ACAN2517 Arduino library
For `MCP2517FD`, in CAN 2.0B mode
Version 1.1.1

Pierre Molinaro

January 31, 2019

Contents

1 Versions

Version	Date	Comment
1.1.1	January 31, 2019	New option: no interrupt pin (section 6.5 page 13).
1.1.0	January 27, 2019	First release running on ESP32 (section 6.4 page 11).
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	November 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 `MCP2517FD` CAN controller, 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 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.

3.1 Data flow in default configuration

The figure 1 illustrates message flow in the default configuration.

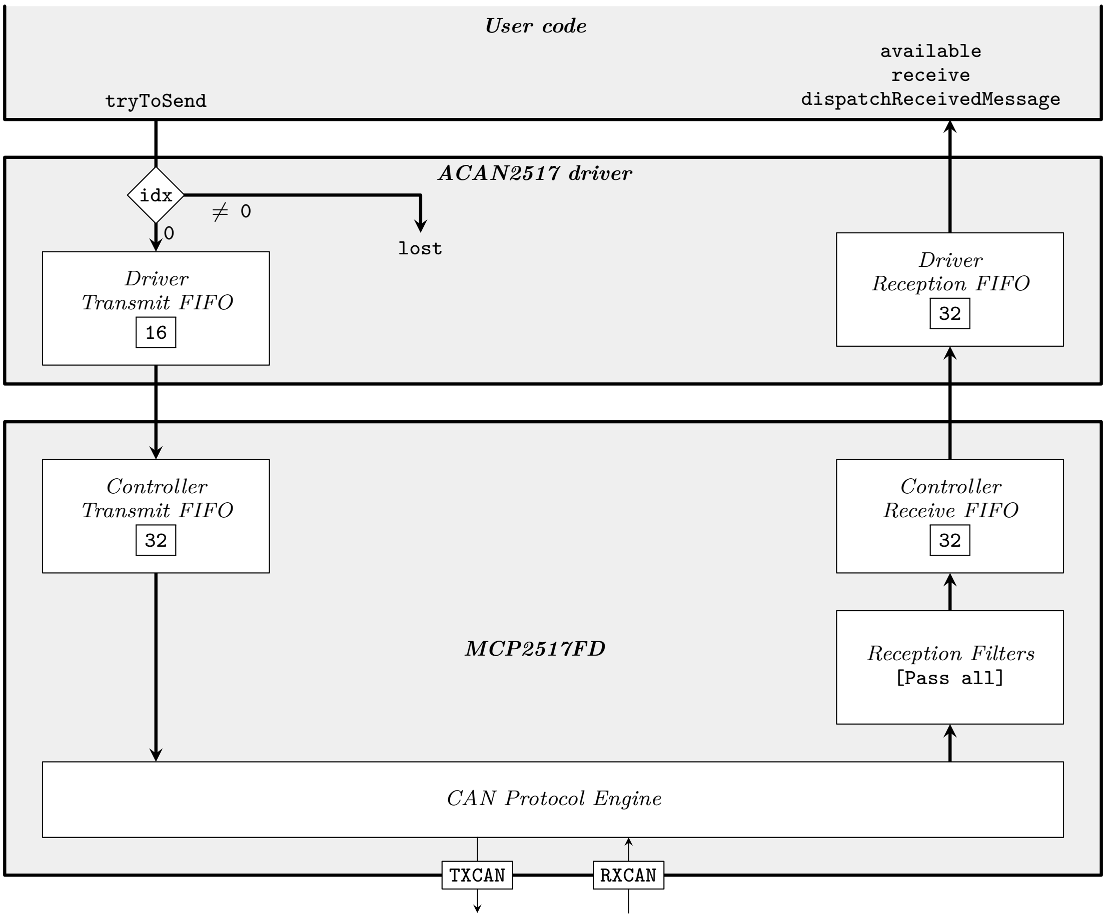

Figure 1 – Message flow in `ACAN2517` driver and `MCP2517FD` CAN Controller, default configuration

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 12 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 14 page 21 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 13.1 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 13 page 19;
- the `dispatchReceivedMessage` method if you have defined the reception filters that name a call-back function – see section 15 page 23.

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.

