ACANFD_FeatherM4CAN Arduino library, for Adafruit Feather M4 CAN Version 1.2.0

Pierre Molinaro

March 12, 2022

Contents

1 Versions

Version	Date	Comment
1.2.0	March 12, 2022	Added `dispatchReceivedMessage` method.
		Added `dispatchReceivedMessageFIFO0` method.
		Added `dispatchReceivedMessageFIFO1` method.
		Added `LoopBackDemoCANFD_CAN1_dispatch` sample sketch.
1.1.0	March 10, 2022	Added handling Rx FIFO 1.
		Added receive standard filters.
		Added receive extended filters.
		Added `LoopBackDemoCANFD_CAN1_StandardFilters` sample sketch.
		Added `LoopBackDemoCANFD_CAN1_ExtendedFilters` sample sketch.
1.0.1	March 9, 2022	Added constraint `settings.mHardwareTransmitTxFIFOSize` $\geqslant 2$.
		Added constraint `settings.mHardwareDedicacedTxBufferCount` $\leqslant 30$.
		Fixed `tryToSendReturnStatusFD`, this method was returning error 1 in release 1.0.0.
1.0.0	March 8, 2022	Initial release (buggy, removed).

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 and `ACAN2517FD` library.

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;
- up to 128 standard reception filters can be easily defined;
- up to 128 extended reception filters can be easily defined;
- driver and controller transmit buffer sizes are customisable;
- driver and controller receive buffer size is customisable;
- overflow of the driver receive buffer is detectable;

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

- 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.

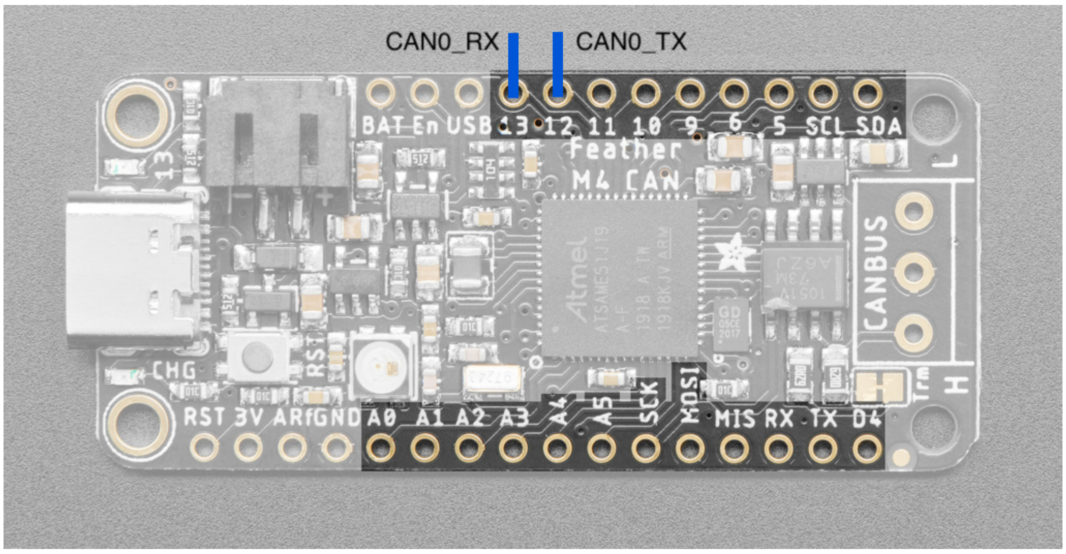
CAN0_RX
CAN0_TX
BAT En USB 13 12 11 10 9 6 5 SCL SDA
Feather M4 CAN
CANBUS
RST 3V ARf GND A0 A1 A2 A3 A4 A5 SCK MOSI MIS RX TX D4

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 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).

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

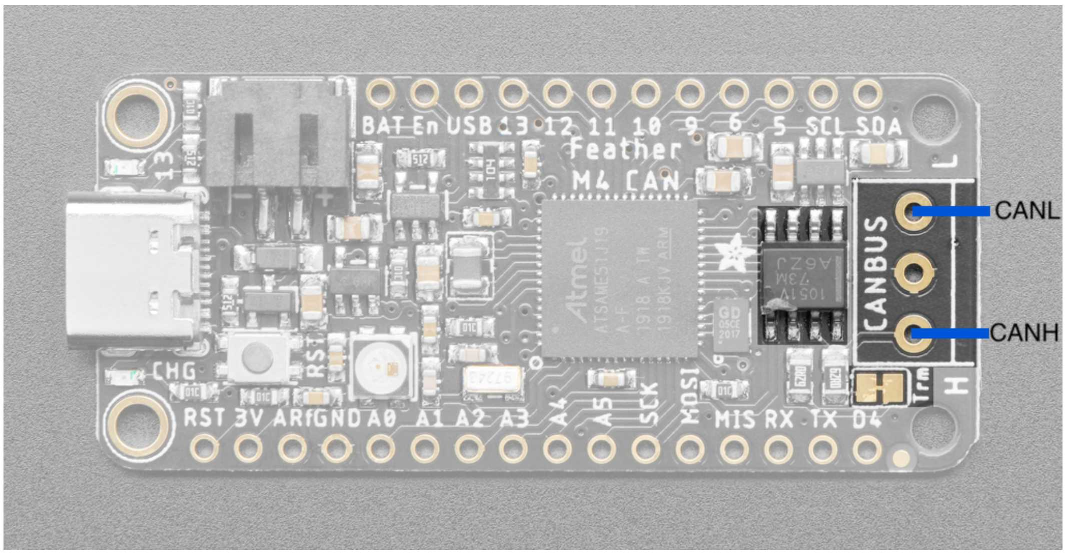

Figure 2 – CAN1 pins

4 Data flow

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

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 `tryToSendReturnStatusFD` 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 13.1 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 to `FIFO0`, see section 15 page 22 for configuring them. Messages that pass the filters are stored in the *Hardware Reception FIFO0* or in the *Hardware Reception FIFO1*. The interrupt service routine transfers the messages from the FIFOi to the *Driver Receive FIFO*i. The size of the *Driver Receive FIFO 0* is 10 by default – see section 14.1 page 21 for changing the default value. Two user methods are available:

