ACAN2517FD Arduino library
For the MCP2517FD and MCP2518FD
CANFD Controllers in CANFD mode
Version 2.1.6

Pierre Molinaro

April 21, 2021

Contents

1 Versions 5

2 Features 6

3 MCP2517FD or MCP2518FD? 7

3.1 Reset . . . 7
3.2 Clock . . . 7
3.3 Restricted Operation Mode . . . 8

4 Data flow 9

4.1 Data flow in default configuration . . . 9
4.2 Data flow, custom configuration . . . 10

5 A simple example: LoopBackDemo 10

6 The CANFDMessage class 13

6.1 Properties . . . 14
6.2 The default constructor . . . 14
6.3 Constructor from CANMessage . . . 15
6.4 The type property . . . 15
6.5 The len property . . . 16
6.6 The idx property . . . 16
6.7 The pad method . . . 16
6.8 The isValid method . . . 16

7 The CANMessage class 17

8 Connecting a MCP2517FD to your microcontroller 17
8.1 Pullup resistor on nCS pin . . . 18
8.2 Using alternate pins on Teensy 3.x . . . 18
8.3 Using alternate pins on an Adafruit Feather M0 . . . 20
8.4 Connecting to an ESP32 . . . 21
8.4.1 Connecting MCP2517_CS and MCP2517_INT . . . 21
8.4.2 Using SPI . . . 21
8.4.3 Using HSPI . . . 22
8.5 Connection with no interrupt pin . . . 22
8.6 Wiring schemes . . . 23
8.6.1 Arduino Uno - MCP2518FDClick . . . 23

9 Clock configuration 23

10 Transmit FIFO 25
10.1 The driverTransmitBufferSize method . . . 26
10.2 The driverTransmitBufferCount method . . . 26
10.3 The driverTransmitBufferPeakCount method . . . 26

11 Transmit Queue (TXQ) 27

12 Receive FIFO 27

13 Payload size 28
13.1 The ACAN2517FDSettings::objectSizeForPayload static method . . . 29

14 RAM usage 29

15 Sending frames: the tryToSend method 30
15.1 Calling tryToSend with an CANMessage argument . . . 31
15.2 Usage example . . . 31

16 Retrieving received messages using the receive method 32
16.1 Driver receive buffer size . . . 33
16.2 The receiveBufferSize method . . . 34
16.3 The receiveBufferCount method . . . 34
16.4 The receiveBufferPeakCount method . . . 34

17 Acceptance filters 34
17.1 An example . . . 35
17.2 The appendPassAllFilter method . . . 36
17.3 The appendFormatFilter method . . . 36
17.4 The appendFrameFilter method . . . 36
17.5 The appendFilter method . . . 37

18 The dispatchReceivedMessage method **37**

19 The ACAN2517FD::begin method reference **38**
19.1 The prototypes . . . 38
19.2 Defining explicitly the interrupt service routine . . . 39
19.3 The error code . . . 39
19.3.1 kRequestedConfigurationModeTimeOut . . . 39
19.3.2 kReadBackErrorWith1MHzSPIClock . . . 39
19.3.3 kTooFarFromDesiredBitRate . . . 40
19.3.4 kInconsistentBitRateSettings . . . 40
19.3.5 kINTPinIsNotAnInterrupt . . . 41
19.3.6 kISRIsNull . . . 41
19.3.7 kFilterDefinitionError . . . 41
19.3.8 kMoreThan32Filters . . . 41
19.3.9 kControllerReceiveFIFOSizeIsZero . . . 41
19.3.10 kControllerReceiveFIFOSizeGreaterThan32 . . . 41
19.3.11 kControllerTransmitFIFOSizeIsZero . . . 41
19.3.12 kControllerTransmitFIFOSizeGreaterThan32 . . . 41
19.3.13 kControllerRamUsageGreaterThan2048 . . . 41
19.3.14 kControllerTXQPriorityGreaterThan31 . . . 42
19.3.15 kControllerTransmitFIFOPriorityGreaterThan31 . . . 42
19.3.16 kControllerTXQSizeGreaterThan32 . . . 42
19.3.17 kRequestedModeTimeOut . . . 42
19.3.18 kX10PLLNotReadyWithin1MS . . . 42
19.3.19 kReadBackErrorWithFullSpeedSPIClock . . . 42
19.3.20 kISRNotNullAndNoIntPin . . . 42
19.3.21 kInvalidTDCO . . . 43

20 ACAN2517FDSettings class reference **43**
20.1 The ACAN2517FDSettings constructor: computation of the CAN bit settings . . . 43
20.2 The CANBitSettingConsistency method . . . 49
20.3 The kArbitrationTQCountNotDivisibleByDataBitRateFactor error . . . 50
20.4 The actualArbitrationBitRate method . . . 50
20.5 The exactArbitrationBitRate method . . . 51
20.6 The exactDataBitRate method . . . 51
20.7 The ppmFromDesiredArbitrationBitRate method . . . 51
20.8 The ppmFromDesiredDataBitRate method . . . 51
20.9 The arbitrationSamplePointFromBitStart method . . . 52
20.10 The dataSamplePointFromBitStart method . . . 52
20.11 Properties of the ACAN2517FDSettings class . . . 52
20.11.1 The mTXCANIsOpenDrain property . . . 52
20.11.2 The mINTIsOpenDrain property . . . 52
20.11.3 The CLKO/SOF pin . . . 52
20.11.4 The mRequestedMode property . . . 54

20.11.5 The mISOCRCEnabled property 54
20.11.6 The mTDCO property 54

21 Other ACAN2517FD methods **54**
21.1 The currentOperationMode method 54
21.2 The recoverFromRestrictedOperationMode method 55
21.3 The errorCounters method 55
21.4 The diagInfos method 55

22 The sendfd-odd and sendfd-even sketches **56**

1 Versions

Version	Date	Comment
2.1.6	April 21, 2021	Added x9 and x10 data bit rate factors (thanks to Pedro Dionisio Pereira Junior). Added Arduino Uno – MCP2518FDClick wiring scheme (thanks to soso49).
2.1.5	January 27, 2021	Fixed retransmission attempts setting bug. Added NoRetransmissionAttemptsDemoTeensy3x.ino sketch.
2.1.4	January 14, 2021	Improved method to read also the BDIAG0_REGISTER diagnostic register (thanks to turmary), see section 21.4 page 55. Fix: mHardwareTxFIFOFull = true will block the transmitter if call begin() multiple times without constructor (thanks to turmary).
2.1.3	October 3, 2020	Add method to read the diagnostic registers (thanks to Flole998), see section 21.4 page 55.
2.1.2	May 31, 2020	Fix retransmission attempts settings (thanks to Flole998)
2.1.1	April 27, 2020	Added dataFloat to CANMessage and CANFDMessage (thanks to Koryphon)
2.1.0	December 31, 2019	For compatibility with ACAN_T4, the DataBitRateFactor enumeration is declared outside of the ACAN2517FDSettings class. Fix commented out line (thank to Flole998).
2.0.1	October 28, 2019	Fix incorrect usage of digitalPinToInterrupt (thank to Flole998).
2.0.0	September 15, 2019	Fixed several bugs. Added ACAN2517FD::currentOperationMode method, see section 21.1 page 54. Added ACAN2517FD::recoverFromRestrictedOperationMode method, see section 21.2 page 55. Added ACAN2517FD::errorCounters method, see section 21.3 page 55. Added description of sendfd-odd and sendfd-even sketches, see section 22 page 56. Added section *MCP2517FD or MCP2518FD?* page 7.
1.1.6	June 6, 2019	Running pinMode (mINT, INPUT_PULLUP) only if mInt pin is used (thanks to Tyler Lewis).
1.1.5	June 2, 2019	Fixed a race condition on ESP32 (thanks to Nick Kirkby).
1.1.4	March 21, 2019	Fixed dual bit rate bug (thanks to danielhenz). Fixed TxQ enable bug (thanks to danielhenz). Added setting of *Enable Edge Filtering during Bus Integration state bit*, for reaching the 8 Mbit/s bit data rate. Updated LoopBackIntensiveTestTeensy3x sample code.
1.1.3	February 8, 2019	Compatibility for Arduino Uno.

