ACANFD_FeatherM4CAN Arduino library, for Adafruit Feather M4 CAN Version ?.?.?

Pierre Molinaro

March 3, 2022

Contents

1 Versions **4**

2 Features **4**

3 CAN Interfaces **4**

3.1 CAN0 . . . 4

3.2 CAN1 . . . 5

4 Data flow **6**

5 A simple example: LoopBackDemoCANFD_CAN1 **7**

6 The CANMessage class **9**

7 The CANFDMessage class **10**

7.1 Properties . . . 10

7.2 The default constructor . . . 11

7.3 Constructor from CANMessage . . . 11

7.4 The type property . . . 12

7.5 The len property . . . 12

7.6 The idx property . . . 12

7.7 The pad method . . . 13

7.8 The isValid method . . . 13

8 Transmit FIFO **13**

8.1 The driverTransmitFIFOSize method . . . 14

8.2 The driverTransmitFIFOCount method . . . 14

8.3 The driverTransmitFIFOPeakCount method 14

9 Transmit buffers (TxBuffer$_i$) 14

10 Receive FIFOs 14

10.1 The hardwareReceiveBufferOverflowCount method 15

10.2 The resetHardwareReceiveBufferOverflowCount method 16

11 Payload size 16

11.1 The ACAN2517FDSettings::objectSizeForPayload static method 17

12 RAM usage 17

13 Sending frames: the tryToSend method 18

13.1 Calling tryToSend with an CANMessage argument 19

13.2 Testing a send buffer: the sendBufferNotFullForIndex method 19

13.3 Usage example 19

14 Retrieving received messages using the receive method 20

14.1 Driver receive buffer size 21

14.2 The receiveBufferSize method 22

14.3 The receiveBufferCount method 22

14.4 The receiveBufferPeakCount method 22

15 Acceptance filters 22

15.1 An example 23

15.2 The appendPassAllFilter method 24

15.3 The appendFormatFilter method 24

15.4 The appendFrameFilter method 24

15.5 The appendFilter method 25

16 The dispatchReceivedMessage method 25

17 The ACAN2517FD::begin method reference 26

17.1 The prototypes 26

17.2 Defining explicitly the interrupt service routine 27

17.3 The error code 27

17.3.1 kRequestedConfigurationModeTimeOut 27

17.3.2 kReadBackErrorWith1MHzSPIClock 27

17.3.3 kTooFarFromDesiredBitRate 27

17.3.4 kInconsistentBitRateSettings 28

17.3.5 kINTPinIsNotAnInterrupt 28

17.3.6 kISRIsNull 29

17.3.7 kFilterDefinitionError 29

17.3.8 kMoreThan32Filters 29

17.3.9 kControllerReceiveFIFOSizeIsZero 29

17.3.10 kControllerReceiveFIFOSizeGreaterThan32 . . . 29
17.3.11 kControllerTransmitFIFOSizeIsZero . . . 29
17.3.12 kControllerTransmitFIFOSizeGreaterThan32 . . . 29
17.3.13 kControllerRamUsageGreaterThan2048 . . . 29
17.3.14 kControllerTXQPriorityGreaterThan31 . . . 30
17.3.15 kControllerTransmitFIFOPriorityGreaterThan31 . . . 30
17.3.16 kControllerTXQSizeGreaterThan32 . . . 30
17.3.17 kRequestedModeTimeOut . . . 30
17.3.18 kX10PLLNotReadyWithin1MS . . . 30
17.3.19 kReadBackErrorWithFullSpeedSPIClock . . . 30
17.3.20 kISRNotNullAndNoIntPin . . . 30
17.3.21 kInvalidTDCO . . . 31

18 ACAN2517FDSettings class reference 31

18.1 The ACAN2517FDSettings constructor: computation of the CAN bit settings . . . 31
18.2 The CANBitSettingConsistency method . . . 37
18.3 The kArbitrationTQCountNotDivisibleByDataBitRateFactor error . . . 38
18.4 The actualArbitrationBitRate method . . . 38
18.5 The exactArbitrationBitRate method . . . 39
18.6 The exactDataBitRate method . . . 39
18.7 The ppmFromDesiredArbitrationBitRate method . . . 39
18.8 The ppmFromDesiredDataBitRate method . . . 39
18.9 The arbitrationSamplePointFromBitStart method . . . 39
18.10 The dataSamplePointFromBitStart method . . . 40
18.11 Properties of the ACANFD_FeatherM4CAN_Settings class . . . 40
18.11.1 The mTXCANIsOpenDrain property . . . 40
18.11.2 The mINTIsOpenDrain property . . . 40
18.11.3 The mRequestedMode property . . . 41
18.11.4 The mTDCO property . . . 42

19 Other ACAN2517FD methods 42

19.1 The currentOperationMode method . . . 42
19.2 The recoverFromRestrictedOperationMode method . . . 42
19.3 The errorCounters method . . . 43
19.4 The diagInfos method . . . 43
19.5 The end method . . . 43

20 The sendfd-odd and sendfd-even sketches 44

1 Versions

Version	Date	Comment
?.?.?	March ?, 2022	Initial release.