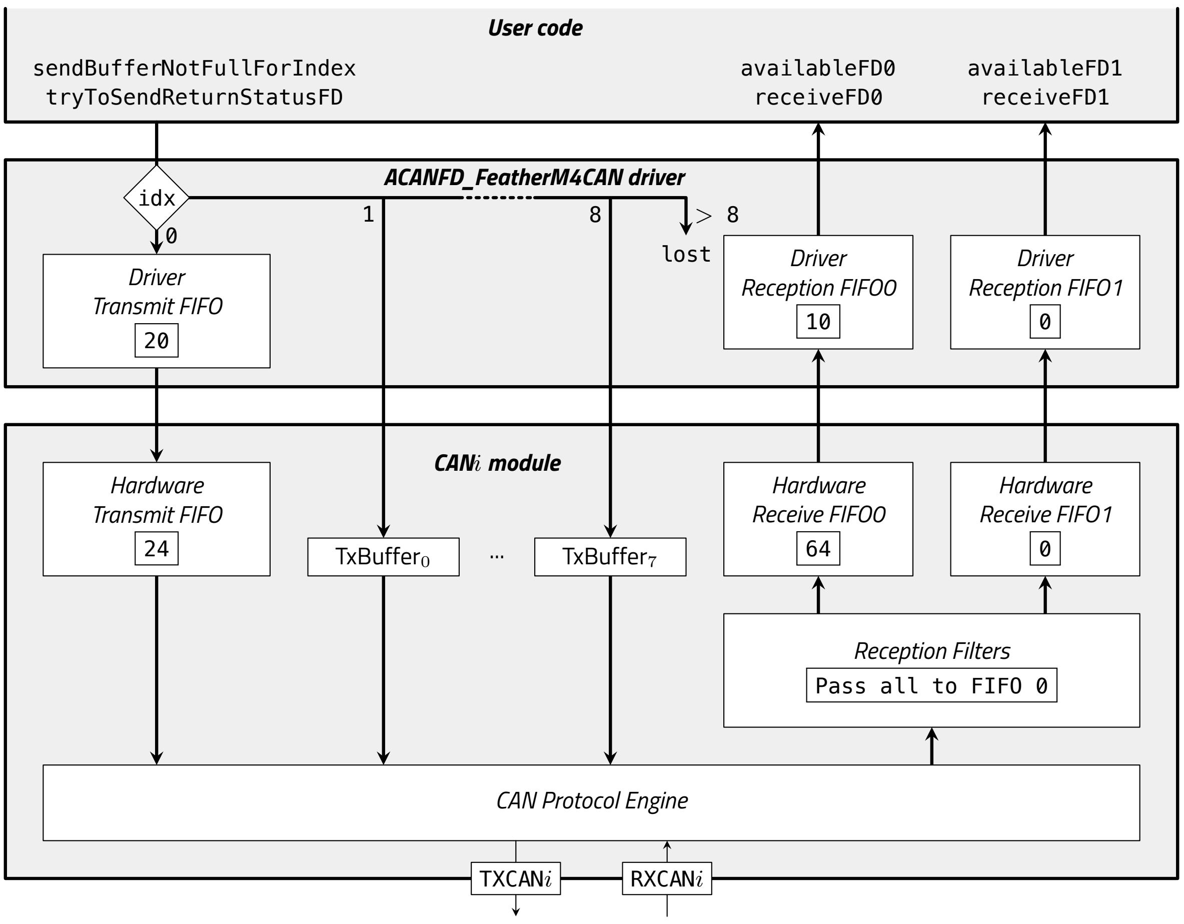

Figure 3 – Message flow in ACANFD_FeatherM4CAN driver and CANi module, default configuration

- 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;
- the `availableFD1` method returns `false` if the *Driver Receive FIFO1* is empty, and `true` otherwise;
- the `receiveFD1` method retrieves messages from the *Driver Receive FIFO1* – see section 14 page 20.

5 A sample sketch: LoopBackDemoCANFD_CAN1

The `LoopBackDemoCANFD_CAN1` sketch is a sample code for introducing the `ACANFD_FeatherM4CAN` library. It demonstrates how to configure the library, 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 section 20.10.1 page 40); the `CAN1` module receives every CANFD frame it sends, and emitted frames can be observed on `CANH`/`CANL` pins.

`ACANFD_FeatherM4CAN` configuration.

```
#define CAN0_MESSAGE_RAM_SIZE (0)
```

```
#define CAN1_MESSAGE_RAM_SIZE (1728)

#include <ACANFD_FeatherM4CAN.h>
```

Before including the ACANFD_FeatherM4CAN library, you should define the CAN0_MESSAGE_RAM_SIZE and the CAN1_MESSAGE_RAM_SIZE macro names.

Each CANFD module uses a private *Message RAM* (section 12 page 17) that is in the first 64 kio of the microcontroller SRAM. Its size depends from the current module configuration, and cannot exceed 4,352 32-bits words (17,408 bytes). Here, CAN0_MESSAGE_RAM_SIZE value is 0, meaning that the CAN0 module is not configured; its TxCAN and RxCAN pins can be freely used for an other function. CAN1_MESSAGE_RAM_SIZE value is 1728, therefore CAN1 module Message RAM has a capacity of 1,728 32-bit words (6,912 bytes), that corresponds to the default configuration.

Note you should include <ACANFD_FeatherM4CAN.h> only once, from the .ino source file. From an other C++ file, you should include <ACANFD_FeatherM4CAN-from-cpp.h>.

If you include <ACANFD_FeatherM4CAN.h> from several files, the can0 and / or can1 variables are multiply-defined, therefore you get a link error.

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 20.10 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 19.2 page 32.

The `pseudoRandomValue` function.

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

```
static uint32_t pseudoRandomValue (void) {
  static uint32_t gSeed = 0 ;
  gSeed = 8253729U * gSeed + 2396403U ;
  return gSeed ;
}
```

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 ACAN2515 driver[3], the ACAN2517 driver[4] and the ACAN2517FD driver[5] contain an identical CANMessage.h header file, enabling using the ACANFD_FeatherM4CAN driver, the 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 this library, the CANMessage class is only used by a CANFDMessage constructor (section 7.3 page 12).

```
class CANMessage {
  public : uint32_t id = 0 ;  // Frame identifier
  public : bool ext = false ; // false -> standard frame, true -> extended frame
  public : bool rtr = false ; // false -> data frame, true -> remote frame
  public : uint8_t idx = 0 ;  // This field is used by the driver
  public : uint8_t len = 0 ;  // Length of data (0 ... 8)
  public : union {
```