		Added demo sketch LoopBackDemoArduinoUno.
		Renamed ACANBuffer to ACANFDBuffer.
1.1.2	February 3, 2019	Added setting mINTIsOpenDrain (section 20.11.2 page 52).
		Remove useless mutex (ESP32).
1.1.1	January 31, 2019	First release running on ESP32 (section 8.4 page 21).
		New option: no interrupt pin (section 8.5 page 22).
1.0.4	January 14, 2019	Fixed mask and acceptance filters for extended messages.
		New LoopBackDemoTeensy3xStandardFilterTest.ino sample code for checking base reception filters.
		New LoopBackDemoTeensy3xExtendedFilterTest.ino sample code for checking extended reception filters.
1.0.3	January 6, 2019	Corrected identifiers for extended messages.
1.0.2	November 2, 2018	added mISOCRCEnabled setting
1.0.1	October 29, 2018	Conformity with Arduino library
1.0.0	October 28, 2018	Initial release

2 Features

The ACAN2517FD library is a MCP2517FD and MCP2518FD CANFD (*Controller Area Network with Flexible Data*) Controller driver for any board running Arduino. It handles CANFD 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 MCP2517FD or MCP2518FD?

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

3.1 Reset

An originality of the MCP2517FD is that it has no reset pin. Resetting the MCP2517FD can only be done by software, by sending a RESET command through the SPI. But sometimes, for reasons I don't know, the reset is not done correctly.We can see this because the value returned by the ACAN2517FD::begin function is not zero (see section 19.3 page 39). Some possible errors are 0x1 (kRequestedConfigurationModeTimeOut, the MCP2517FD cannot reach the *configuration* mode), 0x40000 (kReadBackErrorWithFullSpeedSPIClock, the MCP2517FD RAM cannot be written and read back). Typically, this can happen when uploading and starting a new version of the firmware into the microcontroller. **So I recommend to always check the value returned by the ACAN2517FD::begin function is zero.** In such case, you should power off and the power on.

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

Note you should also add a pullup resistor on the nCS pin (section 8.1 page 18) with a MCP2517FD, I don't think this resistance is necessary with a MCP2518FD.

3.2 Clock

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

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

4MHz crystal oscillator. I have tried a 4MHz crystal oscillator (HC49US-FF3F18-4.0000), with two 22pF capacitors, so the clock setting is ACAN2517FDSettings::OSC_4MHz10xPLL. I noticed that a MCP2517FD worked well for a data bit rate up to 1Mbps; above 1Mbps, the MCP2517FD often enters in *Restricted Operation*

Mode, but maybe it's due to internal bugs (see section 3.3 page 8). A MCP2518FD works prefectly with this oscillator.

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

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

4MHz integrated oscillator. I use a 4MHz integrated oscillator (LFSPXO024978BULK, the supply voltage of my MCP2517FD is 3.3V), connected to OSC1. OSC2 is left unconnected. The clock setting is ACAN2517FDSettings::OSC_4MHz1 I have observed with oscilloscope the OSC1 pin signal, it has the correct frequency, and the amplitude I expected: 3.3V.

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

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

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

3.3 Restricted Operation Mode

For testing transmission and reception, I use the sendfd-odd and sendfd-even sketches, that are provided as sample code in the library (see section 22 page 56). They are designed for a *Teensy 3.5*, but can easily be adapted for other platforms.

For data bit rates higher than 1Mbps with a MCP2517FD, I have noticed the error counters may have not zero values (error counters can be read by the errorCounters method, see section 21.3 page 55), and the MCP2517FD enters sometimes in *Restricted Operation Mode*. The modes operation is described in DS20005678D, figure 2.1 page 9. *Restricted Operation Mode* is reached from *Normal Modes* on *System Error*, as the driver lets the SERR2LOM bit equal to 0.

System Error is described in section 10.5.6, page 63. The MCP2517FD Data Sheet Errata (DS80000792B) gives an explanation: *The SPI Interface can block the CANFD Controller module from accessing RAM in between SPI bytes and between the last byte and the rising edge of the nCS line during an SPI READ or SPI READ CRC instruction while accessing RAM. If the CANFD Controller module is blocked for more than TSPIMAXDLY, a TX MAB underflow or an RX MAB overflow can occur.* Within the CANFD Control Field, TSPIMAXDLY is 3 NBT + 5 DBT, that is for an 1Mbps arbitration bit rate and a data bit factor x8 (8Mbps) : $3 \cdot 1\mu s + 5 \cdot 125ns = 3.625\mu s$. The challenge is to write a driver that checks these constraints. This is not easy, as transfers are made through transfer and transfer16 SPI Arduino routines, and their implementation may vary from one platform to another. In the ACAN2517FD code, I have masked interruptions during transfers to minimize the delay between bytes, and to ensure that the nCS signal becomes inactive (high) as quickly as possible at the end of the transfer.

You can check current MCP2517FD operation mode by calling the ACAN2517FD::currentOperationMode function (section 21.1 page 54. It returns 7 for the *Restricted Operation Mode*. You can recover from *Restricted Operation Mode* by calling the ACAN2517FD::recoverFromRestrictedOperationMode function (section 21.2 page 55); however, some send or receive data has been lost.

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

4 Data flow

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

4.1 Data flow in default configuration

The figure 1 illustrates message flow in the default configuration.