2 Features

The `ACANFD_FeatherM4CAN` library is a CANFD (*Controller Area Network with Flexible Data*) Controller driver for the *Adafruit Feather M4 CAN*[1] board running Arduino. It handles CANFD frames.

This library is compatible with other ACAN librairies.

It has been designed to make it easy to start and to be easily configurable:

- handles the `CAN0` and `CAN1` CANFD modules;
- default configuration sends and receives any frame – no default filter to provide;
- efficient built-in CAN bit settings computation from arbitration and data bit rates;
- user can fully define its own CAN bit setting values;
- driver and controller transmit buffer sizes are customisable;
- driver and controller receive buffer size is customisable;
- overflow of the driver receive buffer is detectable;
- the message RAM allocation is customizable and the driver checks no overflow occurs;
- *internal loop back*, *external loop back* controller modes are selectable.

3 CAN Interfaces

The Adafruit Feather M4 CAN board contains a ATSAME51J19 that implements two CANFD modules: `CAN0` and `CAN1`.

3.1 `CAN0`

The microcontroller `CAN0` pins are available on the board connector: `D12` is `CAN0_TX`, `D13` is `CAN0_RX` (see figure 1). For connecting to a CAN bus, you should add a CANFD transceiver. Note `D13` is also connected to builtin red led.

[1] https://www.adafruit.com/product/4759

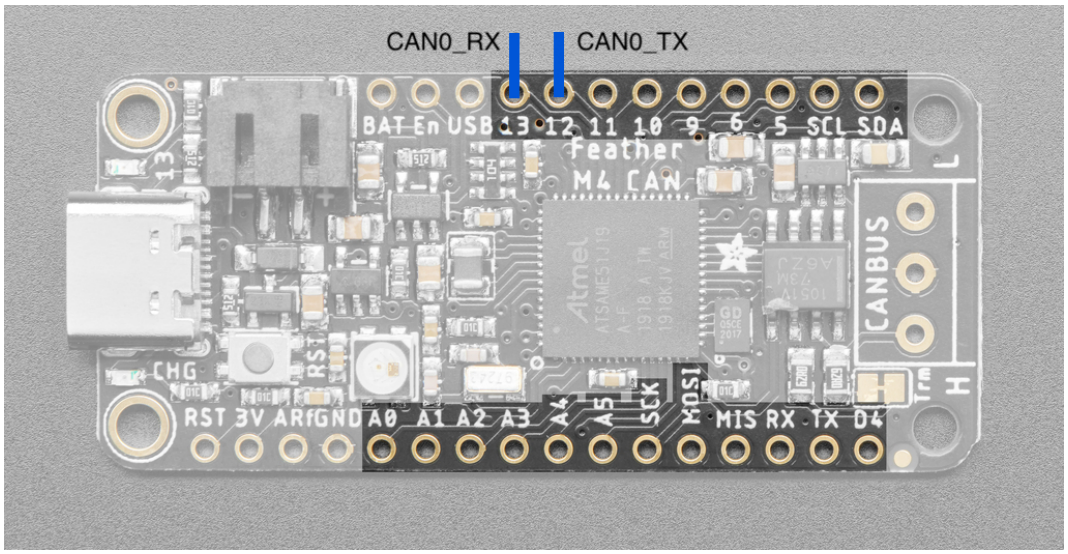

Figure 1 – CAN0 pins

3.2 CAN1

The microcontroller CAN1 pins are not available on the board connector, but CANH and CANL pins (see figure 2). The board includes a a 3V-logic compatible transceiver[2]. Note the library handles two additional signals: PIN_CAN_STANDBY is configured as low digital output (turning off transceiver's STANDBY mode), and pin 4 is configured as high digital output (turning on transceiver's power).