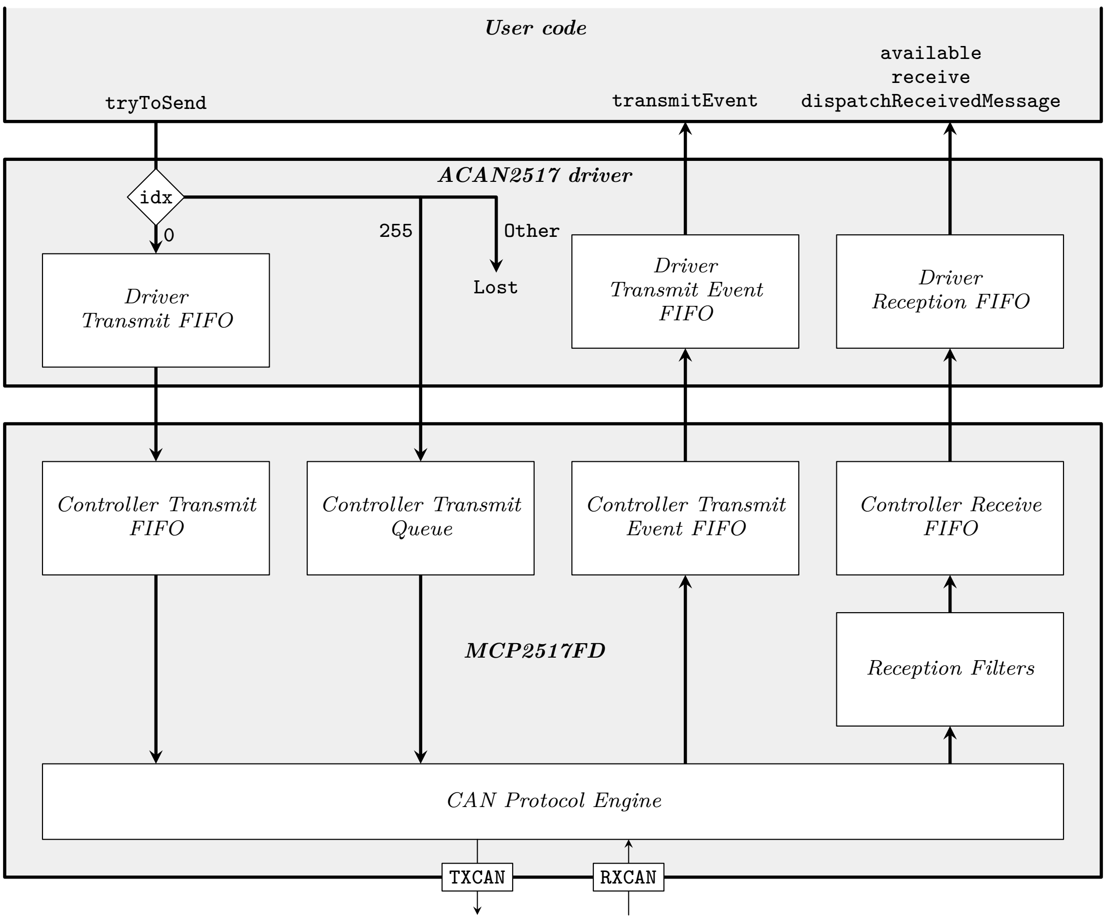

Figure 2 – Message flow in `ACAN2517` driver and `MCP2517FD` CAN Controller, custom configuration

You can allocate the *Controller transmit Queue*: send order is defined by frame priority (see section 9 page 16).

You can also define up to 32 receive filters (see section 14 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 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>
```

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 $\overline{\texttt{CS}}$ and $\overline{\texttt{INT}}$ 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 $\overline{\texttt{CS}}$ 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 6.2 page 9), selecting alternate pins should be done before calling SPI.begin, on Adafruit Feather M0 (section 6.3 page 11), 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 17.7 page 33 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 14 page 21. The second argument is the *interrupt service routine*, and is defined by a C++ lambda expression[1]. See section 16.2 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 16.3 page 25.