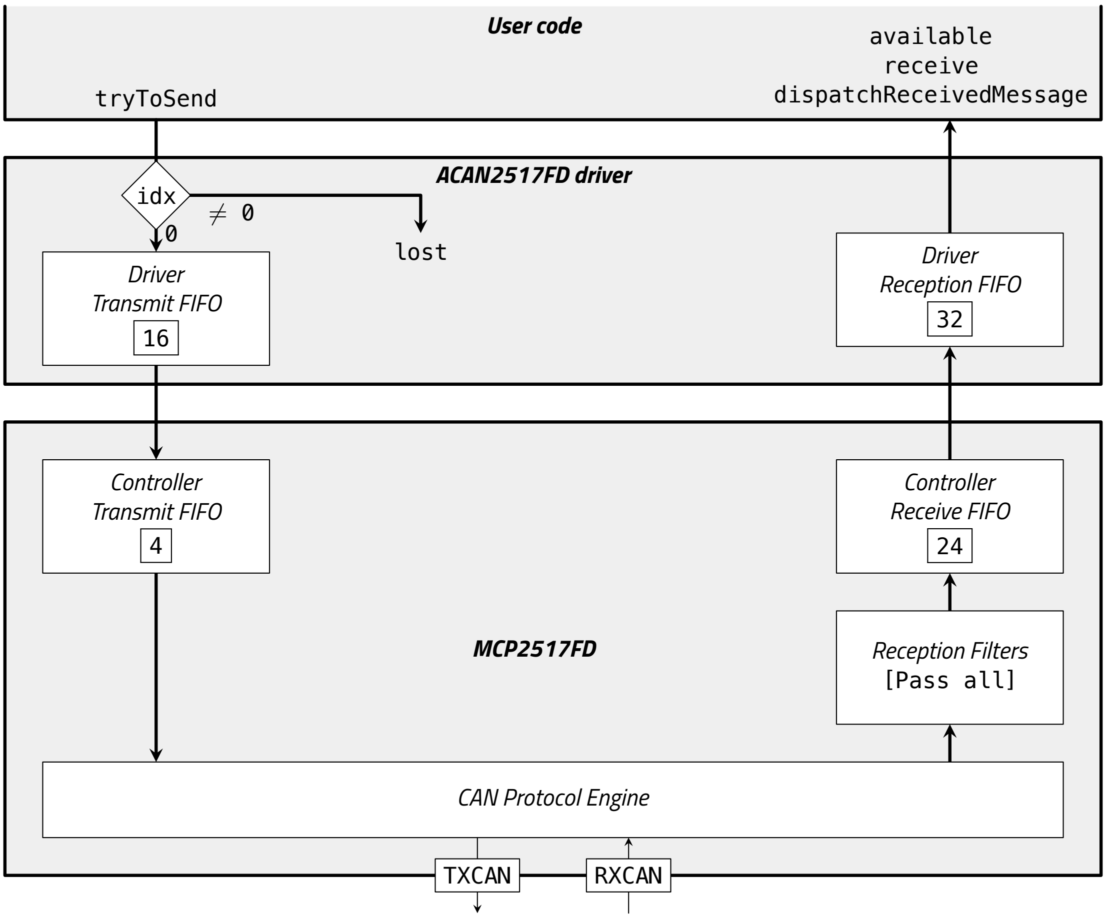

Figure 1 – Message flow in ACAN2517FD driver and MCP2517FD CAN Controller, default configuration

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 capacicity of 2048 bytes, that limits the sizes of the *controller transmit FIFO* and *controller receive FIFO*. See section 14 page 29.

A message is defined by an instance of `CANFDMessage` class. For sending a message, user code calls the `tryToSend` method – see section 15 page 30, 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 17 page 34 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 16.1 page 33 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 16 page 32;
- the `dispatchReceivedMessage` method if you have defined the reception filters that name a call-back function – see section 18 page 37.

4.2 Data flow, custom configuration

The figure 2 illustrates message flow in a custom configuration.

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

You can allocate the *Controller transmit Queue*: send order is defined by frame priority (see section 11 page 27). You can also define up to 32 receive filters (see section 17 page 34). Sizes of `MCP2517FD` internal buffer are easily customizable.

5 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 $\overline{\text{CS}}$ and $\overline{\text{INT}}$ pins (adapt to your design).

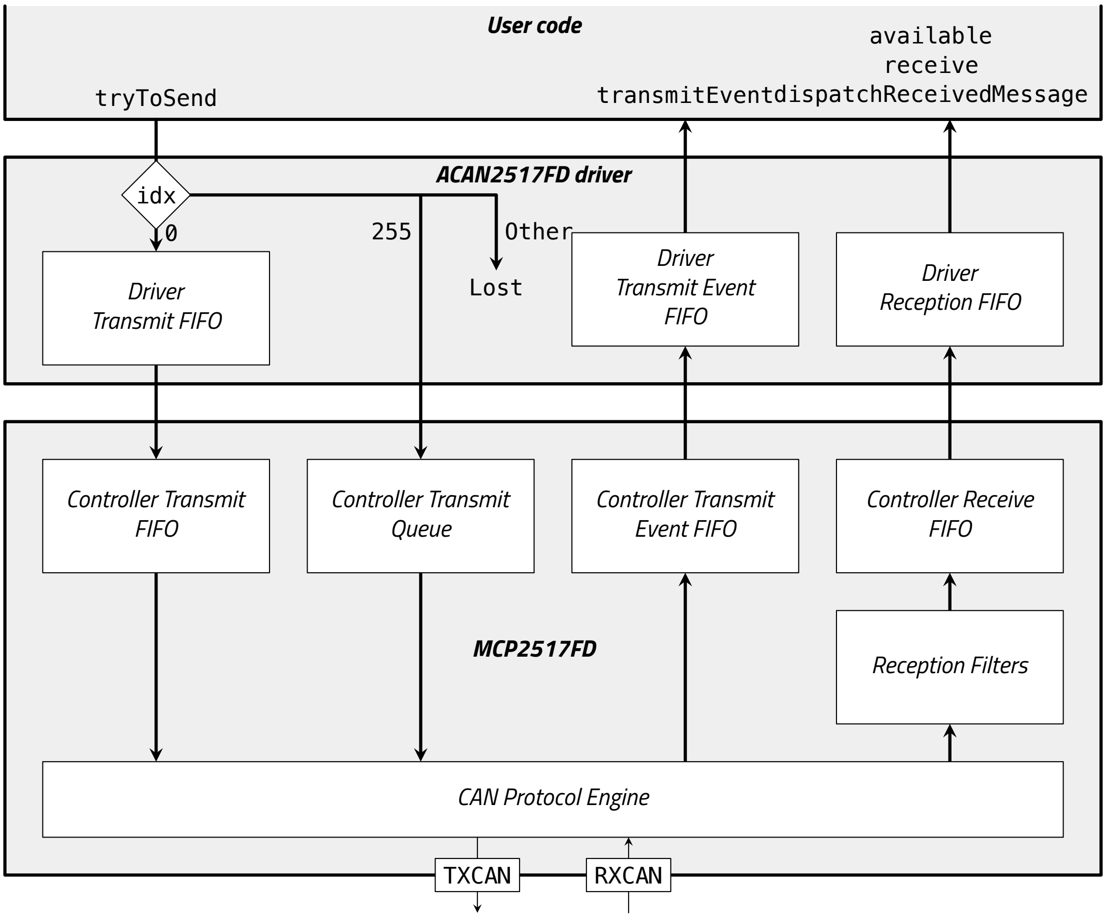

Figure 2 – Message flow in ACAN2517FD driver and MCP2517FD CAN Controller, custom configuration

```
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 $\overline{\text{CS}}$ pin, the SPI object (you can use SPI1, SPI2, ...), the number of the pin connected to the $\overline{\text{INT}}$ pin.

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

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

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

You should call SPI.begin. Many platforms define alternate pins for SPI. On Teensy 3.x (section 8.2 page 18), selecting alternate pins should be done before calling SPI.begin, on Adafruit Feather M0 (section 8.3 page 20), this should be done after. Calling SPI.begin explicitly allows you to fully handle alternate pins.