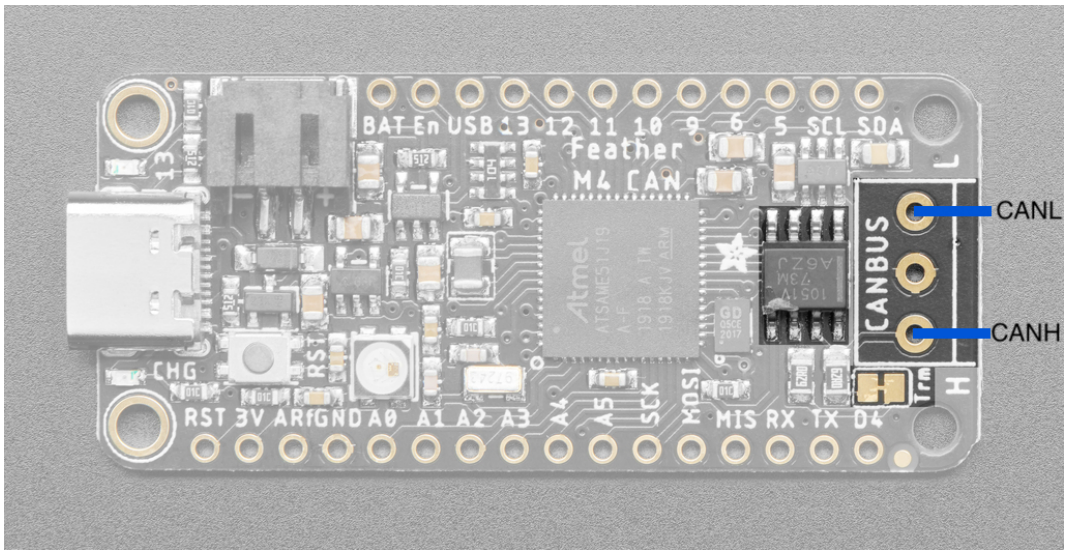

Figure 2 – CAN1 pins

[2] https://learn.adafruit.com/adafruit-feather-m4-can-express/pinouts

4 Data flow

The figure 3 illustrate default message flow of sending and receiving CANFD messages for `CAN0` and `CAN1` modules.

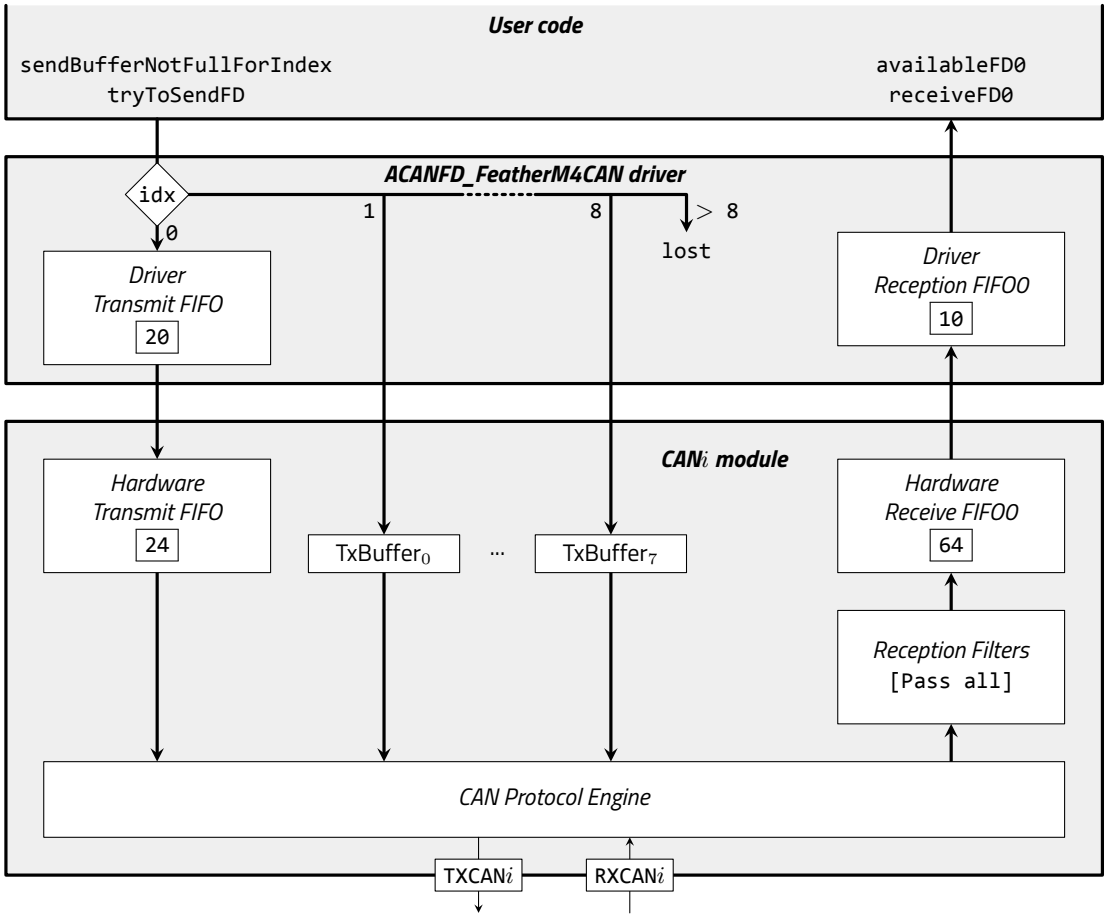

Figure 3 – Message flow in `ACANFD_FeatherM4CAN` driver and `CAN`i module, default configuration

Sending messages. The `ACANFD_FeatherM4CAN` driver defines a *driver transmit FIFO* (default size: 20 messages), and configures the module with a *hardware transmit FIFO* with a size of 24 messages, and 8 individual `TxBuffer` whose capacity is one message.