[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.
[5]The ACAN2517FD driver is a CANFD driver for the MCP2517FD CAN controller in CANFD mode, https://github.com/pierremolinaro/acan2517FD.

```
    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 processor of the `ATSAME51G19A` is 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 29) or 255 if it does not correspond to any filter;
- 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. The `ACAN2517FD` driver[6] contains an identical `CANFDMessage.h` header file, enabling using the `ACANFD_FeatherM4CAN` driver and the `ACAN2517FD` driver in a same sketch.

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]The ACAN2517FD driver is a CANFD driver for the MCP2517FD CAN controller in CANFD mode, `https://github.com/pierremolinaro/acan2517FD`.

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 ACANFD_FeatherM4CAN::tryToSendReturnStatusFD method. You can use the pad method (see section 7.7 page 14) 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 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. Valid lengths are: 0, 1, ..., 8, 12, 16, 20, 24, 32, 48, 64. 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 13, and false otherwise.

8 Transmit FIFO

The transmit FIFO (see figure 3 page 7) 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 tryToSendReturnStatusFD method with a message whose idx property is zero:

- if the *controller transmit FIFO* is not full, the message is appended to it, and tryToSendReturnStatusFD returns 0;

- otherwise, if the *driver transmit FIFO* is not full, the message is appended to it, and tryToSendReturnStatusFD 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 tryToSendReturnStatusFD 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 tryToSendReturnStatusFD is called, the return value of this call is kTransmitBufferOverflo 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 tryToSendReturnStatusFD 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. You can call the sendBufferNotFullForIndex method (section 13.1 page 19) for testing if a TxBuffer is empty or 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`. **By default, only `FIFO0` is enabled, `FIFO1` is not configured.**

the receive FIFO$_i$ $(0 \leqslant i \leqslant 1$, see figure 3 page 7) is composed by:

- the *hardware receive FIFO$_i$* (in the Message RAM, see section 12 page 17), whose size is between 0 and 64 (default 64 for `CAN0`, 0 for `CAN1`); you can change the default size by setting the `mHardwareRxFIFO`$_i$`Size` property of your `settings` object;
- the *driver receive FIFO$_i$* (in library software), whose size is positive (default 10 for `CAN0`, 0 for `CAN1`); you can change the default size by setting the `mDriverReceiveFIFO`$_i$`Size` property of your `settings` object.

The receive FIFO mechanism ensures sequentiality of reception.

11 Payload size

Hardware transmit FIFO, TxBuffers and hardware receive FIFOs objects are stored in the Message RAM, the details of Message RAM usage computation are presented in section 12 page 17. The size of each object depends on the setting applied to the corresponding FIFO or buffer.

By default, all objects accept frames up to 64 data bytes. The size of each object is 72 bytes. If your application sends and / or receives messages with less than 64 bytes, you can reduce Message RAM size by setting the payload properties of `ACANFD_FeatherM4CAN_Settings` class, as described in table 5. The type of these properties is the `ACANFD_FeatherM4CAN_Settings::Payload` enumeration type, and defines 8 values (table 6).

Object Size specification	**Default value**	**Applies to**
`mHardwareTransmitBufferPayload`	`PAYLOAD_64_BYTES`	Hardware transmit FIFO, TxBuffers
`mHardwareRxFIFO0Payload`	`PAYLOAD_64_BYTES`	Hardware receive FIFO 0

Table 5 – Payload properties of `ACANFD_FeatherM4CAN_Settings` class

Object Size specification	**Handles frames up to**	**Object Size**
`ACANFD_FeatherM4CAN_Settings::PAYLOAD_8_BYTES`	8 bytes	4 words = 16 bytes
`ACANFD_FeatherM4CAN_Settings::PAYLOAD_12_BYTES`	12 bytes	5 words = 20 bytes
`ACANFD_FeatherM4CAN_Settings::PAYLOAD_16_BYTES`	16 bytes	6 words = 24 bytes
`ACANFD_FeatherM4CAN_Settings::PAYLOAD_20_BYTES`	20 bytes	7 words = 28 bytes
`ACANFD_FeatherM4CAN_Settings::PAYLOAD_24_BYTES`	24 bytes	8 words = 32 bytes
`ACANFD_FeatherM4CAN_Settings::PAYLOAD_32_BYTES`	32 bytes	10 words = 40 bytes
`ACANFD_FeatherM4CAN_Settings::PAYLOAD_48_BYTES`	48 bytes	14 words = 56 bytes
`ACANFD_FeatherM4CAN_Settings::PAYLOAD_64_BYTES`	64 bytes	18 words = 72 bytes

Table 6 – `ACANFD_FeatherM4CAN_Settings` object size from payload size specification

11.1 The ACANFD_FeatherM4CAN_Settings::wordCountForPayload static method

```
uint32_t ACANFD_FeatherM4CAN_Settings::wordCountForPayload (const Payload inPayload);
```

This static method returns the object word size for a given payload specification, following table 6.

12 Message RAM

Each CAN module of the ATSAME51G19A uses a *Message RAM* for storing TxBuffers, hardware transmit FIFO, hardware receives FIFO, and reception filters.

The two Message RAM have a width of 32 bits and are part of ATSAME51G19A SRAM, and they should be located in the first 64 kio (0x2000'0000 – 0x2000'FFFF). Their size is less than 4352 words (17,408 bytes).

A message RAM contains[7]:

- standard filters (0-128 elements, 0-128 words);
- extended filters (0-64 elements, 0-128 words);
- receive FIFO 0 (0-64 elements, 0-1152 words);
- receive FIFO 1 (0-64 elements, 0-1152 words);
- Rx Buffers (0-64 elements, 0-1152 words);
- Tx Event FIFO (0-32 elements, 0-64 words);
- Tx Buffers (0-32 elements, 0-576 words);

So its size cannot exceed 4352 words (17,408 bytes).

The current release of this library allows to define only the following elements:

- standard filters (0-128 elements, 0-128 words);
- extended filters (0-64 elements, 0-128 words);
- receive FIFO 0 (0-64 elements, 0-1152 words);
- receive FIFO 1 (0-64 elements, 0-1152 words);
- Tx Buffers (0-32 elements, 0-576 words);

Its size is therefore actually limited to 3,136 words (12,144 bytes).

There are five properties of ACANFD_FeatherM4CAN_Settings class that affect the actual message RAM size:

- the mHardwareRxFIFO0Size property sets the hardware receive FIFO 0 element count (0-64);

[7] See DS60001507G, section 39.9.1 page 1177.

- the mHardwareRxFIFO0Payload property sets the size of the hardware receive FIFO 0 element (table 6);
- the mHardwareRxFIFO1Size property sets the hardware receive FIFO 1 element count (0-64);
- the mHardwareRxFIFO1Payload property sets the size of the hardware receive FIFO 1 element (table 6);
- the mHardwareTransmitTxFIFOSize property sets the hardware transmit FIFO element count (0-32);
- the mHardwareDedicacedTxBufferCount property set the number of dedicaced TxBuffers (0-32);
- the mHardwareTransmitBufferPayload property sets the size of the TxBuffers and hardware transmit FIFO element (table 6).

The ACANFD_FeatherM4CAN::messageRamRequiredSize method returns the required word size.

The ACANFD_FeatherM4CAN::begin method checks the message RAM allocated size is greater or equal to the required size. Otherwise, it raises the error code kMessageRamTooSmall. It checks also the message RAM is in the first 64 kio of the SRAM. Otherwise, it raises the error code kMessageRamNotInFirst64kio.

13 Sending frames: the tryToSendReturnStatusFD method

The ACANFD_FeatherM4CAN::tryToSendReturnStatusFD method sends CAN 2.0B and CANFD frames:

