transmit buffer;
- otherwise, if the message responds `true` to the `isValid` method:
  - `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:

```
static uint32_t gSendDate = 0 ;

void loop () {
  if (gSendDate < millis ()) {
    CANFDMessage 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 ;
    }
  }
  ...
}
```

15 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 17 page 24).

This is a basic example:

```
void loop () {
  CANFDMessage 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 () {
  CANFDMessage 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 CANFDMessage & inMessage) {
  ...
}
```

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

15.1 Driver receive buffer size

By default, the driver receive buffer size is 24. You can change it by setting the `mReceiveBufferSize` property of `settings` variable before calling the `begin` method:

```
ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                             125 * 1000, ACAN2517FDSettings::DATA_BITRATE_x4) ;
settings.mReceiveBufferSize = 100 ;
const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
...
```

As the size of `CANFDMessage` class is 72 bytes, the actual size of the driver receive buffer is the value of `settings.mReceiveBufferSize * 72`.

15.2 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 () ;
```

15.3 The receiveBufferCount method

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

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

15.4 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`.

16 Acceptance filters

Note. The acceptance filters implemented in the `ACAN2517` library, that handles a `MCP2517FD` CAN Controller in the CAN 2.0B mode[11], are almost identical, they differ only from the prototype of the callback routine.

If you invoke the `ACAN2517FD.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 `MCP2517FDFilters` instance object, and pass it as third argument of the `ACAN2517FD.begin` method:

```
  MCP2517FDFilters filters ;
  ... // Append filters
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }, filters) ;
  ...
```

[11] https://github.com/pierremolinaro/acan2517

16.1 An example

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

```
  MCP2517FDFilters filters ;
```

First, you instanciate an MCP2517FDFilters 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 17 page 24).

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.

```
// 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:

identifier & 0x70F == 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 () != MCP2517FDFilters::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 MCP2517FDFilters::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 `MCP2517FDFilters::kFiltersOk`).

16.2 The appendPassAllFilter method

```
void MCP2517FDFilters::appendPassAllFilter (const ACANFDCallBackRoutine 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.

16.3 The appendFormatFilter method

```
void MCP2517FDFilters::appendFormatFilter (const tFrameFormat inFormat,
                                           const ACANFDCallBackRoutine 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.

16.4 The appendFrameFilter method

```
void MCP2517FDFilters::appendFrameFilter (const tFrameFormat inFormat,
                                          const uint32_t inIdentifier,
                                          const ACANFDCallBackRoutine 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.

16.5 The appendFilter method

```
void MCP2517FDFilters::appendFilter (const tFrameFormat inFormat,
                                     const uint32_t inMask,
                                     const uint32_t inAcceptance,
                                     const ACANFDCallBackRoutine 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.

17 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.

18 The `ACAN2517FD::begin` method reference

18.1 The prototypes

```
uint32_t ACAN2517FD::begin (const ACAN2517FDSettings & inSettings,
                            void (* inInterruptServiceRoutine) (void)) ;
```

This prototype has two arguments, a `ACAN2517FDSettings` instance that defines the settings, and the interrupt service routine, that can be specified by a lambda expression or a function (see section 18.2 page 25). It configures the controller in such a way that all messages are received (*pass-all* filter).

```
uint32_t ACAN2517FD::begin (const ACAN2517FDSettings & inSettings,
                            void (* inInterruptServiceRoutine) (void),
                            const MCP2517FDFilters & inFilters) ;
```

The second prototype has a third argument, an instance of `MCP2517FDFilters` class that defines the receive filters.