A message is defined by an instance of the `CANFDMessage` or `CANMessage` class. For sending a message, user code calls the `tryToSendFD` method – see section 13 page 18 for details, and the `idx` property of the sent message should be:

- 0 (default value), for sending via *driver transmit FIFO* and *hardware transmit FIFO*;
- 1, for sending via *TxBuffer*$_0$;
- ...
- 8, for sending via *TxBuffer*$_7$.

If the `idx` property is greater than 8, the message is lost.

You can call the `sendBufferNotFullForIndex` method (section 13.2 page 19) for testing if a send buffer is not full.

Receiving messages. The *CAN Protocol Engine* transmits all correct frames to the *reception filters*. By default, they are configured as pass-all, see section 15 page 22 for configuring them. Messages that pass the filters are stored in the *Hardware Reception FIFO0*; its size is 64 messages by default. The interrupt service routine transfers the messages from this FIFO0 to the *Driver Receive FIFO0*. The size of the *Driver Receive Buffer* is 10 by default – see section 14.1 page 21 for changing the default value. Two user methods are available:

- the `availableFD0` method returns `false` if the *Driver Receive FIFO0* is empty, and `true` otherwise;
- the `receiveFD0` method retrieves messages from the *Driver Receive FIFO0* – see section 14 page 20.

5 A simple example: `LoopBackDemoCANFD_CAN1`

The `LoopBackDemoCANFD_CAN1` sketch is a sample code for introducing the `ACANFD_FeatherM4CAN` library. It demonstrates how to configure the driver, to send a CANFD message, and to receive a CANFD message.

Note: this code runs without any CAN connection, the `CAN1` module is configured in `EXTERNAL_LOOP_BACK` mode (see §§); the `CAN1` module receives every CANFD frame it sends, and emitted frames can be observed on `CANH`/`CANL` pins.

```
#include <ACANFD_FeatherM4CAN.h>
```

This line includes the `ACANFD_FeatherM4CAN` library.

The `setup` function.

```
void setup () {
//--- Switch on builtin led
  pinMode (LED_BUILTIN, OUTPUT) ;
  digitalWrite (LED_BUILTIN, HIGH) ;
//--- Start serial
  Serial.begin (115200) ;
//--- 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.

```
  ACANFD_FeatherM4CAN_Settings settings (1000 * 1000, DataBitRateFactor::x2) ;
```

Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACANFD_FeatherM4CAN_Settings` class. The constructor has two parameters: the desired CAN arbitration bit rate (here, 1 Mbit/s), and the data bit rate, given by a multiplicative factor of the arbitration bit rate; here,

the data bit rate is 1 Mbit/s * 2 = 2 Mbit/s. It returns a `settings` object fully initialized with CAN bit settings for the desired arbitration and data bit rates, and default values for other configuration properties.

```
settings.mModuleMode = ACANFD_FeatherM4CAN_Settings::EXTERNAL_LOOP_BACK ;
```

This is the second step. You can override the values of the properties of `settings` object. Here, the `mModuleMode` property is set to `EXTERNAL_LOOP_BACK` – its value is `NORMAL_FD` by default. Setting this property enables *external loop back*, that is you can run this demo sketch even it you have no connection to a physical CAN network. The section 18.11 page 40 lists all properties you can override.

```
  const uint32_t errorCode = can1.beginFD () ;
```

This is the third step, configuration of the `CAN1` driver with `settings` values (for configuring the `CAN0` module, use the `can0` variable). 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 15 page 22.

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

Last step: the configuration of the `can` driver returns an error code, stored in the `errorCode` constant. It has the value 0 if all is ok – see section 17.3 page 27.

The `pseudoRandomValue` function.

This function generates values that are used for generating random CANFD messages.

The global variables.

```
static const uint32_t PERIOD = 1000 ;
static uint32_t gBlinkDate = PERIOD ;
static uint32_t gSentCount = 0 ;
static uint32_t gReceiveCount = 0 ;
static CANFDMessage gSentFrame ;
static bool gOk = true ;
```

The `gBlinkDate` global variable is used for sending a CAN message every second. The `gSentCount` global variable counts the number of sent messages. The sent message is stored in the `gSentFrame` variable. While `gOk` is true, the received message is compared to the sent message. If they are different, `gOk` is set to false, and no more message is sent. The `gReceivedCount` global variable counts the number of sucessfully received messages.

The `loop` function.