```
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 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 `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 ;  // 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} ;
  } ;
} ;
```

[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.

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 15 page 23);
- on sending messages, it is used for selecting the transmit buffer (see section 12 page 18).

6 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 $\overline{\mathtt{CS}}$ signal requires one digital pin, that the driver configures as an `OUTPUT` ;
- connecting the $\overline{\mathtt{INT}}$ 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 $\overline{\mathtt{INT0}}$ and $\overline{\mathtt{INT1}}$ signals are not used by driver and are left not connected.

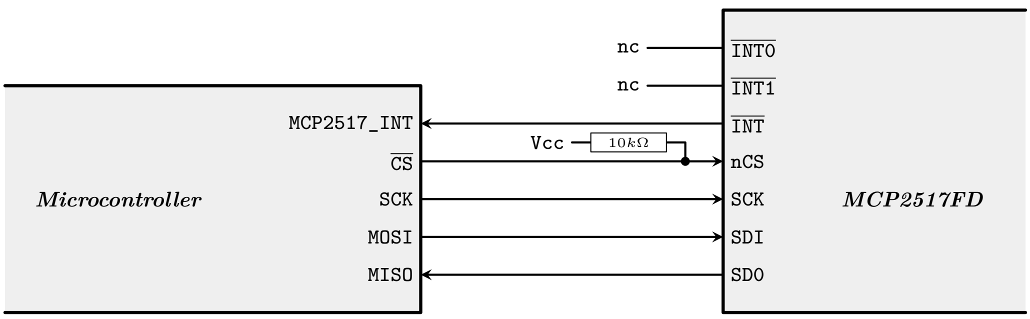

Figure 3 – `MCP2517FD` connection to a microcontroller

6.1 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 $\overline{\mathtt{CS}}$ 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.

6.2 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]:

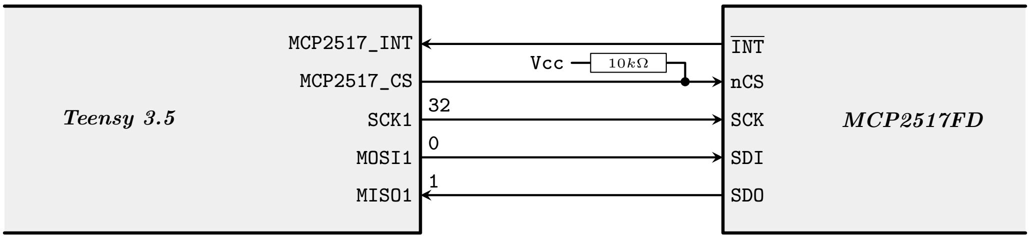

Figure 4 – Using SPI alternate pins on a Teensy 3.5

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:

```
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 () ;
```

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

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

6.3 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.

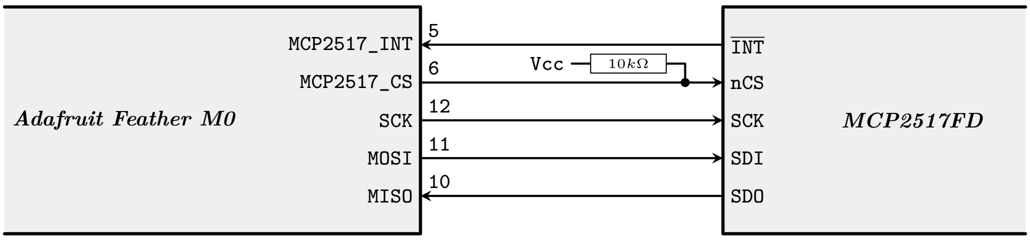

Figure 5 – Using SPI alternate pins on an Adafruit Feather M0

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

6.4 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 1. Note that ACAN2517 does not use hardware CS.

Port	SCK	MOSI	MISO	CS, not used by ACAN2517
VSPI	IO18	IO23	IO19	IO5
HSPI	IO14	IO13	IO12	IO15

Table 1 – ESP32 SPI default pins

6.4.1 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 17.7.1 page 33).

6.4.2 Using SPI

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

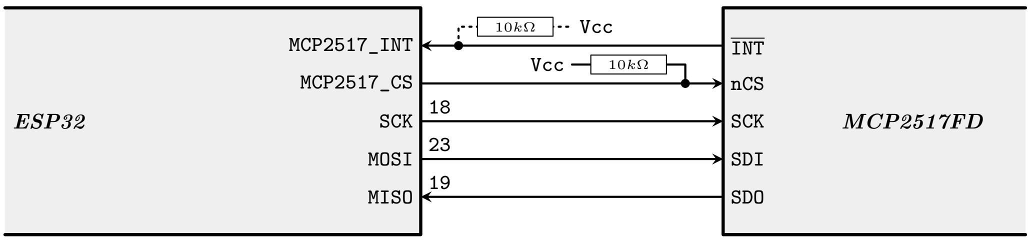

Figure 6 – Using VSPI default pins on an ESP32

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
```

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.

6.4.3 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`.

6.5 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.