18.2 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) ;
```

18.3 The error code

The `ACAN2517FD::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 `ACAN2517FD` class defines static constants for naming errors.

Bit	Static constant Name	Link
0	`kRequestedConfigurationModeTimeOut`	section 18.3.1 page 26
1	`kReadBackErrorWith1MHzSPIClock`	section 18.3.2 page 26
2	`kTooFarFromDesiredBitRate`	section 18.3.3 page 26
3	`kInconsistentBitRateSettings`	section 18.3.4 page 27
4	`kINTPinIsNotAnInterrupt`	section 18.3.5 page 27
5	`kISRIsNull`	section 18.3.6 page 27
6	`kFilterDefinitionError`	section 18.3.7 page 27
7	`kMoreThan32Filters`	section 18.3.8 page 27
8	`kControllerReceiveFIFOSizeIsZero`	section 18.3.9 page 27
9	`kControllerReceiveFIFOSizeGreaterThan32`	section 18.3.10 page 27
10	`kControllerTransmitFIFOSizeIsZero`	section 18.3.11 page 27
11	`kControllerTransmitFIFOSizeGreaterThan32`	section 18.3.12 page 27
12	`kControllerRamUsageGreaterThan2048`	section 18.3.13 page 28
13	`kControllerTXQPriorityGreaterThan31`	section 18.3.14 page 28
14	`kControllerTransmitFIFOPriorityGreaterThan31`	section 18.3.15 page 28
15	`kControllerTXQSizeGreaterThan32`	section 18.3.16 page 28
16	`kRequestedModeTimeOut`	section 18.3.17 page 28
17	`kX10PLLNotReadyWithin1MS`	section 18.3.18 page 28
18	`kReadBackErrorWithFullSpeedSPIClock`	section 18.3.19 page 28

Table 4 – The `ACAN2517FD::begin` method error code bits

18.3.1 `kRequestedConfigurationModeTimeOut`

The `ACAN2517FD::begin` method first configures SPI with a 1 Mbit/s clock, and then requests the configuration mode. This error is raised when the `LCP2517FD` does not reach the configuration mode with 2ms. It means that the `MCP2517FD` cannot be accessed via SPI.

18.3.2 `kReadBackErrorWith1MHzSPIClock`

Then, the `ACAN2517FD::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.

18.3.3 `kTooFarFromDesiredBitRate`

This error occurs when the `mArbitrationBitRateClosedToDesiredRate` property of the `settings` object is `false`. This means that the `ACAN2517FDSettings` constructor cannot compute a CAN bit configuration close enough to the desired bit rate. For example:

```
void setup () {
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               1, ACAN2517FDSettings::DATA_BITRATE_x1) ; // 1 bit/s !!!
  // Here, settings.mArbitrationBitRateClosedToDesiredRate is false
  const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
```