```
void loop () {
  if (gBlinkDate <= millis ()) {
    gBlinkDate += PERIOD ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
    if (gOk) {
      ... build random CANFD frame ...
```

```
      const uint32_t sendStatus = can1.tryToSendReturnStatusFD (gSentFrame) ;
      if (sendStatus == 0) {
        gSentCount += 1 ;
        Serial.print ("Sent ") ;
        Serial.println (gSentCount) ;
      }else{
        Serial.print ("Sent error 0x") ;
        Serial.println (sendStatus) ;
      }
    }
  }
//--- Receive frame
  CANFDMessage frame ;
  if (gOk && can1.receiveFD0 (frame)) {
    bool sameFrames = ... compare frame and gSentFrame ... ;
    if (sameFrames) {
      gReceiveCount += 1 ;
      Serial.print ("Received ") ;
      Serial.println (gReceiveCount) ;
    }else{
      gOk = false ;
      ... Print error ...
    }
  }
}
```

6 The CANMessage class

Note. The CANMessage class is declared in the CANMessage.h header file. The class declaration is protected by an include guard that causes the macro GENERIC_CAN_MESSAGE_DEFINED to be defined. The ACAN[3] (version 1.0.3 and above) driver, the ACAN2515[4] driver and the ACAN2517[5] 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 7.3 page 11).

```
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
```

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

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

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

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

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

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

- for a received message, it contains the acceptance filter index (see section 16 page 25);
- on sending messages, it is used for selecting the transmit buffer (see section 13 page 18).

7 The `CANFDMessage` class

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

A 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
```

7.1 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).

7.2 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

7.3 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

7.4 The type property

The type property 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

7.5 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 7.7 page 13) for padding with 0x00 bytes to the next valid length.

7.6 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 16 page 25);
- on sending messages, it is used for selecting the transmit buffer (see section 13 page 18).

7.7 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], frame.data [22], frame.data [23] are 0
```

7.8 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 12, and `false` otherwise.

8 Transmit FIFO

The transmit FIFO (see figure 3 page 6) is composed by:

- the *driver transmit FIFO*, whose size is positive or zero (default 20); you can change the default size by setting the `mDriverTransmitFIFOSize` property of your `settings` object;
- the *hardware transmit FIFO*, whose size is between 1 and 32 (default 24); you can change the default size by setting the `mHardwareTransmitTxFIFOSize` property of your `settings` object.

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

- if the *controller transmit FIFO* is not full, the message is appended to it, and `tryToSendFD` returns `0`;
- otherwise, if the *driver transmit FIFO* is not full, the message is appended to it, and `tryToSendFD` returns `0`; the interrupt service routine will transfer messages from *driver transmit FIFO* to the *hardware transmit FIFO* while it is not full;

- otherwise, both FIFOs are full, the message is not stored and tryToSendFD returns the kTransmitBufferOverflow error.

The transmit FIFO ensures sequentiality of emission.

8.1 The driverTransmitFIFOSize method

The driverTransmitFIFOSize method returns the allocated size of this driver transmit FIFO, that is the value of settings.mDriverTransmitFIFOSize when the begin method is called.

```
const uint32_t s = can0.driverTransmitFIFOSize () ;
```

8.2 The driverTransmitFIFOCount method

The driverTransmitFIFOCount method returns the current number of messages in the driver transmit FIFO.

```
const uint32_t n = can0.driverTransmitFIFOCount () ;
```

8.3 The driverTransmitFIFOPeakCount method

The driverTransmitFIFOPeakCount method returns the peak value of message count in the driver transmit FIFO