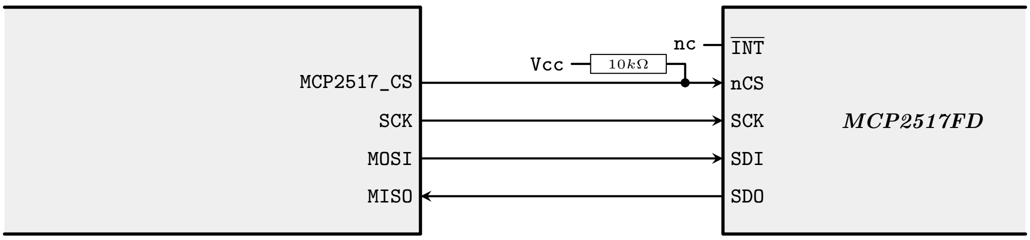

Figure 7 – Connection with no interrupt pin

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;
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 Clock configuration

The `MCP251xFD` Oscillator Block Diagram is given in figure 8. 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]

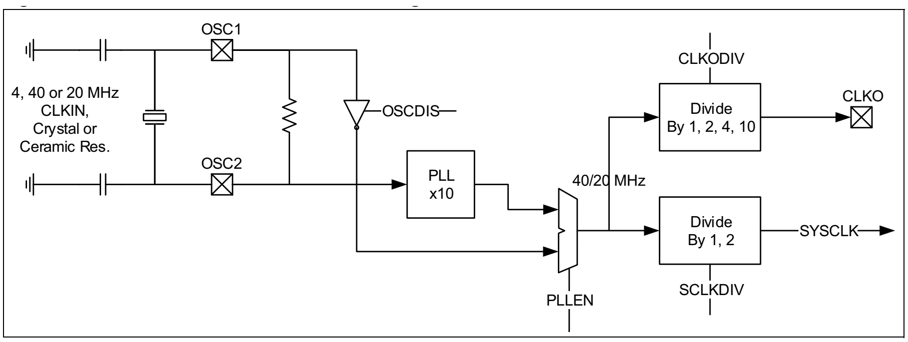

Figure 8 – `MCP251xFD` Oscillator Block Diagram (`DS20005678B`, figure 3.1 page 13)

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,
    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 2. 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, for resetting the `MCP2517FD` and programming the `PLLEN` and `SCLKDIV` bits. Then SPI clock is set to a frequency equal to

[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 `ACAN2517` oscillator selection

`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`.

8 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 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 11 page 17).

8.1 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 () ;
```

8.2 The driverTransmitBufferCount method

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

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

8.3 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.

9 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 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 11 page 17).

[7] DS20005678B, section 4.5, page 28.

10 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.

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).

11 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;
- 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)
```

[8] DS20005688B, section 3.3, page 63.

[9] 16 bytes because the `MCP2517FD` is in the CAN 2.0B mode, otherwise a CANFD message object can require up to 72 bytes.

```
  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].

12 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 8 page 15) ;
- 255 for the transmit Queue (section 9 page 16).

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:

```
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.

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

```
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 ;
    }
  }
  ...
}
```

13 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 15 page 23).

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.

13.1 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`.

13.2 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 () ;
```

13.3 The `receiveBufferCount` method

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

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

13.4 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`.

14 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) ;
...
```

14.1 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 15 page 23).

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:

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

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

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

```
//---------------------------------- 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`).

14.2 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.

14.3 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.

14.4 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.

14.5 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.

15 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.

16 The `ACAN2517::begin` method reference

16.1 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 16.2 page 25). 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.

16.2 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) ;
```

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

Bit	Static constant Name	Link
0	kRequestedConfigurationModeTimeOut	section 16.3.1 page 25
1	kReadBackErrorWith1MHzSPIClock	section 16.3.2 page 25
2	kTooFarFromDesiredBitRate	section 16.3.3 page 26
3	kInconsistentBitRateSettings	section 16.3.4 page 26
4	kINTPinIsNotAnInterrupt	section 16.3.5 page 26
5	kISRIsNull	section 16.3.6 page 26
6	kFilterDefinitionError	section 16.3.7 page 26
7	kMoreThan32Filters	section 16.3.8 page 26
8	kControllerReceiveFIFOSizeIsZero	section 16.3.9 page 26
9	kControllerReceiveFIFOSizeGreaterThan32	section 16.3.10 page 26
10	kControllerTransmitFIFOSizeIsZero	section 16.3.11 page 27
11	kControllerTransmitFIFOSizeGreaterThan32	section 16.3.12 page 27
12	kControllerRamUsageGreaterThan2048	section 16.3.13 page 27
13	kControllerTXQPriorityGreaterThan31	section 16.3.14 page 27
14	kControllerTransmitFIFOPriorityGreaterThan31	section 16.3.15 page 27
15	kControllerTXQSizeGreaterThan32	section 16.3.16 page 27
16	kRequestedModeTimeOut	section 16.3.17 page 27
17	kX10PLLNotReadyWithin1MS	section 16.3.18 page 27
18	kReadBackErrorWithFullSpeedSPIClock	section 16.3.19 page 27
19	kISRNotNullAndNoIntPin	section 16.3.20 page 28

Table 3 – The `ACAN2517::begin` method error code bits

16.3.1 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 `LCP2517FD` does not reach the configuration mode with 2ms. It means that the `MCP2517FD` cannot be accessed via SPI.