```
  // Here, errorCode contains ACAN2517FD::kCANBitConfigurationTooFarFromDesiredBitRate
}
```

18.3.4 `kInconsistentBitRateSettings`

The `ACAN2517FDSettings` 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`, `mArbitrationPhaseSegment1`, `mArbitrationPhaseSegment2`, `mArbitrationSJW`), and one or more resulting values are inconsistent. See section 19.2 page 34.

18.3.5 `kINTPinIsNotAnInterrupt`

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

18.3.6 `kISRIsNull`

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

18.3.7 `kFilterDefinitionError`

`settings.filterStatus()` returns a value different than `MCP2517FDFilters::kFiltersOk`, meaning that one or more filters are erroneous. See section 16.1 page 22.

18.3.8 `kMoreThan32Filters`

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

18.3.9 `kControllerReceiveFIFOSizeIsZero`

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

18.3.10 `kControllerReceiveFIFOSizeGreaterThan32`

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

18.3.11 `kControllerTransmitFIFOSizeIsZero`

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

18.3.12 `kControllerTransmitFIFOSizeGreaterThan32`

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

18.3.13 `kControllerRamUsageGreaterThan2048`

The configuration you have defined requires more than 2048 bytes of `MCP2517FD` internal RAM. See section 13 page 17.

18.3.14 `kControllerTXQPriorityGreaterThan31`

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

18.3.15 `kControllerTransmitFIFOPriorityGreaterThan31`

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

18.3.16 `kControllerTXQSizeGreaterThan32`

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

18.3.17 `kRequestedModeTimeOut`

During configuration by the `ACAN2517FD::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.

18.3.18 `kX10PLLNotReadyWithin1MS`

You have requested the `QUARTZ_4MHz10xPLL` oscillator mode, enabling the 10x PLL. The `ACAN2517FD::begin` method waits during 2ms the PLL to be locked. This error is raised when the PLL is not locked within 2 ms.

18.3.19 `kReadBackErrorWithFullSpeedSPIClock`

After the oscillator configuration has been established, the `ACAN2517FD::begin` method configures the SPI at its full speed (`SYSCLK/2`, and checks accessibility by writing and reading back 32 32-bit values at the first `MCP2517FD` RAM address (`0x400`). The 32 used values are $1 << n$, with $0 \leqslant n \leqslant 31$. This error is raised when the read value is different from the written one.

19 `ACAN2517FDSettings` class reference

Note. The `ACAN2517FDSettings` class is not Arduino specific. You can compile it on your desktop computer with your favorite C++ compiler. In the `https://github.com/pierremolinaro/acan2517FD-dev` GitHub repository, a command line tool is defined for exploring all CAN arbitration bit rates from 1 bit/s to 1 Mbit/s. It also checks that computed CAN bit decompositions are all consistent, even if they are too far from the desired baud rate.

19.1 The ACAN2517FDSettings constructor: computation of the CAN bit settings

The constructor of the `ACAN2517FDSettings` has three mandatory arguments: the oscillator frequency, the desired arbitration bit rate, and the data bit rate factor. It tries to compute the CAN bit settings for theses bit rates. If it succeeds, the constructed object has its `mArbitrationBitRateClosedToDesiredRate` property set to `true`, otherwise it is set to `false`. For example, for an 1 Mbit/s arbitration bit rate and an 8 Mbit/s data bit rate:

```
void setup () {
 // Arbitration bit rate: 1 Mbit/s, data bit rate: 8 Mbit/s
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                              1 * 1000 * 1000, ACAN2517FDSettings::DATA_BITRATE_x8) ;
  // Here, settings.mArbitrationBitRateClosedToDesiredRate is true
  ...
}
```

Note the data bit rate is not defined by its frequency, but by its multiplicative factor from arbitration bit rate. If you want a single bit rate, use `ACAN2517FDSettings::DATA_BITRATE_x1` as data bit rate factor.

Of course, with a 40 MHz or 20 MHz `SYSCLK`, CAN bit computation allways succeeds for classical arbitration bit rates: 1 Mbit/s, 500 kbit/s, 250 kbit/s, 125 kbit/s. With a 40 MHz `SYSCLK`, there are 184 exact arbitration / data bit rate combinations (table 5 page 30), and 178 with a 20 MHz `SYSCLK` (table 6 page 31). Note a 8 MHz data bit rate cannot be performed with a 20 MHz `SYSCLK`. By "exact", we mean that arbitration bit rate and data bit rate are both exactly integer values. There is no such combination for data bit rate factors `3x`, `6x`, `7x`.

But this does not mean there is no possibility to get such data bit rates factors. For example, we can have a data bit rate of 4 Mbit/s, and an arbitration bit rate of 4/7 Mbit/s = 571 428 kbit/s:

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                              571428, ACAN2517FDSettings::DATA_BITRATE_x7) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("Actual Arbitration Bit Rate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  571428 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 1 ppm= 0,0001 %
  Serial.print ("Actual Data Bit Rate: ") ;
  Serial.println (settings.actualDataBitRate ()) ; //  4 Mbit/s
  ...
}
```

Due to integer computations, and the distance from desired arbitration bit rate is 1 ppm. "ppm" stands for "part-per-million", and 1 ppm $= 10^{-6}$. In other words, $10,000$ ppm $= 1\%$.