```
const uint32_t max = can0.driverTransmitFIFOPeakCount () ;
```

If the transmit FIFO is full when tryToSend is called, the return value of this call is kTransmitBufferOverflow. In such case, the following calls of driverTransmitBufferPeakCount() will return driverTransmitFIFOSize ()+1.

So, when driverTransmitFIFOPeakCount() returns a value lower or equal to transmitFIFOSize (), it means that calls to tryToSendFD do not provide any overflow of the driver transmit FIFO.

9 Transmit buffers (TxBuffer$_i$)

You can use settings.mHardwareDedicacedTxBufferCount TxBuffers for sending messages. A TxBuffer has a capacity of 1 message. So it is either empty, either full.

The settings.mHardwareDedicacedTxBufferCount property can be set to any integer value between 0 and 32.

10 Receive FIFOs

A CAN module contains two receive FIFOs, FIFO0 and FIFO1. **Currently, only FIFO0 is handled, FIFO1 is not configured.**

The receive FIFO (see figure 3 page 6) is composed by:

- the *controller receive FIFO* (in the MCP2517FD RAM), whose size is between 1 and 32 (default 27); you can change the default size by setting the mControllerReceiveFIFOSize property of your settings object;
- the *driver receive FIFO* (in library software), whose size is positive (default 32); you can change the default size by setting the mDriverReceiveFIFOSize property of your settings object.

The receive FIFO mechanism ensures sequentiality of reception. The ACAN2517FD::available, ACAN2517FD::receive and ACAN2517FD::dispatchReceivedMessage methods work only with the *driver receive FIFO*.

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 11 page 16.

When a valid incoming CANFD message is received, the MCP2517FD submits it to the *reception filters*. If it is accepted by a receive filter, it is transferred to the *controller receive FIFO*. Then, the behaviour depends from the library release.

Releases <= 2.1.6. When an incoming message has been accepted by a receive filter:

- the message is removed from the *controller receive FIFO*;
- if the *driver receive FIFO* is not full, it is stored in the *driver 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*. If the *driver receive FIFO* is full, the message is lost. So the *driver receive FIFO* and the *controller receive FIFO* never overflow.

Releases >= 2.1.7. When an incoming message has been accepted by a receive filter:

- if the *driver receive FIFO* is not full, it is removed from the *controller receive FIFO* and stored in the *driver receive FIFO*;
- otherwise, the message remains in the *controller receive FIFO*.

So the *driver receive FIFO* never overflows, but *controller receive FIFO* may (you can get the overflow count by call the hardwareReceiveBufferOverflowCount method, see section 10.1 page 15).

As soon as the *driver receive FIFO* becomes not full, messages from *controller receive FIFO* are transferred to the *driver receive FIFO* by the *interrupt service routine* until the *driver receive FIFO* becomes full again or the *driver receive FIFO* becomes empty.

10.1 The hardwareReceiveBufferOverflowCount method

```
uint8_t ACAN2517FD::hardwareReceiveBufferOverflowCount (void) const ;
```

The driver maintains an uint8_t counter of *controller receive FIFO* overflows, saturating at 255. The method returns the current value of the counter.

10.2 The resetHardwareReceiveBufferOverflowCount method

```
void ACAN2517FD::resetHardwareReceiveBufferOverflowCount (void) ;
```

The driver maintains an uint8_t counter of *controller receive FIFO* overflows. The method resets the current value of the counter.

11 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 12 page 17.

Note the ACAN2517 library[6] handles an MCP2517FD in CAN 2.0B mode. As CAN 2.0B frames contains at most 8 bytes, the size of each object is 16 bytes, allowing using up to 128 objects.

With the mControllerTransmitFIFOPayload, the mControllerTXQBufferPayload and the mControllerReceiveFIFOPayload properties, you can adjust the object size following your application requirements. The table 5 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::PAYLOAD_24, leading to an object size equal to 32 bytes. If your application also receives data frames with no more than 24 bytes, you can also set the mControllerReceiveFIFOPayload property to ACAN2517FDSettings::PAYLOAD_24. All your objects require 32 bytes, allowing 64 objects in the MCP2517FD RAM. The benefit is you can now increase controller buffer sizes, for example:

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

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

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 5 – ACAN2517FD object size from payload size specification

11.1 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 5.

12 RAM usage

The MCP2517FD contains a 2048 bytes RAM that is used to store message objects[7]. 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.

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

[7] DS20005688B, section 3.3, page 63.

```
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[8].

13 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 8 page 13) ;
- 255 for the transmit Queue (section **??** page **??**).

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.

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

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

The tryToSend method returns:

- false if the message responds false to the isValid method (see section 7.8 page 13), 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.

13.1 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.

13.2 Testing a send buffer: the sendBufferNotFullForIndex method

13.3 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 ;
    }
  }
  ...
}
```

14 Retrieving received messages using the `receive` method

There are two ways for retrieving received messages :

- using the `receive` method, as explained in this section;
- using the `dispatchReceivedMessage` method (see section 16 page 25).

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.

14.1 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.

14.2 The receiveBufferSize method

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

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

14.3 The receiveBufferCount method

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

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

14.4 The receiveBufferPeakCount method

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

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

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

15 Acceptance filters

Note. The acceptance filters implemented in the ACAN2517 library, that handles a MCP2517FD CAN Controller in the CAN 2.0B mode[9], are almost identical, they differ only from the prototype of the callback routine.

If you invoke the ACAN2517FD.begin method with two arguments, it configures the MCP2517FD for receiving all messages.

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

If you want to define receive filters, you have to set up an MCP2517FDFilters instance object, and pass it as third argument of the ACAN2517FD.begin method:

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

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

15.1 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 16 page 25).

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).

15.2 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.

15.3 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.

15.4 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.

15.5 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:

$$identifier \ \& \ inMask == inAcceptance$$

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:

$$inAcceptance \ \& \ inMask == inAcceptance$$

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

If inFormat is equal to kStandard:

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

If inFormat is equal to kExtended:

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

16 The dispatchReceivedMessage method

Sample sketch: the LoopBackDemoTeensy3xWithFilters shows how using the dispatchReceivedMessage method.

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

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

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