```
uint32_t ACANFD_FeatherM4CAN::tryToSendReturnStatusFD (const CANFDMessage & inMessage);
```

You call the tryToSendReturnStatusFD method for sending a message in the CAN network. Note this function returns before the message is actually sent; this function only adds the message to a transmit buffer. It returns:

- kInvalidMessage (value: 1) if the message is not valid (see section 7.8 page 14);
- kTransmitBufferIndexTooLarge (value: 2) if the idx property value does not specify a valid transmit buffer (see below);
- kTransmitBufferOverflow (value: 3) if the transmit buffer specified by the idx property value is full;
- 0 (no error) if the message has been successfully added to the transmit buffer specified by the idx property value.

The idx property of the message specifies the transmit buffer:

- 0 for the transmit FIFO (section 8 page 14) ;
- 1 ... settings.mHardwareDedicacedTxBufferCount for a dedicaced TxBuffer (section 9 page 15).

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 ACANFD_FeatherM4CAN_Settings::DATA_BITRATE_x$_n$ setting;
- CANFD_WITH_BIT_RATE_SWITCH, with the ACANFD_FeatherM4CAN_Settings::DATA_BITRATE_x1 setting, the frame is sent in CANFD format at arbitration bit rate, and otherwise in CANFD format with bit rate switch.

13.1 Testing a send buffer: the sendBufferNotFullForIndex method

```
bool ACANFD_FeatherM4CAN::sendBufferNotFullForIndex (const uint32_t inTxBufferIndex);
```

This method returns true if the corresponding transmit buffer is not full, and false otherwise (table 7).

inTxBufferIndex	Operation
0	true if the transmit FIFO is not full, and false otherwise
1 ... settings.mHardwareDedicacedTxBufferCount	true if the TxBuffer$i$ is empty, and false if it is full
> settings.mHardwareDedicacedTxBufferCount	false

Table 7 – Value returned by the sendBufferNotFullForIndex method

13.2 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 uint32_t sendStatus = can0.tryToSendReturnStatusFD (message) ;
    if (sendStatus == 0) {
      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 uint32_t sendStatus = can0.tryToSendReturnStatusFD (message) ;
    if (sendStatus == 0) {
      gSendMessage = false ;
    }
  }
  ...
}
```

14 Retrieving received messages using the `receiveFD`i method

```
bool ACANFD_FeatherM4CAN::receiveFD0 (CANFDMessage & outMessage) ;
bool ACANFD_FeatherM4CAN::receiveFD1 (CANFDMessage & outMessage) ;
```

If the receive FIFO i is not empty, the oldest message is removed, assigned to `outMessage`, and the method returns `true`. If the receive FIFO i is empty, the method returns `false`.

This is a basic example:

```
void loop () {
  CANFDMessage message ;
  if (can0.receiveFD0 (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 (can0.receiveFD0 (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 FIFO i size

By default, the driver receive FIFO 0 size is 10 and the driver receive FIFO 1 size is 0. You can change them by setting the mDriverReceiveFIFO0Size property and the mDriverReceiveFIFO1Size property of settings variable before calling the begin method:

```
ACANFD_FeatherM4CAN_Settings settings (125 * 1000,
                                       DataBitRateFactor::x4) ;
settings.mDriverReceiveFIFO0Size = 100 ;
const uint32_t errorCode = can0.begin (settings) ;
...
```

As the size of CANFDMessage class is 72 bytes, the actual size of the driver receive FIFO 0 is the value of settings.mDriverReceiveFIFO0Size $*$ 72, and the actual size of the driver receive FIFO 1 is the value of settings.mDriverReceiveFIFO1Size $*$ 72.

14.2 The driverReceiveFIFOiSize method

The `driverReceiveFIFO`i`Size` method returns the size of the driver FIFO i, that is the value of the `mDriver-ReceiveFIFO`i`Size` property of `settings` variable when the the `begin` method is called.

```
const uint32_t s = can0.driverReceiveFIFO0Size () ;
```

14.3 The driverReceiveFIFOiCount method

The `driverReceiveFIFO`i`Count` method returns the current number of messages in the driver receive FIFO i.

```
const uint32_t n = can0.driverReceiveFIFO0Count () ;
```

14.4 The driverReceiveFIFOiPeakCount method

The `driverReceiveFIFO`i`PeakCount` method returns the peak value of message count in the driver receive FIFO i.

```
const uint32_t max = can0.driverReceiveFIFO0PeakCount () ;
```

If an overflow occurs, further calls of `can0.driverReceiveFIFO`i`PeakCount ()` return `can0.driverReceiveFIFO`i`Size ()`+1.

14.5 The resetDriverReceiveFIFOiPeakCount method

The `resetDriverReceiveFIFO`i`PeakCount` method assign the current count to the peak value.