By default, a desired bit rate is accepted if the distance from the computed actual bit rate is lower or equal to $1,000$ ppm $= 0.1$ %. You can change this default value by adding your own value as fourth argument of `ACAN2517FDSettings` constructor. Foe example, with an arbitration bit rate equal to 727 kbit/s:

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                              727 * 1000, ACAN2517FDSettings::DATA_BITRATE_x1,
                              100) ; // 100 ppm
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual arbitration bit rate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  727272 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 375 ppm
```

Arbitration Bit Rate	Valid Data Rate factors
500 bit/s	1x 8x
625 bit/s	1x 8x
640 bit/s	1x
800 bit/s	1x 5x 8x
1 kbit/s	1x 4x 5x 8x
1250 bit/s	1x 4x 5x 8x
1280 bit/s	1x 5x
1600 bit/s	1x 4x 5x 8x
2 kbit/s	1x 2x 4x 5x 8x
2500 bit/s	1x 2x 4x 5x 8x
2560 bit/s	1x 5x
3125 bit/s	1x 2x 4x 5x 8x
3200 bit/s	1x 2x 4x 5x
4 kbit/s	1x 2x 4x 5x 8x
5 kbit/s	1x 2x 4x 5x 8x
6250 bit/s	1x 2x 4x 5x 8x
6400 bit/s	1x 2x 5x
8 kbit/s	1x 2x 4x 5x 8x
10 kbit/s	1x 2x 4x 5x 8x
12500 bit/s	1x 2x 4x 5x 8x
12800 bit/s	1x 5x
15625 bit/s	1x 2x 4x 5x 8x
16 kbit/s	1x 2x 4x 5x
20 kbit/s	1x 2x 4x 5x 8x
25 kbit/s	1x 2x 4x 5x 8x
31250 bit/s	1x 2x 4x 5x 8x
32 kbit/s	1x 2x 5x
40 kbit/s	1x 2x 4x 5x 8x
50 kbit/s	1x 2x 4x 5x 8x
62500 bit/s	1x 2x 4x 5x 8x
64 kbit/s	1x 5x
78125 bit/s	1x 2x 4x 8x
80 kbit/s	1x 2x 4x 5x
100 kbit/s	1x 2x 4x 5x 8x
125 kbit/s	1x 2x 4x 5x 8x
156250 bit/s	1x 2x 4x 8x
160 kbit/s	1x 2x 5x
200 kbit/s	1x 2x 4x 5x 8x
250 kbit/s	1x 2x 4x 5x 8x
312500 bit/s	1x 2x 4x 8x
320 kbit/s	1x 5x
400 kbit/s	1x 2x 4x 5x
500 kbit/s	1x 2x 4x 5x 8x
625 kbit/s	1x 2x 4x 8x
800 kbit/s	1x 2x 5x
1000 kbit/s	1x 2x 4x 5x 8x

Table 5 – 40 MHz SYSCLK: the 184 exact bit rates

```
  ...
}
```

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

```
void setup () {
```

Arbitration Bit Rate	Valid Data Rate factors
250 bit/s	1x 8x
320 bit/s	1x
400 bit/s	1x 5x 8x
500 bit/s	1x 4x 5x 8x
625 bit/s	1x 4x 5x 8x
640 bit/s	1x 5x
800 bit/s	1x 4x 5x 8x
1 kbit/s	1x 2x 4x 5x 8x
1250 bit/s	1x 2x 4x 5x 8x
1280 bit/s	1x 5x
1600 bit/s	1x 2x 4x 5x
2 kbit/s	1x 2x 4x 5x 8x
2500 bit/s	1x 2x 4x 5x 8x
3125 bit/s	1x 2x 4x 5x 8x
3200 bit/s	1x 2x 5x
4 kbit/s	1x 2x 4x 5x 8x
5 kbit/s	1x 2x 4x 5x 8x
6250 bit/s	1x 2x 4x 5x 8x
6400 bit/s	1x 5x
8 kbit/s	1x 2x 4x 5x
10 kbit/s	1x 2x 4x 5x 8x
12500 bit/s	1x 2x 4x 5x 8x
15625 bit/s	1x 2x 4x 5x 8x
16 kbit/s	1x 2x 5x
20 kbit/s	1x 2x 4x 5x 8x
25 kbit/s	1x 2x 4x 5x 8x
31250 bit/s	1x 2x 4x 5x 8x
32 kbit/s	1x 5x
40 kbit/s	1x 2x 4x 5x
50 kbit/s	1x 2x 4x 5x 8x
62500 bit/s	1x 2x 4x 5x 8x
78125 bit/s	1x 2x 4x 8x
80 kbit/s	1x 2x 5x
100 kbit/s	1x 2x 4x 5x 8x
125 kbit/s	1x 2x 4x 5x 8x
156250 bit/s	1x 2x 4x 8x
160 kbit/s	1x 5x
200 kbit/s	1x 2x 4x 5x
250 kbit/s	1x 2x 4x 5x 8x
312500 bit/s	1x 2x 4x 8x
400 kbit/s	1x 2x 5x
500 kbit/s	1x 2x 4x 5x 8x
625 kbit/s	1x 2x 4x 8x
800 kbit/s	1x 5x
1000 kbit/s	1x 2x 4x 5x

Table 6 – 20 MHz SYSCLK: the 178 exact bit rates