```
ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                             125UL * 1000UL, DataBitRateFactor::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.

Note. For releases before 2.1.0, the data bit rate enumerated type was declared within the ACAN2517FDSettings class, so the declaration was ACAN2517FDSettings::DATA_BITRATE_x4. In release 2.1.0 and above, the DataBitRateFactor enumerated type is declared outside any class, enabling its compatibility with other CANFD librairies, as ACAN_T4.

```
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 20.11 page 52 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 (base / extended, data / remote, CAN / CANFD) frame, and to receive all (base / extended, data / remote, CAN / CANFD) frames. If you want to define reception filters, see section 17 page 34. The second argument is the *interrupt service routine*, and is defined by a C++ lambda expression[1]. See section 19.2 page 39 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 19.3 page 39.

```
static uint32_t gBlinkLedDate = 0 ;
static uint32_t gReceivedFrameCount = 0 ;
static uint32_t gSentFrameCount = 0 ;
```

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

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 base data frame, with an identifier equal to 0, and without any data – see section 6 page 13.

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

We try to send the data message. Actually, we try to transfer it into the *Driver transmit buffer*. The transfer succeeds if the buffer is not full. The `tryToSend` method returns `false` if the buffer is full, and `true` otherwise. Note the returned value only tells if the transfer into the *Driver transmit buffer* is successful or not: we have no way to know if the frame is actually sent on the the CAN network. Then, we act the successfull transfer by setting `gSendDate` to the next send date and incrementing the `gSentCount` variable. Note if the transfer did fail, the send date is not changed, so the `tryToSend` method will be called on the execution of the `loop` function.

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

As the `MCP2517FD` controller is configured in *loop back* mode, all sent messages are received. The `receive` method returns `false` if no message is available from the *driver reception buffer*. It returns `true` if a message has been successfully removed from the *driver reception buffer*. This message is assigned to the `message` object. If a message has been received, the `gReceivedCount` is incremented ans displayed.

6 The CANFDMessage class

Note. The `CANFDMessage` class did change in release 2.0.0: the `rtr` property has been removed, the `type` property has been added.

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 CANFD message is an object that contains all CANFD frame user informations.

Example: The `message` object describes an extended frame, with identifier equal to `0x123`, that contains 12 bytes of data:

```
CANFDMessage message ; // message is fully initialized with default values
message.id = 0x123 ; // Set the message identifier (it is 0 by default)
message.ext = true ;  // message is an extended one (it is a base one by default)
message.len = 12 ; // message contains 12 bytes (0 by default)
message.data [0] = 0x12 ; // First data byte is 0x12
...
message.data [11] = 0xCD ; // 11th data byte is 0xCD
```

6.1 Properties

```
class CANFDMessage {
  ...
  public : uint32_t id;  // Frame identifier
  public : bool ext ; // false -> base frame, true -> extended frame
  public : Type type ;
  public : uint8_t idx ;  // Used by the driver
  public : uint8_t len ;  // 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
    float    dataFloat [16] ; // 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, 8 x 64-bit or 16 x 32-bit floats. Be aware that multi-byte integers are subject to endianness (Cortex M4 processors of Teensy 3.x are little-endian).

6.2 The default constructor

All properties are initialized by default, and represent a base data frame, with an identifier equal to 0, and without any data (table 2).

Property	Initial value	Comment
id	0	
ext	false	Base frame
type	CANFD_WITH_BIT_RATE_SWITCH	CANFD frame, with bit rate switch
idx	0	
len	0	No data
data	–	*unitialized*

Table 2 – CANFDMessage default constructor initialization

6.3 Constructor from CANMessage

```
class CANFDMessage {
...
CANFDMessage (const CANMessage & inCANMessage) ;
...
} ;
```

All properties are initialized from the inCANMessage (table 3). Note that only data64[0] is initialized from inCANMessage.data64.

Property	Initial value
id	inCANMessage.id
ext	inCANMessage.ext
type	inCANMessage.rtr ? CAN_REMOTE : CAN_DATA
idx	inCANMessage.idx
len	inCANMessage.len
data64[0]	inCANMessage.data64

Table 3 – CANFDMessage constructor CANMessage

6.4 The type property

The type property has been added in release 2.0.0. Its value is an instance of an enumerated type:

```
class CANFDMessage {
...
public: typedef enum : uint8_t {
  CAN_REMOTE,
  CAN_DATA,
  CANFD_NO_BIT_RATE_SWITCH,
  CANFD_WITH_BIT_RATE_SWITCH
} Type ;
...
} ;
```

The type property specifies the frame format, as indicated in the table 4.

type property	Meaning	Constraint on len
CAN_REMOTE	CAN 2.0B remote frame	0 ... 8
CAN_DATA	CAN 2.0B data frame	0 ... 8
CANFD_NO_BIT_RATE_SWITCH	CANFD frame, no bit rate switch	0 ... 8, 12, 16, 20, 24, 32, 48, 64
CANFD_WITH_BIT_RATE_SWITCH	CANFD frame, bit rate switch	0 ... 8, 12, 16, 20, 24, 32, 48, 64

Table 4 – CANFDMessage type property

6.5 The len property

Note that len property contains the actual length, not its encoding in CANFD 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 section 6.7 page 16) for padding with 0x00 bytes to the next valid length.

6.6 The idx property

The idx property is not used in CANFD frames, but:

- for a received message, it contains the acceptance filter index (see section 18 page 37);
- on sending messages, it is used for selecting the transmit buffer (see section 15 page 30).

6.7 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
```

6.8 The isValid method

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

Not all settings of CANFDMessage instances represent a valid frame. For example, there is no CANFD remote frame, so a remote frame should have its length lower than or equal to 8. There is no constraint on extended / base identifier (ext property).

The isValid returns true if the contraints on the len property are checked, as indicated the table 4 page 16, and false otherwise.

7 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 and the ACAN2517[4] driver contain an identical CANMessage.h file header, enabling using ACAN driver, ACAN2515 driver, ACAN2517 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 base data frame, with an identifier equal to 0, and without any data. In the ACAN2517FD library, the CANMessage class is only used by a CANFDMessage constructor (section 6.3 page 15).

```
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 ACAN 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
    float    dataFloat [2] ; // 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, one 64-bit or two 32-bit floats. Be aware that multi-byte integers and floats are subject to endianness (Cortex M4 processors of Teensy 3.x are little-endian).

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

- for a received message, it contains the acceptance filter index (see section 18 page 37);
- on sending messages, it is used for selecting the transmit buffer (see section 15 page 30).

8 Connecting a MCP2517FD to your microcontroller

Connecting a MCP2517FD requires 5 pins (figure 3):

[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.
[4] The ACAN2517 driver is a CAN driver for the MCP2517FD CAN controller in CAN 2.0B mode, https://github.com/pierremolinaro/acan2517.

- hardware SPI requires you use dedicaced pins of your microcontroller. You can use alternate pins (see below), and if your microcontroller supports several hardware SPIs, you can select any of them;
- connecting the $\overline{\text{CS}}$ signal requires one digital pin, that the driver configures as an `OUTPUT` ;
- connecting the $\overline{\text{INT}}$ signal requires one other digital pin, that the driver configures as an external interrupt input; so this pin should have interrupt capability (checked by the `begin` method of the driver object);
- the $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ signals are not used by driver and are left not connected.