```
can0.resetDriverReceiveFIFO0PeakCount () ;
```

15 Acceptance filters

The microcontroller bases the filtering of the received frames on the nature of their identifier: standard or extended. It is not possible to filter by length or by CAN2.0B / CANFD format. The only possibility is to reject all remote frames.

15.1 Acceptance filters for standard frames

for an example sketch, see `LoopBackDemoCANFD_CAN1_StandardFilters`.

You have three ways to act on standard frame filtering:

- setting the `mDiscardReceivedStandardRemoteFrames` property of the `ACANFD_FeatherM4CAN_Settings` class discards every received remote frame (it is `false` by default);

- the `mNonMatchingStandardFrameReception` property value of the `ACANFD_FeatherM4CAN_Settings` class is applied to every standard frame that do not match any filter; its value can be `FIFO0` (default), `FIFO1` or `REJECT`;
- define standard filters (as described from section 15.1.1 page 23), up to 128, none by default.

The standard frame filtering is illustrated by figure 4.

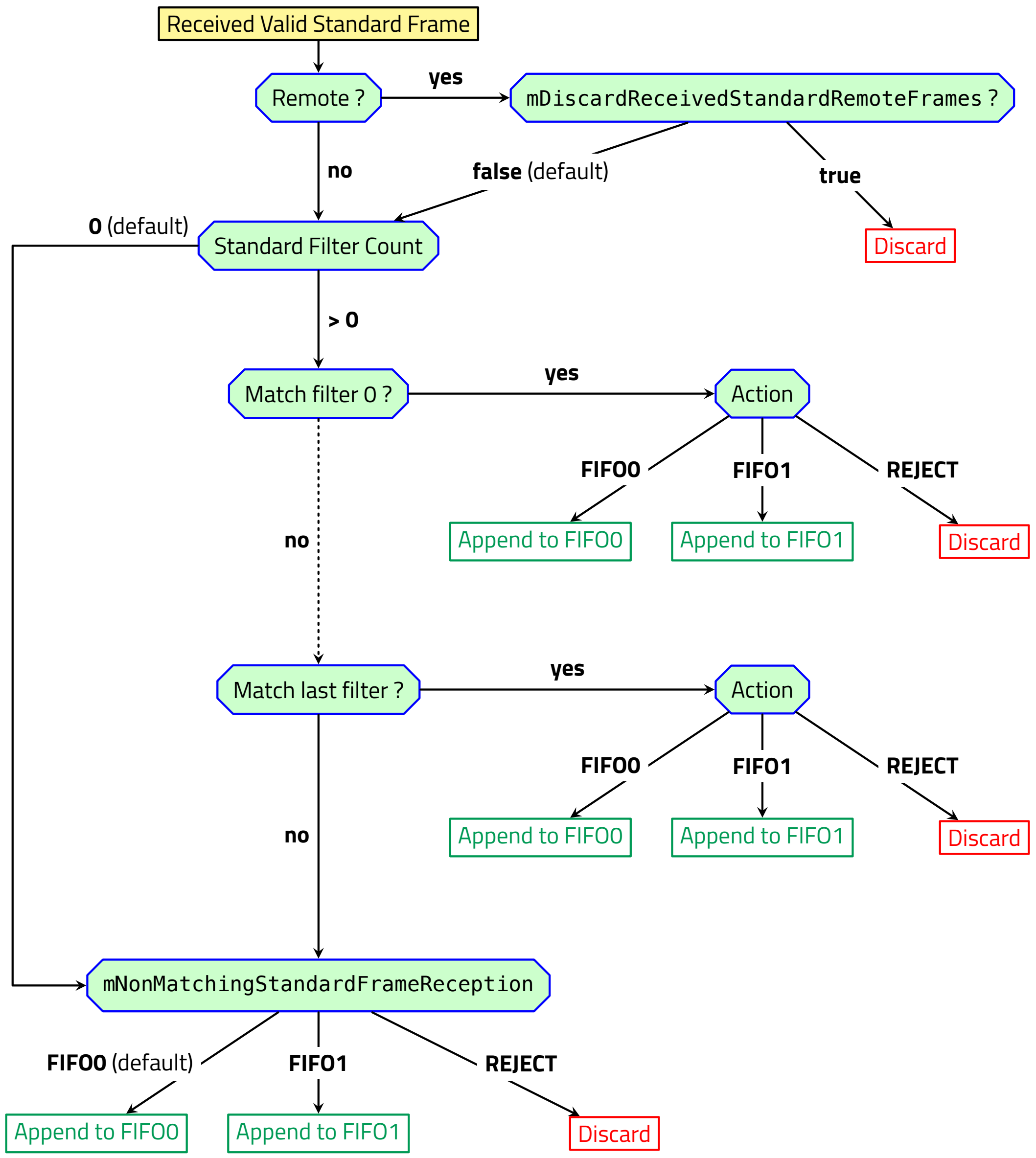

Figure 4 – Standard frame filtering

15.1.1 Defining standard frame filters

```
ACANFD_FeatherM4CAN_Settings settings (..., ...) ;
...
```

```
  ACANFD_FeatherM4CAN::StandardFilters standardFilters ;
  standardFilters.addSingle (0x55, ACANFD_FeatherM4CAN_FilterAction::FIFO0) ;
  ...
//--- Reject standard frames that do not match any filter
  settings.mNonMatchingStandardFrameReception = ACANFD_FeatherM4CAN_FilterAction::REJECT;
  ...
  const uint32_t errorCode = can1.beginFD (settings, standardFilters) ;
  ...
```

The `ACANFD_FeatherM4CAN::StandardFilters` class handles a standard frame filter list. Default constructor constructs an empty list. For appending filters, use the `addSingle` (section 15.1.2 page 24), `addDual` (section 15.1.3 page 24), addRange (section 15.1.4 page 24) or `addClassic` (section 15.1.5 page 25) methods. Then, add the `standardFilters` as second argument of `beginFD` call.

Note. Do not forget to set `settings.mNonMatchingStandardFrameReception` to REJECT, otherwise all frames rejected by the filters are appended to FIFO 0 (see figure 4 for detail).

15.1.2 Add single filter

```
bool StandardFilters::addSingle (const uint16_t inIdentifier,
                                 const ACANFD_FeatherM4CAN_FilterAction inAction,
                                 const ACANFDCallBackRoutine inCallBack = nullptr) ;
```

This filter is valid if `inIdentifier` is lower or equal to `0x7FF`. The method returns `true` if the filter is valid, and `false` otherwise. If the filter is valid, this method appends a filter that matches if the received standard frame identifier is equal to `inIdentifier`. If the filter is not valid, the filter is not appended.

The last argument is optional and associates a callback routine to the filter. See section 16 page 29.

15.1.3 Add dual filter

```
bool StandardFilters::addDual (const uint16_t inIdentifier1,
                               const uint16_t inIdentifier2,
                               const ACANFD_FeatherM4CAN_FilterAction inAction,
                               const ACANFDCallBackRoutine inCallBack = nullptr) ;
```

This filter is valid if `inIdentifier1` is lower or equal to `0x7FF` and `inIdentifier2` is lower or equal to `0x7FF`. The method returns `true` if the filter is valid, and `false` otherwise. If the filter is valid, this method appends a filter that matches if the received standard frame identifier is equal to `inIdentifier1` or is equal to `inIdentifier2`. If the filter is not valid, the filter is not appended.

The last argument is optional and associates a callback routine to the filter. See section 16 page 29.

15.1.4 Add range filter

```
bool StandardFilters::addRange (const uint16_t inIdentifier1,
                                         const uint16_t inIdentifier2,
                                         const ACANFD_FeatherM4CAN_FilterAction inAction,
                                         const ACANFDCallBackRoutine inCallBack = nullptr) ;
```

This filter is valid if `inIdentifier1` is lower or equal to `inIdentifier2` and `inIdentifier2` is lower or equal to `0x7FF`. The method returns `true` if the filter is valid, and `false` otherwise. If the filter is valid, this method appends a filter that matches if the received standard frame identifier is greater or equal to `inIdentifier1` and is lower or equal to `inIdentifier2`. If the filter is not valid, the filter is not appended.

The last argument is optional and associates a callback routine to the filter. See section 16 page 29.

15.1.5 Add classic filter

```
bool StandardFilters::addClassic (const uint16_t inIdentifier,
                                   const uint16_t inMask,
                                   const ACANFD_FeatherM4CAN_FilterAction inAction,
                                   const ACANFDCallBackRoutine inCallBack = nullptr) ;
```

This filter is valid if all the following conditions are met:

- `inIdentifier` is lower or equal to `0x7FF`;
- `inMask` is lower or equal to `0x7FF`;
- (`inIdentifier` & `inMask`) is equal to `inIdentifier`.

The method returns `true` if the filter is valid, and `false` otherwise. If the filter is valid, this method appends a filter that matches if the received standard frame identifier verifies (`receivedFrameIdentifier` & `inMask`) is equal to `inIdentifier`. That means:

- if a mask bit is a `1`, the received standard frame identifier corresponding bit should match the `inIdentifier` corresponding bit;
- if a mask bit is a `0`, the received standard frame identifier corresponding bit can have any value, the `inIdentifier` corresponding bit should be `0`.

If the filter is not valid, the filter is not appended.

The last argument is optional and associates a callback routine to the filter. See section 16 page 29.

For example:

```
standardFilters.addClassic (0x405, 0x7D5, ACANFD_FeatherM4CAN_FilterAction::FIFO0) ;
```

This filter is valid because (`0x405` & `0x7D5`) is equal to `0x405`.

	10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`inIdentifier`: `0x405`	1	0	0	0	0	0	0	0	1	0	1
`inMask`: `0x7D5`	1	1	1	1	1	0	1	0	1	0	1
Matching identifiers	1	0	0	0	0	x	0	x	1	x	1

Therefore there are 8 matching identifiers: `0x405`, `0x407`, `0x40B`, `0x40F`, `0x425`, `0x427`, `0x42B`, `0x42F`.

15.2 Acceptance filters for extended frames

for an example sketch, see `LoopBackDemoCANFD_CAN1_ExtendedFilters`.

You have three ways to act on extended frame filtering:

- setting the `mDiscardReceivedExtendedRemoteFrames` property of the `ACANFD_FeatherM4CAN_Settings` class discards every received remote frame (it is `false` by default);
- the `mNonMatchingExtendedFrameReception` property value of the `ACANFD_FeatherM4CAN_Settings` class is applied to every extended frame that do not match any filter; its value can be `FIFO0` (default), `FIFO1` or `REJECT`;
- define extended filters (as described from section 15.2.1 page 26), up to 128, none by default.

The extended frame filtering is illustrated by figure 5.

15.2.1 Defining extended frame filters

```
  ACANFD_FeatherM4CAN_Settings settings (..., ...) ;
  ...
  ACANFD_FeatherM4CAN::ExtendedFilters extendedFilters ;
  extendedFilters.addSingle (0x55, ACANFD_FeatherM4CAN_FilterAction::FIFO0) ;
  ...
//--- Reject extended frames that do not match any filter
  settings.mNonMatchingExtendedFrameReception = ACANFD_FeatherM4CAN_FilterAction::REJECT;
  ...
  const uint32_t errorCode = can1.beginFD (settings, extendedFilters) ;
  ...
```

The `ACANFD_FeatherM4CAN::ExtendedFilters` class handles an extended frame filter list. Default constructor constructs an empty list. For appending filters, use the `addSingle` (section 15.2.2 page 26), `addDual` (section 15.2.3 page 27), `addRange` (section 15.2.4 page 28) or `addClassic` (section 15.2.5 page 28) methods. Then, add the `ExtendedFilters` as second argument of `beginFD` call.

Note. Do not forget to set `settings.mNonMatchingExtendedFrameReception` to `REJECT`, otherwise all frames rejected by the filters are appended to FIFO 0 (see figure 5 for detail).

15.2.2 Add single filter

```
bool ExtendedFilters::addSingle (const uint32_t inIdentifier,
                                 const ACANFD_FeatherM4CAN_FilterAction inAction,
                                 const ACANFDCallBackRoutine inCallBack = nullptr) ;
```

This filter is valid if `inIdentifier` is lower or equal to `0x1FFF_FFFF`. The method returns `true` if the filter is valid, and `false` otherwise. If the filter is valid, this method appends a filter that matches if the received extended frame identifier is equal to `inIdentifier`. If the filter is not valid, the filter is not appended.

The last argument is optional and associates a callback routine to the filter. See section 16 page 29.

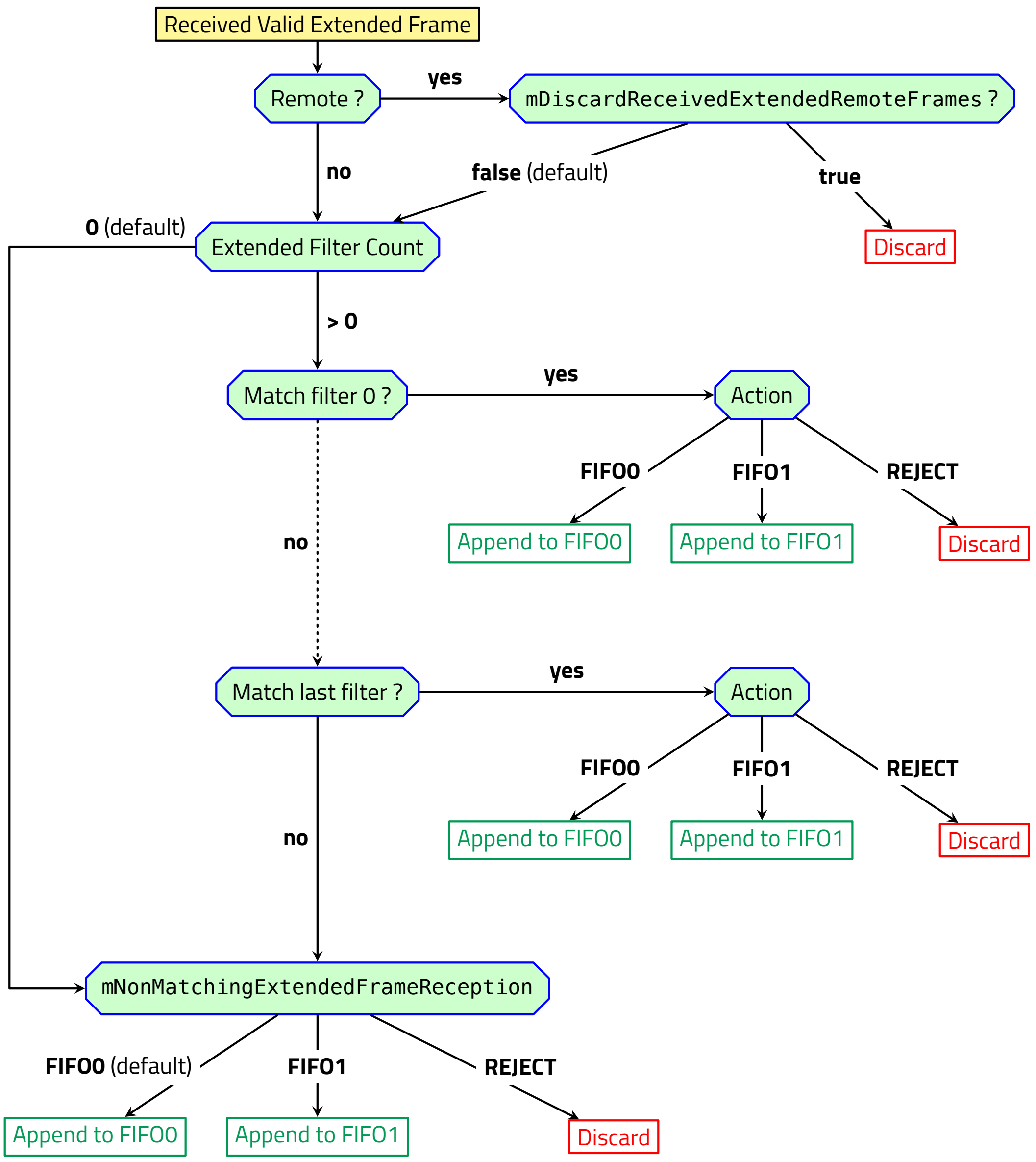

Figure 5 – Extended frame filtering

15.2.3 Add dual filter