```
...
ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                             500 * 1000, ACAN2517FDSettings::DATA_BITRATE_x1,
                             0) ; // Max distance is 0 ppm
Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (--> is true)
Serial.print ("actual arbitration bit rate: ") ;
Serial.println (settings.actualArbitrationBitRate ()) ; //  500,000 bit/s
```

```
  Serial.print ("distance:␣") ;
  Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 0 ppm
  ...
}
```

In any way, the bit rate computation allways gives a consistent result, resulting an actual arbitration / data bit rates closest from the desired bit rate. For example, we query a 423 kbit/s arbitration bit rate, and a 423 kbit/s * 3 = 1 269 kbit/s data bit rates:

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               423 * 1000, ACAN2517FDSettings::DATA_BITRATE_x6) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("Actual␣Arbitration␣Bit␣Rate:␣") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  416 666 bit/s
  Serial.print ("Actual␣Data␣Bit␣Rate:␣") ;
  Serial.println (settings.actualDataBitRate ()) ; //  1 250 kbit/s
  Serial.print ("distance:␣") ;
  Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 14972 ppm
  ...
}
```

The resulting bit rates settings are far from the desired values, the CAN bit decomposition is consistent. You can get its details:

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               423 * 1000, ACAN2517FDSettings::DATA_BITRATE_x6) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("Actual␣Arbitration␣Bit␣Rate:␣") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  416 666 bit/s
  Serial.print ("Actual␣Data␣Bit␣Rate:␣") ;
  Serial.println (settings.actualDataBitRate ()) ; //  1 250 kbit/s
  Serial.print ("distance:␣") ;
  Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 14972 ppm
  Serial.print ("Bit␣rate␣prescaler:␣") ;
  Serial.println (settings.mBitRatePrescaler) ; // BRP = 2
  Serial.print ("Arbitration␣Phase␣segment␣1:␣") ;
  Serial.println (settings.mArbitrationPhaseSegment1) ; // PS1 = 38
  Serial.print ("Arbitration␣Phase␣segment␣2:␣") ;
  Serial.println (settings.mArbitrationPhaseSegment2) ; // PS2 = 9
  Serial.print ("Arbitration␣Resynchronization␣Jump␣Width:␣") ;
  Serial.println (settings.mArbitrationSJW) ; // SJW = 9
  Serial.print ("Arbitration␣Sample␣Point:␣") ;
  Serial.println (settings.arbitrationSamplePointFromBitStart ()) ; // 81, meaning 81%
  Serial.print ("Data␣Phase␣segment␣1:␣") ;
  Serial.println (settings.mDataPhaseSegment1) ; // PS1 = 12
  Serial.print ("Data␣Phase␣segment␣2:␣") ;
  Serial.println (settings.mDataPhaseSegment2) ; // PS2 = 3
  Serial.print ("Data␣Resynchronization␣Jump␣Width:␣") ;
  Serial.println (settings.mDataSJW) ; // SJW = 3
  Serial.print ("Data␣Sample␣Point:␣") ;
  Serial.println (settings.dataSamplePointFromBitStart ()) ; // 81, meaning 81%
```

```
  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 `mArbitrationPhaseSegment1` value, and decrement the `mArbitrationPhaseSegment2` value in order to sample the `CAN Rx` pin later.