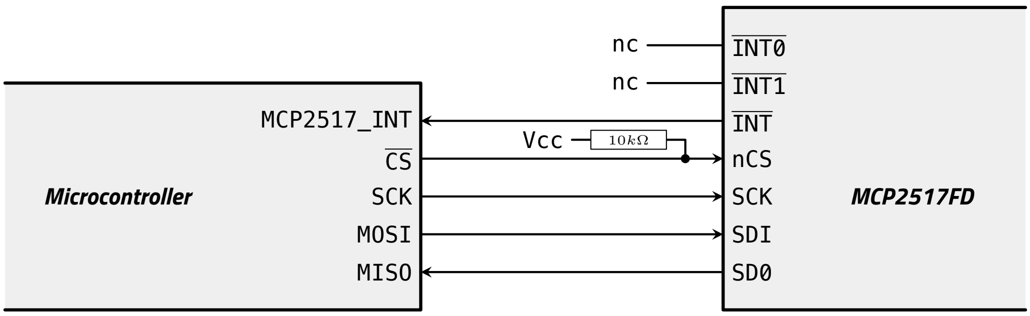

Figure 3 – MCP2517FD connection to a microcontroller

8.1 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 $\overline{\text{CS}}$ 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.

However, I noticed that the MCP2518FD reset properly even without any pullup resistor.

8.2 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.*"[5]

For example, the `LoopBackDemoTeensy3x` sketch uses `SPI1` on a Teensy 3.5 with these alternate pins[6]:

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

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

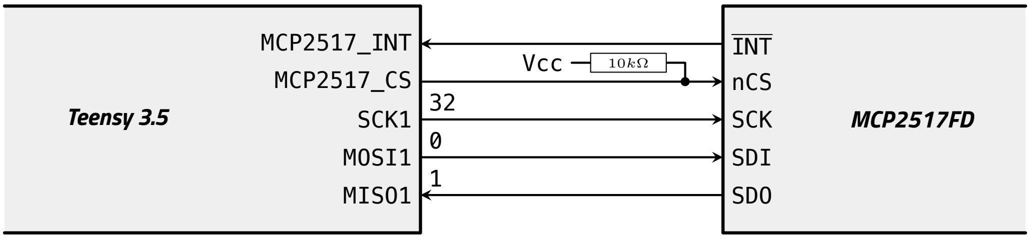

Figure 4 – Using SPI alternate pins on a Teensy 3.5

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

```
...
```

8.3 Using alternate pins on an Adafruit Feather M0

Demo sketch: `LoopBackDemoAdafruitFeatherM0`.

Link: https://learn.adafruit.com/using-atsamd21-sercom-to-add-more-spi-i2c-serial-ports/overview

This document 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.

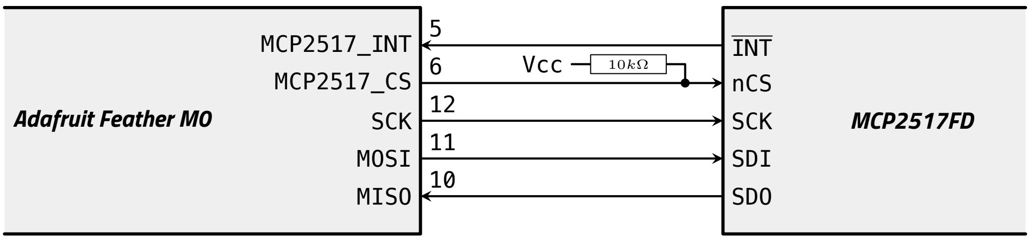

Figure 5 – Using SPI alternate pins on an Adafruit Feather M0

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

8.4 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 5.

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

Table 5 – ESP32 SPI default pins

8.4.1 Connecting MCP2517_CS and MCP2517_INT

For MCP2517_CS, you can use any port that can be configured as digital output. ACAN2517FD 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 20.11.2 page 52).

8.4.2 Using SPI

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

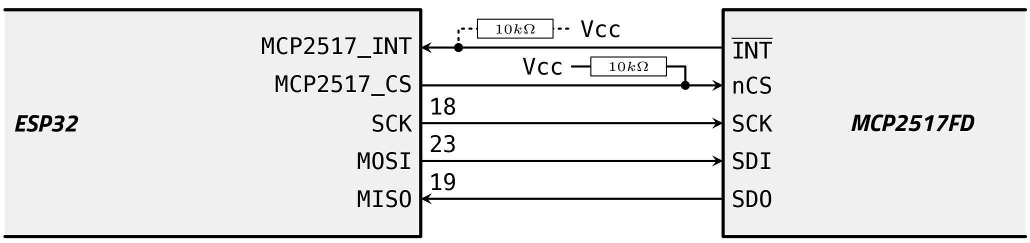

Figure 6 – Using VSPI default pins on an ESP32

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 ACAN2517FD.

8.4.3 Using HSPI

The ESP32 demo sketch SPI_Multiple_Busses shows how to use both HSPI and VSPI. However for ACAN2517FD, we proceed in a slightly different way:

```
#include <SPI.h>
....
SPIClass hspi (HSPI) ;
ACAN2517FD 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.

8.5 Connection with no interrupt pin

See the LoopBackDemoTeensy3xNoInt and LoopBackDemoESP32NoInt sketches.

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

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

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

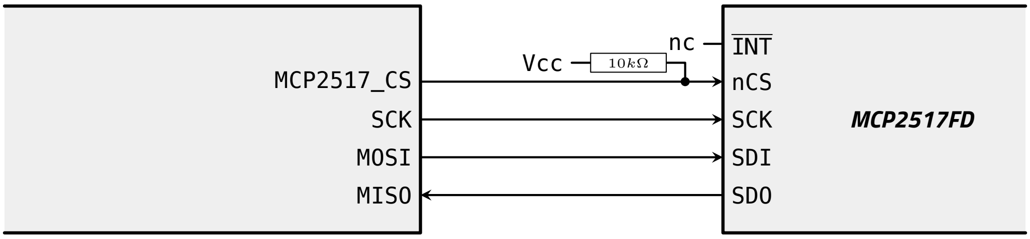

Figure 7 – Connection with no interrupt pin

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

8.6 Wiring schemes

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

8.6.1 Arduino Uno - MCP2518FDClick

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

9 Clock configuration

The `MCP251xFD` Oscillator Block Diagram is given in figure 9. Microchip recommends using a 4, 40 or 20 MHz CLKIN, Crystal or Ceramic Resonator. *A PLL can be enabled to multiply a 4 MHz clock by 10 by setting the PLLEN bit. Setting the SCLKDIV bit divides the SYSCLK by 2.*[7] **My opinion is that it is better to use an external clock (see section 3.2 page 7).**

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