```
  ...
}
```

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

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

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

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

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

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

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

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

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

17 The `ACAN2517FD::begin` method reference

17.1 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 17.2 page 27). 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.

17.2 Defining explicitly the interrupt service routine

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

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

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

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

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

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

17.3 The error code

The `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 6. An error code could report several errors. The `ACAN2517FD` class defines static constants for naming errors.

17.3.1 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.

17.3.2 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.

17.3.3 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,
```

Bit	Code	Static constant Name	Link
0	0x1	kRequestedConfigurationModeTimeOut	section 17.3.1 page 27
1	0x2	kReadBackErrorWith1MHzSPIClock	section 17.3.2 page 27
2	0x4	kTooFarFromDesiredBitRate	section 17.3.3 page 27
3	0x8	kInconsistentBitRateSettings	section 17.3.4 page 28
4	0x10	kINTPinIsNotAnInterrupt	section 17.3.5 page 28
5	0x20	kISRIsNull	section 17.3.6 page 29
6	0x40	kFilterDefinitionError	section 17.3.7 page 29
7	0x80	kMoreThan32Filters	section 17.3.8 page 29
8	0x100	kControllerReceiveFIFOSizeIsZero	section 17.3.9 page 29
9	0x200	kControllerReceiveFIFOSizeGreaterThan32	section 17.3.10 page 29
10	0x400	kControllerTransmitFIFOSizeIsZero	section 17.3.11 page 29
11	0x800	kControllerTransmitFIFOSizeGreaterThan32	section 17.3.12 page 29
12	0x1000	kControllerRamUsageGreaterThan2048	section 17.3.13 page 29
13	0x2000	kControllerTXQPriorityGreaterThan31	section 17.3.14 page 30
14	0x4000	kControllerTransmitFIFOPriorityGreaterThan31	section 17.3.15 page 30
15	0x8000	kControllerTXQSizeGreaterThan32	section 17.3.16 page 30
16	0x1_0000	kRequestedModeTimeOut	section 17.3.17 page 30
17	0x2_0000	kX10PLLNotReadyWithin1MS	section 17.3.18 page 30
18	0x4_0000	kReadBackErrorWithFullSpeedSPIClock	section 17.3.19 page 30
19	0x8_0000	kISRNotNullAndNoIntPin	section 17.3.20 page 30
20	0x10_0000	kInvalidTDCO	section 17.3.21 page 31

Table 6 – The ACAN2517FD::begin method error code bits

```
                                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
}
```

17.3.4 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, mArbitrationPhaseSegment2, mArbitrationSJW), and one or more resulting values are inconsistent. See section 18.2 page 37.

17.3.5 kINTPinIsNotAnInterrupt

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

17.3.6 kISRIsNull

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

17.3.7 kFilterDefinitionError

settings.filterStatus() returns a value different than MCP2517FDFilters::kFiltersOk, meaning that one or more filters are erroneous. See section 15.1 page 23.

17.3.8 kMoreThan32Filters

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

17.3.9 kControllerReceiveFIFOSizeIsZero

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

17.3.10 kControllerReceiveFIFOSizeGreaterThan32

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

17.3.11 kControllerTransmitFIFOSizeIsZero

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

17.3.12 kControllerTransmitFIFOSizeGreaterThan32

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

17.3.13 kControllerRamUsageGreaterThan2048

The configuration you have defined requires more than 2048 bytes of MCP2517FD internal RAM. See section 12 page 17.

17.3.14 kControllerTXQPriorityGreaterThan31

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

17.3.15 kControllerTransmitFIFOPriorityGreaterThan31

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

17.3.16 kControllerTXQSizeGreaterThan32

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

17.3.17 kRequestedModeTimeOut

During configuration by the `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.

17.3.18 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.

17.3.19 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.

17.3.20 kISRNotNullAndNoIntPin

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

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

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

```
  ...
}
```

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

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

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

See the `LoopBackDemoTeensy3xNoInt` and `LoopBackDemoESP32NoInt` sketches.

17.3.21 `kInvalidTDCO`

`TDCO` should be a 7-bit signed integer (i.e. $-64 \leqslant \texttt{TDCO} \leqslant 63$). `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 () ; }) ;
```

18 `ACAN2517FDSettings` class reference

Note. The `ACAN2517FDSettings` class is not Arduino specific. You can compile it on your desktop computer with your favorite C++ compiler.

18.1 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 7 page 33), and 178 with a 20 MHz SYSCLK (table 8 page 34). 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
  ...
}
```

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

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 7 – 40 MHz SYSCLK: the 184 exact bit rates

computed actual bit to be exactly the desired bit rate:

```
void setup () {
  ...
  ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_4MHz10xPLL,
                               500 * 1000, DataBitRateFactor::DATA_BITRATE_x1,
                               0) ; // Max distance is 0 ppm
```