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               500 * 1000, ACAN2517FDSettings::DATA_BITRATE_x1) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate:␣") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (--> is true)
  settings.mArbitrationPhaseSegment1 -= 8 ; // 63 -> 55: safe, 1 <= PS1 <= 256
  settings.mArbitrationPhaseSegment2 += 8 ; // 16 -> 24: safe, 1 <= PS2 <= 128
  settings.mArbitrationSJW += 8           ; // 16 -> 24: safe, 1 <= SJW <= PS2
  Serial.print ("Sample␣Point:␣") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 68, meaning 68%
  Serial.print ("actual␣arbitration␣bit␣rate:␣") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; // 500000: ok, no change
  Serial.print ("Consistency:␣") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

Be aware to allways respect CAN bit timing consistency! The `MCP2517FD` constraints are:

$$1 \leqslant \texttt{mBitRatePrescaler} \leqslant 256$$
$$2 \leqslant \texttt{mArbitrationPhaseSegment1} \leqslant 256$$
$$1 \leqslant \texttt{mArbitrationPhaseSegment2} \leqslant 128$$
$$1 \leqslant \texttt{mArbitrationSJW} \leqslant \texttt{mArbitrationPhaseSegment2}$$
$$2 \leqslant \texttt{mDataPhaseSegment1} \leqslant 32$$
$$1 \leqslant \texttt{mDataPhaseSegment2} \leqslant 16$$
$$1 \leqslant \texttt{mDataSJW} \leqslant \texttt{mDataPhaseSegment2}$$

Miucrochips recommends using the same bit rate prescaler for arbitration and data bit rates.

Resulting actual bit rates are given by:

$$\text{Actual Arbitration Bit Rate} = \frac{\texttt{SYSCLK}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mArbitrationPhaseSegment1} + \texttt{mArbitrationPhaseSegment2})}$$

$$\text{Actual Data Bit Rate} = \frac{\texttt{SYSCLK}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mDataPhaseSegment1} + \texttt{mDataPhaseSegment2})}$$

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

$$\text{Arbitration Sampling Point} = 100 \cdot \frac{1 + \texttt{mArbitrationPhaseSegment1}}{1 + \texttt{mArbitrationPhaseSegment1} + \texttt{mArbitrationPhaseSegment2}}$$

$$\text{Data Sampling Point} = 100 \cdot \frac{1 + \texttt{mDataPhaseSegment1}}{1 + \texttt{mDataPhaseSegment1} + \texttt{mDataPhaseSegment2}}$$

19.2 The `CANBitSettingConsistency` method

This method checks the CAN bit decomposition (given by `mBitRatePrescaler`, `mArbitrationPhaseSegment1`, `mArbitrationPhaseSegment2`, `mArbitrationSJW`, `mDataPhaseSegment1`, `mDataPhaseSegment2`, `mDataSJW` property values) is consistent.

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               500 * 1000, ACAN2517FDSettings::DATA_BITRATE_x2) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (--> is true)
  settings.mDataPhaseSegment1 = 0 ; // Error, mDataPhaseSegment1 should be >= 1 (and <= 32)
  Serial.print ("Consistency: 0x") ;
  Serial.println (settings.CANBitSettingConsistency (), HEX) ; // != 0, 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 7.

The `ACAN2517FDSettings` class defines static constant properties that can be used as mask error. For example:

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

19.3 The `kArbitrationTQCountNotDivisibleByDataBitRateFactor` error

This error occurs when you have changed the properties relative to arbitration and / or data bit rates, and the resulting values provide a data bit rate that is not an integer multiple of arbitration bit rate, that is the `ACAN2517FDSettings::dataBitRateIsAMultipleOfArbitrationBitRate` method returns `false`.