```
class ACAN2517FDSettings {
  public: typedef enum {
    OSC_4MHz,
```

[7] DS20005678B, page 13.

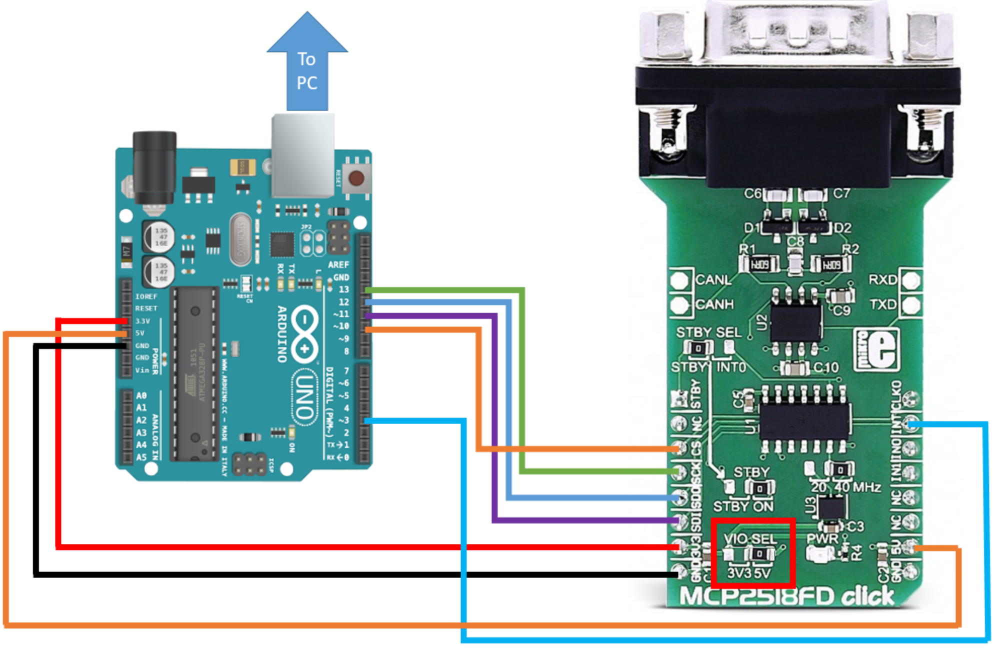

Figure 8 – Connecting an Arduino Uno with a MCP2518FDClick board

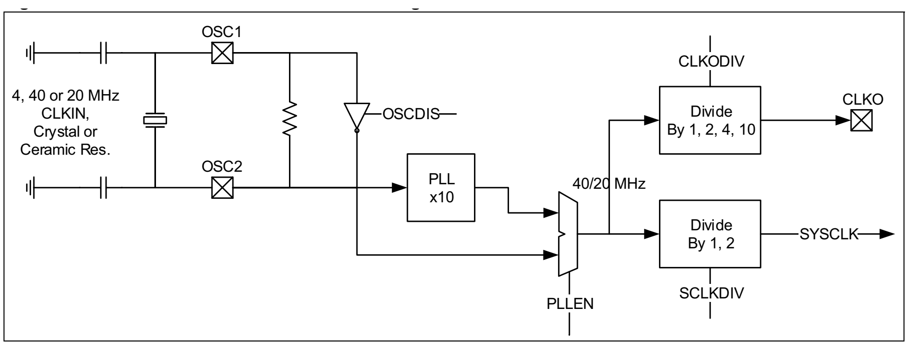

Figure 9 – MCP251xFD Oscillator Block Diagram (DS20005678B, figure 3.1 page 13)

```
    OSC_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 6. Note Microchip recommends a 40 MHz or 20 MHz `SYSCLK`.

The ACAN2517FDSettings class has two accessors that return current settings: oscillator() and sysClock().

Oscillator Frequency	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 6 – 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 oscillator). 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.

10 Transmit FIFO

The transmit FIFO (see figure 1 page 9) 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 CANFD frame; see section 13 page 28.

The *controller transmit FIFO* is located in the MCP2517FD RAM. It requires 16 bytes for each message (see section 14 page 29).

10.1 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 () ;
```

10.2 The driverTransmitBufferCount method

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

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

10.3 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.

11 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*[8].

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;
- 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 CANFD frame; see section 13 page 28.

The *transmit queue* is located in the MCP2517FD RAM. It requires 16 bytes for each message (see section 14 page 29).

12 Receive FIFO

The receive FIFO (see figure 1 page 9) 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 CANFD frame. See section 13 page 28.

When an incoming message is accepted by a receive filter:

[8] DS20005678B, section 4.5, page 28.

- 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).

13 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 14 page 29.

Note the `ACAN2517` library[9] 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 `mControllerReceiveF` properties, you can adjust the object size following your application requirements. The table 7 shows the possible values of these properties and the corresponding payload and object size.

By example, suppose your application always send data frames with no more than 24 bytes. You can set the `mControllerTransmitFIFOPayload` and `mControllerReceiveFIFOPayload` properties to `ACAN2517FDSettings::` 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;

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

- 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 7 – ACAN2517FD object size from payload size specification

13.1 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 7.

14 RAM usage

The MCP2517FD contains a 2048 bytes RAM that is used to store message objects[10]. 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.

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.

[10] DS20005688B, section 3.3, page 63.

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[11].

15 Sending frames: the `tryToSend` method

The `ACAN2517FD::tryToSend` method sends CAN 2.0B and CANFD frames:

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

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` property of the message specifies the transmit buffer:

- 0 for the transmit FIFO (section 10 page 25) ;
- 255 for the transmit Queue (section 11 page 27).

The `type` property of `inMessage` specifies how the frame is sent:

- `CAN_REMOTE`, the frame is sent in the CAN 2.0B remote frame format;
- `CAN_DATA`, the frame is sent in the CAN 2.0B data frame format;
- `CANFD_NO_BIT_RATE_SWITCH`, the frame is sent in CANFD format at arbitration bit rate, regardless of the `ACAN2517FDSettings::DATA_BITRATE_x`$_n$ setting;
- `CANFD_WITH_BIT_RATE_SWITCH`, with the `ACAN2517FDSettings::DATA_BITRATE_x1` setting, the frame is sent in CANFD format at arbitration bit rate, and otherwise in CANFD format with bit rate switch.

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

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

The tryToSend method returns:

- false if the message responds false to the isValid method (see section 6.8 page 16), 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.

15.1 Calling tryToSend with an CANMessage argument

The CANFDMessage class provides a constructor from a CANMessage object, so it is valid to call the tryToSend method with an CANMessage argument.

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

So, if the message.rtr is:

- true, the frame is sent in the CAN 2.0B remote frame format;
- false, the frame is sent in the CAN 2.0B data frame format.

15.2 Usage example

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 ;
    }
  }
  ...
}
```

16 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 18 page 37).

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.

The `type` property contains the received frame format:

- `CAN_REMOTE`, the received frame is a CAN 2.0B remote frame;
- `CAN_DATA`, the received frame is a CAN 2.0B data frame;
- `CANFD_NO_BIT_RATE_SWITCH`, the frame received frame is a CANFD frame, received at at arbitration bit rate;
- `CANFD_WITH_BIT_RATE_SWITCH`, the frame received frame is a CANFD frame, received with bit rate switch.

You need to manually dispatch the received messages. If you did not provide any receive filter, you should check the `type` property (remote or data frame?), the `ext` bit (base 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) ;  // Base data frame, id is 0x234
    }else if (message.rtr && !message.ext && (message.id == 0x542)) {
      handle_myMessage_2 (message) ;  // Base 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.