16.3.2 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.

16.3.3 `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
}
```

16.3.4 `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 17.2 page 31.

16.3.5 `kINTPinIsNotAnInterrupt`

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

16.3.6 `kISRIsNull`

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

16.3.7 `kFilterDefinitionError`

`settings.filterStatus()` returns a value different than `ACAN2517Filters::kFiltersOk`, meaning that one or more filters are erroneous. See section 14.1 page 21.

16.3.8 `kMoreThan32Filters`

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

16.3.9 `kControllerReceiveFIFOSizeIsZero`

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

16.3.10 `kControllerReceiveFIFOSizeGreaterThan32`

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

16.3.11 kControllerTransmitFIFOSizeIsZero

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

16.3.12 kControllerTransmitFIFOSizeGreaterThan32

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

16.3.13 kControllerRamUsageGreaterThan2048

The configuration you have defined requires more than 2048 bytes of `MCP2517FD` internal RAM. See section 11 page 17.

16.3.14 kControllerTXQPriorityGreaterThan31

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

16.3.15 kControllerTransmitFIFOPriorityGreaterThan31

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

16.3.16 kControllerTXQSizeGreaterThan32

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

16.3.17 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.

16.3.18 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.

16.3.19 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.

16.3.20 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.

17 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.

17.1 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 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 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:

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

17.2 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 4.

The `ACAN2517Settings` class defines static constant properties that can be used as mask error. For example:

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

17.3 The actualBitRate method

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

```
void setup () {
  ...
```

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 4 – The `ACAN2517Settings::CANBitSettingConsistency` method error codes

```
  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 17.2 page 31), the returned value is irrelevant.

17.4 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 17.2 page 31), 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.

17.5 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 ppm $= 10^{-6}$. In other words, $10,000$ 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 17.2 page 31), the returned value is irrelevant.

17.6 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 17.2 page 31), the returned value is irrelevant.

17.7 Properties of the `ACAN2517Settings` class

All properties of the `ACAN2517Settings` class are declared `public` and are initialized (table 5). 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.

17.7.1 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.

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 17.1 page 28
`mPhaseSegment1`	`uint16_t`	`0`	See section 17.1 page 28
`mPhaseSegment2`	`uint8_t`	`0`	See section 17.1 page 28
`mSJW`	`uint8_t`	`0`	See section 17.1 page 28
`mBitRateClosedToDesiredRate`	`bool`	`false`	See section 17.1 page 28
`mTXCANIsOpenDrain`	`bool`	`false`	See section 17.7.1 page 33
`mCLKOPin`	`CLKOpin`	`CLKO_DIVIDED_BY_10`	See section 17.7.2 page 34
`mRequestedMode`	`RequestedMode`	`Normal20B`	See section 17.7.3 page 35
`mDriverTransmitFIFOSize`	`uint16_t`	`16`	See section 8 page 15
`mControllerTransmitFIFOSize`	`uint8_t`	`32`	See section 8 page 15
`mControllerTransmitFIFOPriority`	`uint8_t`	`0`	See section 8 page 15
`mControllerTransmitFIFO-RetransmissionAttempts`	`RetransmissionAttempts`	`UnlimitedNumber`	See section 8 page 15
`mControllerTXQSize`	`uint8_t`	`0`	See section 9 page 16
`mControllerTXQBufferPriority`	`uint8_t`	`31`	See section 9 page 16
`mControllerTXQBuffer-RetransmissionAttempts`	`RetransmissionAttempts`	`UnlimitedNumber`	See section 9 page 16
`mDriverReceiveFIFOSize`	`uint16_t`	`32`	See section 10 page 17
`mControllerReceiveFIFOSize`	`uint8_t`	`32`	See section 10 page 17

Table 5 – Properties of the `ACAN2517Settings` class

17.7.2 The `CLKO/SOF` pin

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

- output *internally generated clock*;
- 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:

[12] *Internally generated clock* is not `SYSCLK`, see figure 8 page 14.

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

17.7.3 The `mRequestedMode` property

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