19.4 The `actualArbitrationBitRate` method

The `actualArbitrationBitRate` method returns the actual bit computed from `mBitRatePrescaler`, `mPropagationSegment`, `mArbitrationPhaseSegment1`, `mArbitrationPhaseSegment2`, `mArbitrationSJW` property values.

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               440 * 1000, ACAN2517FDSettings::DATA_BITRATE_x1) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (--> is false)
```

Bit	Error Name	Error
0	kBitRatePrescalerIsZero	mBitRatePrescaler == 0
1	kBitRatePrescalerIsGreaterThan256	mBitRatePrescaler > 256
2	kArbitrationPhaseSegment1IsLowerThan2	mArbitrationPhaseSegment1 < 2
3	kArbitrationPhaseSegment1IsGreaterThan256	mArbitrationPhaseSegment1 > 256
4	kArbitrationPhaseSegment2IsZero	mArbitrationPhaseSegment2 == 0
5	kArbitrationPhaseSegment2IsGreaterThan128	mArbitrationPhaseSegment2 > 128
6	kArbitrationSJWIsZero	mArbitrationSJW == 0
7	kArbitrationSJWIsGreaterThan128	mArbitrationSJW > 128
8	kArbitrationSJWIsGreaterThanPhaseSegment1	mArbitrationSJW > mArbitrationPhaseSegment1
9	kArbitrationSJWIsGreaterThanPhaseSegment2	mArbitrationSJW > mArbitrationPhaseSegment2
10	kArbitrationTQCountNotDivisibleByDataBitRateFactor	See section 19.3 page 34
11	kDataPhaseSegment1IsLowerThan2	mDataPhaseSegment1 < 2
12	kDataPhaseSegment1IsGreaterThan32	mDataPhaseSegment1 > 32
13	kDataPhaseSegment2IsZero	mDataPhaseSegment2 == 0
14	kDataPhaseSegment2IsGreaterThan16	mDataPhaseSegment2 > 16
15	kDataSJWIsZero	mDataSJW == 0
16	kDataSJWIsGreaterThan16	mDataSJW > 16
17	kDataSJWIsGreaterThanPhaseSegment1	mDataSJW > mDataPhaseSegment1
18	kDataSJWIsGreaterThanPhaseSegment2	mDataSJW > mDataPhaseSegment2

Table 7 – The ACAN2517FDSettings::CANBitSettingConsistency method error codes

```
  Serial.print ("actual arbitration bit rate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  444,444 bit/s
  ...
}
```

Note. If CAN bit settings are not consistent (see section 19.2 page 34), the returned value is irrelevant.

19.5 The exactArbitrationBitRate method

```
bool ACAN2517FDSettings::exactArbitrationBitRate (void) const ;
```

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

Note. If CAN bit settings are not consistent (see section 19.2 page 34), the returned value is irrelevant.

19.6 The exactDataBitRate method

```
bool ACAN2517FDSettings::exactDataBitRate (void) const ;
```

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

Note. If CAN bit settings are not consistent (see section 19.2 page 34), the returned value is irrelevant.

19.7 The ppmFromDesiredArbitrationBitRate method

```
uint32_t ACAN2517FDSettings::ppmFromDesiredArbitrationBitRate (void) const ;
```

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

Note. If CAN bit settings are not consistent (see section 19.2 page 34), the returned value is irrelevant.

19.8 The `ppmFromDesiredDataBitRate` method

```
uint32_t ACAN2517FDSettings::ppmFromDesiredDataBitRate (void) const ;
```

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

Note. If CAN bit settings are not consistent (see section 19.2 page 34), the returned value is irrelevant.

19.9 The `arbitrationSamplePointFromBitStart` method