16.1 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, DataBitRateFactor::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`.

16.2 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 () ;
```

16.3 The receiveBufferCount method

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

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

16.4 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.

17 Acceptance filters

Note. The acceptance filters implemented in the ACAN2517 library, that handles a MCP2517FD CAN Controller in the CAN 2.0B mode[12], 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:

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

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

17.1 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 base 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 18 page 37).

The appendFrameFilter defines a filter that matches for an extended or base 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 base 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 base frames. So the accepted base 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`).

17.2 The `appendPassAllFilter` method

```
void MCP2517FDFilters::appendPassAllFilter (const ACANFDCallBackRoutine inCallBackRoutine) ;
```

This defines a filter that accepts all (base / 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.

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

17.4 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 base remote frames and all base 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.

17.5 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 base (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.

18 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.

19 The ACAN2517FD::begin method reference

19.1 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 19.2 page 39). 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.

19.2 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) ;
```

19.3 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 8. An error code could report several errors. The `ACAN2517FD` class defines static constants for naming errors.

19.3.1 `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.

19.3.2 `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.

Bit	Code	Static constant Name	Link
0	0x1	kRequestedConfigurationModeTimeOut	section 19.3.1 page 39
1	0x2	kReadBackErrorWith1MHzSPIClock	section 19.3.2 page 39
2	0x4	kTooFarFromDesiredBitRate	section 19.3.3 page 40
3	0x8	kInconsistentBitRateSettings	section 19.3.4 page 40
4	0x10	kINTPinIsNotAnInterrupt	section 19.3.5 page 41
5	0x20	kISRIsNull	section 19.3.6 page 41
6	0x40	kFilterDefinitionError	section 19.3.7 page 41
7	0x80	kMoreThan32Filters	section 19.3.8 page 41
8	0x100	kControllerReceiveFIFOSizeIsZero	section 19.3.9 page 41
9	0x200	kControllerReceiveFIFOSizeGreaterThan32	section 19.3.10 page 41
10	0x400	kControllerTransmitFIFOSizeIsZero	section 19.3.11 page 41
11	0x800	kControllerTransmitFIFOSizeGreaterThan32	section 19.3.12 page 41
12	0x1000	kControllerRamUsageGreaterThan2048	section 19.3.13 page 41
13	0x2000	kControllerTXQPriorityGreaterThan31	section 19.3.14 page 42
14	0x4000	kControllerTransmitFIFOPriorityGreaterThan31	section 19.3.15 page 42
15	0x8000	kControllerTXQSizeGreaterThan32	section 19.3.16 page 42
16	0x1_0000	kRequestedModeTimeOut	section 19.3.17 page 42
17	0x2_0000	kX10PLLNotReadyWithin1MS	section 19.3.18 page 42
18	0x4_0000	kReadBackErrorWithFullSpeedSPIClock	section 19.3.19 page 42
19	0x8_0000	kISRNotNullAndNoIntPin	section 19.3.20 page 42
20	0x10_0000	kInvalidTDCO	section 19.3.21 page 43

Table 8 – The `ACAN2517FD::begin` method error code bits

19.3.3 `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, DataBitRateFactor::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
}
```

19.3.4 `kInconsistentBitRateSettings`

The `ACAN2517FDSettings` 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`, `mArbitrationPhaseSegment1`, `mArbitrationPhaseSegm` `mArbitrationSJW`), and one or more resulting values are inconsistent. See section 20.2 page 49.

19.3.5 `kINTPinIsNotAnInterrupt`

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

19.3.6 `kISRIsNull`

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

19.3.7 `kFilterDefinitionError`

`settings.filterStatus()` returns a value different than `MCP2517FDFilters::kFiltersOk`, meaning that one or more filters are erroneous. See section 17.1 page 35.

19.3.8 `kMoreThan32Filters`

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

19.3.9 `kControllerReceiveFIFOSizeIsZero`

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

19.3.10 `kControllerReceiveFIFOSizeGreaterThan32`

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

19.3.11 `kControllerTransmitFIFOSizeIsZero`

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

19.3.12 `kControllerTransmitFIFOSizeGreaterThan32`

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

19.3.13 `kControllerRamUsageGreaterThan2048`

The configuration you have defined requires more than 2048 bytes of `MCP2517FD` internal RAM. See section 14 page 29.

19.3.14 kControllerTXQPriorityGreaterThan31

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

19.3.15 kControllerTransmitFIFOPriorityGreaterThan31

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

19.3.16 kControllerTXQSizeGreaterThan32

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

19.3.17 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.

19.3.18 kX10PLLNotReadyWithin1MS

You have requested the `OSC_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.

19.3.19 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.

19.3.20 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.

19.3.21 kInvalidTDCO

TDCO should be a 7-bit signed integer (i.e. $-64 \leqslant$ TDCO $\leqslant 63$). ACAN2517FDSettings constructor ensures this constraint, and provides a valid value in mTDCO property.

This error occurs when you have manually change the mTDCO property, for example:

```
ACAN2517FDSettings settings (... arguments ...) ;
settings.mTDCO = 100 ; // Invalid value
const uint32_t errorCode = can.begin (settings, [] { can.isr () ; }) ;
```

20 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.

20.1 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,
                               1000 * 1000, DataBitRateFactor::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 always 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 9 page 45), and 178 with a 20 MHz SYSCLK (table 10 page 46). 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, DataBitRateFactor::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, DataBitRateFactor::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 9 – 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:

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 10 – 20 MHz SYSCLK: the 178 exact bit rates

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                              500 * 1000, DataBitRateFactor::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 always 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 rate:

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               423 * 1000, DataBitRateFactor::DATA_BITRATE_x3) ;
  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, DataBitRateFactor::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 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 mArbitrationPhaseSegment1 property value, and decrement the mArbitrationPhaseSegment2 property value in order to sample the CAN Rx pin later.

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               500 * 1000, DataBitRateFactor::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 always 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:

$$\text{Arbitration Sampling Point} = 100 \cdot \frac{1 + \texttt{mArbitrationPhaseSegment1}}{1 + \texttt{mArbitrationPhaseSegment1} + \texttt{mArbitrationPhaseSegment2}}$$
$$\text{Data Sampling Point} = 100 \cdot \frac{1 + \texttt{mDataPhaseSegment1}}{1 + \texttt{mDataPhaseSegment1} + \texttt{mDataPhaseSegment2}}$$

20.2 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, DataBitRateFactor::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 11.

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

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

Bit	Error Name	Error
0	kBitRatePrescalerIsZero	mBitRatePrescaler == 0
1	kBitRatePrescalerIsGreaterThan256	mBitRatePrescaler > 256
2	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 20.3 page 50
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 11 – The ACAN2517FDSettings::CANBitSettingConsistency method error codes

20.3 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.

20.4 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, DataBitRateFactor::DATA_BITRATE_x1) ;
```

```
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual arbitration bit rate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  444,444 bit/s
  ...
}
```

Note. If CAN bit settings are not consistent (see section 20.2 page 49), the returned value is irrelevant.

20.5 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 20.2 page 49), the returned value is irrelevant.

20.6 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 20.2 page 49), the returned value is irrelevant.

20.7 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 20.2 page 49), the returned value is irrelevant.