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 8 – 20 MHz SYSCLK: the 178 exact bit rates

```
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:

$$\text{Actual Arbitration Bit Rate} = \frac{\text{SYSCLK}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mArbitrationPhaseSegment1} + \texttt{mArbitrationPhaseSegment2})}$$
$$\text{Actual Data Bit Rate} = \frac{\text{SYSCLK}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mDataPhaseSegment1} + \texttt{mDataPhaseSegment2})}$$

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

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

18.2 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 9.

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 18.3 page 38
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 9 – The ACAN2517FDSettings::CANBitSettingConsistency method error codes

18.3 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.

18.4 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 18.2 page 37), the returned value is irrelevant.

18.5 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 18.2 page 37), the returned value is irrelevant.

18.6 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 18.2 page 37), the returned value is irrelevant.

18.7 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 18.2 page 37), the returned value is irrelevant.

18.8 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 18.2 page 37), the returned value is irrelevant.

18.9 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): $1\ \text{ppc} = 1\% = 10^{-2}$. 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 18.2 page 37), the returned value is irrelevant.

18.10 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): $1\ \text{ppc} = 1\% = 10^{-2}$. 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 18.2 page 37), the returned value is irrelevant.

18.11 Properties of the ACANFD_FeatherM4CAN_Settings class

All properties of the ACAN2517FDSettings class are declared public and are initialized (table 10).

18.11.1 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.

18.11.2 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.

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 ;
  ...
} ;
```

Property	Type	Initial value	Comment
mOscillator	Oscillator	Constructor argument	
mSysClock	uint32_t	Constructor argument	
mDesiredBitRate	uint32_t	Constructor argument	
mBitRatePrescaler	uint16_t	0	See section 18.1 page 31
mArbitrationPhaseSegment1	uint16_t	0	See section 18.1 page 31
mArbitrationPhaseSegment2	uint8_t	0	See section 18.1 page 31
mArbitrationSJW	uint8_t	0	See section 18.1 page 31
mArbitrationBitRateClosedToDesiredRate	bool	false	See section 18.1 page 31
mDataPhaseSegment1	uint16_t	0	See section 18.1 page 31
mDataPhaseSegment2	uint8_t	0	See section 18.1 page 31
mDataSJW	uint8_t	0	See section 18.1 page 31
mDataBitRateClosedToDesiredRate	bool	false	See section 18.1 page 31
mTXCANIsOpenDrain	bool	false	See section 18.11.1 page 40
mINTIsOpenDrain	bool	false	See section 18.11.2 page 40
mRequestedMode	RequestedMode	NormalFD	See section 18.11.3 page 41
mDriverTransmitFIFOSize	uint16_t	16	See section 8 page 13
mControllerTransmitFIFOSize	uint8_t	1	See section 8 page 13
mControllerTransmitFIFOPayload	PayloadSize	PAYLOAD_64	See section 8 page 13
mControllerTransmitFIFOPriority	uint8_t	0	See section 8 page 13
mControllerTransmitFIFORetransmissionAttempts	RetransmissionAttempts	UnlimitedNumber	See section 8 page 13
mControllerTXQSize	uint8_t	0	See section **??** page **??**
mControllerTXQBufferPayload	PayloadSize	PAYLOAD_64	See section **??** page **??**
mControllerTXQBufferPriority	uint8_t	31	See section **??** page **??**
mControllerTXQBufferRetransmissionAttempts	RetransmissionAttempts	UnlimitedNumber	See section **??** page **??**
mDriverReceiveFIFOSize	uint16_t	32	See section 10 page 14
mControllerReceiveFIFOPayload	PayloadSize	PAYLOAD_64	See section 10 page 14
mControllerReceiveFIFOSize	uint8_t	27	See section 10 page 14
mTDCO	int8_t	0	See section 18.11.4 page 42

Table 10 – Properties of the ACAN2517FDSettings class

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 () ; }) ;
```

18.11.3 The mRequestedMode property

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

18.11.4 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.

19 Other ACAN2517FD methods

19.1 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 ;
  ...
} ;
```

19.2 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.

19.3 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.

19.4 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.

19.5 The end method

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

This method has not been tested with the ESP32.

This method disables the library and the MCP2517FD chip. It performs:

1. detach interrupt pin (if any);
2. repeatedly requests the *configuration mode*, and waits for 2 ms until this mode is reached;
3. resets the MCP2517FD;
4. ESP32 only: delete associated task;
5. deallocate buffers.

Note the SPI is not disabled.

If the MCP2517FD reaches the *configuration mode* within 2 ms, the end method returns true.

If the MCP2517FD does not reach the *configuration mode* after 2 ms, the end method returns false.

The LoopBackTestEndFunctionTeensy3x sketch is an example of end method call. Every 1 000 sent messages, the end method is called, the CAN driver is released, a new one is allocated and configured with the begin method.

20 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 and 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 adapt 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.