```
uint32_t ACAN2517FDSettings::arbitrationSamplePointFromBitStart (void) const ;
```

The `arbitrationSamplePointFromBitStart` method returns the distance of sample point from the start of the arbitration CAN bit, expressed in part-per-cent (ppc): $1\ \texttt{ppc} = 1\% = 10^{-2}$. 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%.

Note. If CAN bit settings are not consistent (see section 19.2 page 34), the returned value is irrelevant.

19.10 The `dataSamplePointFromBitStart` method

```
uint32_t ACAN2517FDSettings::dataSamplePointFromBitStart (void) const ;
```

The `dataSamplePointFromBitStart` method returns the distance of sample point from the start of the data CAN bit, expressed in part-per-cent (ppc): $1\ \texttt{ppc} = 1\% = 10^{-2}$. 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%.

Note. If CAN bit settings are not consistent (see section 19.2 page 34), the returned value is irrelevant.

19.11 Properties of the `ACAN2517FDSettings` class

All properties of the `ACAN2517FDSettings` class are declared `public` and are initialized (table 8). 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.

19.11.1 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.

19.11.2 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 6 page 13.

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 19.1 page 29
mArbitrationPhaseSegment1	uint16_t	0	See section 19.1 page 29
mArbitrationPhaseSegment2	uint8_t	0	See section 19.1 page 29
mArbitrationSJW	uint8_t	0	See section 19.1 page 29
mArbitrationBitRateClosedTo-DesiredRate	bool	false	See section 19.1 page 29
mDataPhaseSegment1	uint16_t	0	See section 19.1 page 29
mDataPhaseSegment2	uint8_t	0	See section 19.1 page 29
mDataSJW	uint8_t	0	See section 19.1 page 29
mDataBitRateClosedToDesiredRate	bool	false	See section 19.1 page 29
mTXCANIsOpenDrain	bool	false	See section 19.11.1 page 36
mCLKOPin	CLKOpin	CLKO_DIVIDED_BY_10	See section 19.11.2 page 36
mISOCRCEnabled	bool	true	See section 19.11.4 page 38
mRequestedMode	RequestedMode	NormalFD	See section 19.11.3 page 38
mDriverTransmitFIFOSize	uint16_t	16	See section 9 page 14
mControllerTransmitFIFOSize	uint8_t	32	See section 9 page 14
mControllerTransmitFIFOPayload	PayloadSize	PAYLOAD_64	See section 9 page 14
mControllerTransmitFIFOPriority	uint8_t	0	See section 9 page 14
mControllerTransmitFIFO-RetransmissionAttempts	RetransmissionAttempts	UnlimitedNumber	See section 9 page 14
mControllerTXQSize	uint8_t	0	See section 10 page 15
mControllerTXQBufferPayload	PayloadSize	PAYLOAD_64	See section 10 page 15
mControllerTXQBufferPriority	uint8_t	31	See section 10 page 15
mControllerTXQBuffer-RetransmissionAttempts	RetransmissionAttempts	UnlimitedNumber	See section 10 page 15
mDriverReceiveFIFOSize	uint16_t	32	See section 11 page 16
mControllerReceiveFIFOPayload	PayloadSize	PAYLOAD_64	See section 11 page 16
mControllerReceiveFIFOSize	uint8_t	32	See section 11 page 16

Table 8 – Properties of the ACAN2517FDSettings 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 ACAN2517FDSettings class defines an enumerated type for specifying these settings:

```
class ACAN2517FDSettings {
  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:

```
ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL, CAN_BIT_RATE) ;
...
settings.mCLKOPin = ACAN2517FDSettings::SOF ;
...
const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
```

19.11.3 The `mRequestedMode` property

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

19.11.4 The `mISOCRCEnabled` property

This property enables ISO CRC in CAN FD Frames bit:

- `true` (default): include Stuff Bit Count in CRC Field and use Non-Zero CRC Initialization Vector according to ISO 11898-1:2015:
- `false`: do NOT include Stuff Bit Count in CRC Field and use CRC Initialization Vector with all zeros.

This setting correspondonds to the `ISOCRCEN` bit of the `CiCON` register.