20.8 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 ppm $= 10^{-6}$. In other words, $10,000$ ppm $= 1\%$.

Note. If CAN bit settings are not consistent (see section 20.2 page 49), the returned value is irrelevant.

20.9 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 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 20.2 page 49), the returned value is irrelevant.

20.10 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 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 20.2 page 49), the returned value is irrelevant.

20.11 Properties of the ACAN2517FDSettings class

All properties of the ACAN2517FDSettings class are declared public and are initialized (table 12).

20.11.1 The mTXCANIsOpenDrain property

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

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

20.11.2 The mINTIsOpenDrain property

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

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

20.11.3 The CLKO/SOF pin

The CLKO/SOF pin of the MCP2517FD controller is an output pin has five functions[13]:

[13] *Internally generated clock* is not SYSCLK, see figure 9 page 24.

20.11 Properties of the ACAN2517FDSettings class

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 20.1 page 43
mArbitrationPhaseSegment1	uint16_t	0	See section 20.1 page 43
mArbitrationPhaseSegment2	uint8_t	0	See section 20.1 page 43
mArbitrationSJW	uint8_t	0	See section 20.1 page 43
mArbitrationBitRateClosedTo-DesiredRate	bool	false	See section 20.1 page 43
mDataPhaseSegment1	uint16_t	0	See section 20.1 page 43
mDataPhaseSegment2	uint8_t	0	See section 20.1 page 43
mDataSJW	uint8_t	0	See section 20.1 page 43
mDataBitRateClosedToDesiredRate	bool	false	See section 20.1 page 43
mTXCANIsOpenDrain	bool	false	See section 20.11.1 page 52
mINTIsOpenDrain	bool	false	See section 20.11.2 page 52
mCLKOPin	CLKOpin	CLKO_DIVIDED_BY_10	See section 20.11.3 page 52
mISOCRCEnabled	bool	true	See section 20.11.5 page 54
mRequestedMode	RequestedMode	NormalFD	See section 20.11.4 page 54
mDriverTransmitFIFOSize	uint16_t	16	See section 10 page 25
mControllerTransmitFIFOSize	uint8_t	1	See section 10 page 25
mControllerTransmitFIFOPayload	PayloadSize	PAYLOAD_64	See section 10 page 25
mControllerTransmitFIFOPriority	uint8_t	0	See section 10 page 25
mControllerTransmitFIFO-RetransmissionAttempts	RetransmissionAttempts	UnlimitedNumber	See section 10 page 25
mControllerTXQSize	uint8_t	0	See section 11 page 27
mControllerTXQBufferPayload	PayloadSize	PAYLOAD_64	See section 11 page 27
mControllerTXQBufferPriority	uint8_t	31	See section 11 page 27
mControllerTXQBuffer-RetransmissionAttempts	RetransmissionAttempts	UnlimitedNumber	See section 11 page 27
mDriverReceiveFIFOSize	uint16_t	32	See section 12 page 27
mControllerReceiveFIFOPayload	PayloadSize	PAYLOAD_64	See section 12 page 27
mControllerReceiveFIFOSize	uint8_t	27	See section 12 page 27
mTDCO	int8_t	0	See section 20.11.6 page 54

Table 12 – Properties of the ACAN2517FDSettings class

- 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 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 () ; }) ;
```

20.11.4 The `mRequestedMode` property

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

20.11.5 The `mISOCRCEnabled` property

This property enables ISO CRC in CANFD 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.

20.11.6 The `mTDCO` property

Transmitter Delay Compensation is required when data phase bit time that is shorter than the transceiver loop delay. The `mTDCO` property is by default set to `mBitRatePrescaler` * `mDataPhaseSegment1` by the `ACAN2517FDSettings` constructor.

For more details, see `DS20005678D`, sections 3.4.3 to 3.4.8, pages 18 to 20.

21 Other ACAN2517FD methods

21.1 The `currentOperationMode` method

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

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

```
class ACAN2517FD {
  ...
  public: typedef enum : uint8_t {
    NormalFD = 0,
    Sleep = 1,
    InternalLoopBack = 2,
    ListenOnly = 3,
    Configuration = 4,
    ExternalLoopBack = 5,
    Normal20B = 6,
    RestrictedOperation = 7
  } OperationMode ;
  ...
} ;
```

21.2 The `recoverFromRestrictedOperationMode` method

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

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

This method returns `true` if both conditions are met:

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

It returns `false` otherwise.

21.3 The `errorCounters` method

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

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

21.4 The `diagInfos` method

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

Thanks to `Flole998` and `turmary`. This method returns:

- if `inIndex` is equal to 0, the `C1BDIAG0` register value, as described in `DS20005688B`, `REGISTER 3-20` page 42;

- if `inIndex` is not equal to 0, the `C1BDIAG1` register value, as described in `DS20005688B`, `REGISTER 3–21` page 43.

22 The `sendfd–odd` and `sendfd–even` sketches

I use theses two sketches for testing transmission and reception of CANFD frames. They try to send the frames as quickly as possible, repeatedly calling the `tryToSend` function.

They are designed for Teensy 3.5, with the `MCP2517FD` connected to `SPI1`. It is easy to adapt them to any other platform, although it can be tricky for an Arduino Uno which has little RAM ans small computation power.

Make a small CANFD network with two boards, one running the `sendfd–odd` sketch, the other running the `sendfd–even` sketch. Both display results in the Arduino Serial Monitor, you need two desktop computers.

The `sendfd–odd` sketch sends 50,000 CANFD base frames with an odd identifier, and waits for receiving 50,000 frames. Identifier is computed randomly, by `((micros () & 0x7FE) | 1)`.

The `sendfd–even` sketch sends 50,000 CANFD base frames with an even identifier, and waits for receiving 50,000 frames. Identifier is computed randomly, by `(micros () & 0x7FE)`.

In a CANFD network, as in a CAN network, two stations must not transmit frames with the same identifier: the arbitration does not operate, and a collision occurs when the DLC field or data is transmitted. As an odd identifier is always different from an even identifier, it is safe to run the two sketches in the same network.

You should adopt the same settings for the two sketches: same arbitration bit rate, same data bit rate factor.

Start the `sendfd–odd` sketch first: after initialization, it displays `Ready` in the Arduino Serial Monitor, meaning it is waiting for receiving frames.

Then, start the `sendfd–even` sketch: it sends frames immediatly; when the `sendfd–odd` sketch receives the first frame, it begins to send frames. Both sides send 50,000 frames. When the `sendfd–odd` sketch has sent and received 50,000 frames, it displays the duration from the reception of the first frame.

Every second, each sketch displays on its Arduino Serial Monitor:

- the sent frame count;
- the received frame count;
- the `MCP2517FD` error counter (`0`) if no error;
- the `MCP2517FD` operation mode (`0` in normal mode, `7` if it reaches the *Restricted Operation Mode*);
- the driver receive buffer peak count;
- the `MCP2517FD` receive buffer overflow count.

It is safe to observe that one side is stopping temporarily, while the other sends continously. For example, consider the case where the `sendfd–odd` sketch tries to send a frame with the `0x7FF` identifier; any frame with an even identifier has higher priority, so the `sendfd–even` sketch sends all remaining frames before the `sendfd–odd` sketch resumes sending.