```
bool ExtendedFilters::addDual (const uint32_t inIdentifier1,
                               const uint32_t inIdentifier2,
                               const ACANFD_FeatherM4CAN_FilterAction inAction,
                               const ACANFDCallBackRoutine inCallBack = nullptr) ;
```

This filter is valid if `inIdentifier1` is lower or equal to `0x1FFF_FFFF` and `inIdentifier2` is lower or equal to `0x1FFF_FFFF`. The method returns `true` if the filter is valid, and `false` otherwise. If the filter is valid, this method appends a filter that matches if the received extended frame identifier is equal to `inIdentifier1` or is equal to `inIdentifier2`. If the filter is not valid, the filter is not appended.

The last argument is optional and associates a callback routine to the filter. See section 16 page 29.

15.2.4 Add range filter

```
bool ExtendedFilters::addRange (const uint32_t inIdentifier1,
                                const uint32_t inIdentifier2,
                                const ACANFD_FeatherM4CAN_FilterAction inAction,
                                const ACANFDCallBackRoutine inCallBack = nullptr) ;
```

This filter is valid if `inIdentifier1` is lower or equal to `inIdentifier2` and `inIdentifier2` is lower or equal to `0x1FFF_FFFF`. The method returns `true` if the filter is valid, and `false` otherwise. If the filter is valid, this method appends a filter that matches if the received extended frame identifier is greater or equal to `inIdentifier1` and is lower or equal to `inIdentifier2`. If the filter is not valid, the filter is not appended.

The last argument is optional and associates a callback routine to the filter. See section 16 page 29.

15.2.5 Add classic filter

```
bool ExtendedFilters::addClassic (const uint32_t inIdentifier,
                                  const uint32_t inMask,
                                  const ACANFD_FeatherM4CAN_FilterAction inAction,
                                  const ACANFDCallBackRoutine inCallBack = nullptr) ;
```

This filter is valid if all the following conditions are met:

- `inIdentifier` is lower or equal to `0x1FFF_FFFF`;
- `inMask` is lower or equal to `0x1FFF_FFFF`;
- (`inIdentifier` & `inMask`) is equal to `inIdentifier`.

The method returns `true` if the filter is valid, and `false` otherwise. If the filter is valid, this method appends a filter that matches if the received extended frame identifier verifies (`receivedFrameIdentifier` & `inMask`) is equal to `inIdentifier`. That means:

- if a mask bit is a 1, the received extended frame identifier corresponding bit should match the `inIdentifier` corresponding bit;
- if a mask bit is a `0`, the received extended frame identifier corresponding bit can have any value, the `inIdentifier` corresponding bit should be `0`.

If the filter is not valid, the filter is not appended.

The last argument is optional and associates a callback routine to the filter. See section 16 page 29.

For example:

```
extendedFilters.addClassic (0x6789, 0x1FFF67BD, ACANFD_FeatherM4CAN_FilterAction::FIFO0) ;
```

This filter is valid because (`0x6789` & `0x1FFF67BD`) is equal to `0x6789`.

	28 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`inIdentifier`: `0x6789`	0	0	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1
`inMask`: `0x1FFF67BD`	1	0	1	1	0	0	1	1	1	1	0	1	1	1	1	0	1
Matching identifiers	0	x	1	1	x	x	1	1	1	1	x	1	1	1	0	x	1

Therefore there are 32 matching identifiers.

16 The `dispatchReceivedMessage` method

Sample sketch: the `LoopBackDemoCANFD_CAN1_dispatch` sketch shows how using the `dispatchReceivedMessage` method.

Instead of calling the `receiveFD0` and the `receiveFD1` methods, call the `dispatchReceivedMessage` method in your `loop` function. For every message extracted from `FIFO0` and `FIFO1`, it calls the callback function associated with the corresponding filter.

If you have not defined any filter, do not use this function, call the `receiveFD0` and / or the `receiveFD1` methods.

```
void loop () {
  can1.dispatchReceivedMessage () ; // Do not call can1.receiveFD0, can1.receiveFD1 any more
  ...
}
```

The `dispatchReceivedMessage` method handles one `FIFO0` message and one `FIFO1` message on each call. Specifically:

- if `FIFO0` and `FIFO01` are both empty, it returns `false`;
- if `FIFO0` is not empty, its oldest message is extracted and its associated callback is called; then, if `FIFO1` is not empty, its oldest message is extracted and its associated callback is called; the `true` value is returned.

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

The return value can used for emptying and dispatching all received messages:

```
void loop () {
  while (can1.dispatchReceivedMessage ()) {
  }
  ...
}
```

16.1 Dispatching non matching standard frames

Following the figure 4 page 23, non matching standard frames are stored in `FIFO0` if `mNonMatchingStandardFrameReception` is equal to `FIFO0`, or in `FIFO1` if `mNonMatchingStandardFrameReception` is equal to

FIFO1. As theses frames do not correspond to a filter, there is no associated callback function by default. Therefore, they are lost when the dispatchReceivedMessage method is called.

You can assign a callback function to the mNonMatchingStandardMessageCallBack property of the ACANFD-_FeatherM4CAN_Settings class. This provides a callback function to non matching standard frames, so they are dispatched by a the dispatchReceivedMessage method. By default, mNonMatchingStandardMessageCallBack value is nullptr.

If mNonMatchingStandardFrameReception is equal to REJECT, the mNonMatchingStandardMessageCallBack value is never used.

16.2 Dispatching non matching extended frames

Following the figure 5 page 27, non matching extended frames are stored in FIFO0 if mNonMatchingExtendedFrameReception is equal to FIFO0, or in FIFO1 if mNonMatchingExtendedFrameReception is equal to FIFO1. As theses frames do not correspond to a filter, there is no associated callback function by default. Therefore, they are lost when the dispatchReceivedMessage method is called.

You can assign a callback function to the mNonMatchingExtendedMessageCallBack property of the ACANFD-_FeatherM4CAN_Settings class. This provides a callback function to non matching extended frames, so they are dispatched by a the dispatchReceivedMessage method. By default, mNonMatchingExtendedMessageCallBack value is nullptr.

If mNonMatchingExtendedFrameReception is equal to REJECT, the mNonMatchingExtendedMessageCallBack value is never used.

17 The dispatchReceivedMessageFIFO0 method

The dispatchReceivedMessageFIFO0 method dispatches the messages stored in the FIFO0. The messages stored is FIFO1 are retrieved using the receiveFD1 method.

```
void loop () {
  can1.dispatchReceivedMessageFIFO0 () ; // Do not call can1.receiveFD0 any more
  CANFDMessage ;
  if (can1.receiveFD1 (message)) {
    ... handle FIFO1 message ...
  }
  ...
}
```

Instead of calling the receiveFD0 method, call the dispatchReceivedMessageFIFO0 method in your loop function. For every message extracted from FIFO0, it calls the callback function associated with the corresponding filter.

If you have not defined any filter that targets the FIFO0, do not use this function (messages will be not dispatched and therefore lost), call the receiveFD0 method.

The `dispatchReceivedMessageFIFO0` method handles one `FIFO0` message on each call. Specifically:

- if `FIFO0` is empty, it returns `false`;
- if `FIFO0` is not empty, its oldest message is extracted and its associated callback is called and the `true` value is returned.

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

The return value can used for emptying and dispatching all received messages:

```
void loop () {
  while (can1.dispatchReceivedMessageFIFO0 ()) {
  }
  CANFDMessage ;
  if (can1.receiveFD1 (message)) {
    ... handle FIFO1 message ...
  }
  ...
}
```

18 The `dispatchReceivedMessageFIFO1` method

The `dispatchReceivedMessageFIFO1` method dispatches the messages stored in the `FIFO1`. The messages stored is `FIFO0` are retrieved using the `receiveFD0` method.

```
void loop () {
  can1.dispatchReceivedMessageFIFO1 () ; // Do not call can1.receiveFD1 any more
  CANFDMessage ;
  if (can1.receiveFD0 (message)) {
    ... handle FIFO0 message ...
  }
  ...
}
```

Instead of calling the `receiveFD1` method, call the `dispatchReceivedMessageFIFO1` method in your `loop` function. For every message extracted from `FIFO1`, it calls the callback function associated with the corresponding filter.

If you have not defined any filter that targets the `FIFO1`, do not use this function (messages will be not dispatched and therefore lost), call the `receiveFD1` method.

The `dispatchReceivedMessageFIFO1` method handles one `FIFO1` message on each call. Specifically:

- if `FIFO1` is empty, it returns `false`;
- if `FIFO1` is not empty, its oldest message is extracted and its associated callback is called and the `true` value is returned.

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

The return value can used for emptying and dispatching all received messages:

```
void loop () {
  while (can1.dispatchReceivedMessageFIFO1 ()) {
  }
  CANFDMessage ;
  if (can1.receiveFD0 (message)) {
    ... handle FIFO0 message ...
  }
  ...
}
```

19 The `ACANFD_FeatherM4CAN::beginFD` method reference

19.1 The prototypes

```
uint32_t ACANFD_FeatherM4CAN::beginFD (const ACANFD_FeatherM4CAN_Settings & inSettings,
                        const StandardFilters & inStandardFilters = StandardFilters (),
                        const ExtendedFilters & inExtendedFilters = ExtendedFilters ()) ;

uint32_t ACANFD_FeatherM4CAN::beginFD (const ACANFD_FeatherM4CAN_Settings & inSettings,
                               const ExtendedFilters & inExtendedFilters) ;
```

The first argument is a `ACANFD_FeatherM4CAN_Settings` instance that defines the settings.

The second one is optional, and specifies the standard filter list (see section 15.1 page 22). By default, the standard filter list is empty.

The third one is optional, and specifies the extended filter list (see section 15.2 page 26). By default, the extended filter list is empty.

19.2 The error codes

The `ACANFD_FeatherM4CAN::beginFD` 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 `ACANFD_FeatherM4CAN` class defines static constants for naming errors. Bits 0 to 16 denote a bit configuration error, see table 10 page 39.

19.2.1 The `kTxBufferCountGreaterThan32` error code

There are 32 available TxBuffers, for hardware transmit FIFO and dedicaced TxBuffers. Therefore, the sum of `settings.mHardwareDedicacedTxBufferCount` and `settings.mHardwareTransmitTxFIFOSize` should be lower or equal to 32.

Bit	Code	Static constant Name	Comment
0	0x1	kBitRatePrescalerIsZero	See table 10 page 39
…	…	….	See table 10 page 39
16	0x1_0000	kDataSJWIsGreaterThanPhaseSegment2	See table 10 page 39
20	0x10_0000	kMessageRamTooSmall	See section 12 page 17
21	0x20_0000	kMessageRamNotInFirst64kio	See section 12 page 17
22	0x40_0000	kHardwareRxFIFO0SizeGreaterThan64	settings.mHardwareRxFIFO0Size > 64
23	0x80_0000	kHardwareTransmitFIFOSizeGreaterThan32	settings.mHardwareTransmitTxFIFOSize > 32
24	0x100_0000	kDedicacedTransmitTxBufferCountGreaterThan30	settings.mHardwareDedicacedTxBufferCount > 30
25	0x200_0000	kTxBufferCountGreaterThan32	See section 19.2.1 page 32
26	0x400_0000	kHardwareTransmitFIFOSizeLowerThan2	See settings.mHardwareTransmitTxFIFOSize < 2
27	0x800_0000	kHardwareRxFIFO1SizeGreaterThan64	settings.mHardwareRxFIFO1Size > 64
28	0x1000_0000	kStandardFilterCountGreaterThan128	More than 128 standard filters, see section 15.1 page 22
29	0x2000_0000	kExtendedFilterCountGreaterThan128	More than 128 extended filters, see section 15.2 page 26

Table 8 – The ACANFD_FeatherM4CAN::beginFD method error code bits

20 ACANFD_FeatherM4CAN_Settings class reference

Note. The ACANFD_FeatherM4CAN_Settings class is not Arduino specific. You can compile it on your desktop computer with your favorite C++ compiler.

20.1 The ACANFD_FeatherM4CAN_Settings constructor: computation of the CAN bit settings

```
void setup () {
ACANFD_FeatherM4CAN_Settings::
ACANFD_FeatherM4CAN_Settings (const uint32_t inDesiredArbitrationBitRate,
                              const DataBitRateFactor inDataBitRateFactor,
                              const uint32_t inTolerancePPM = 1000) ;
```

The constructor of the ACANFD_FeatherM4CAN_Settings has two mandatory arguments: 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
  ACANFD_FeatherM4CAN_Settings settings (1000 * 1000, DataBitRateFactor::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 DataBitRateFactor::x1 as data bit rate factor.

There are 313 exact arbitration / data bit rate combinations (table 9 page 34).

Arbitration Bit Rate	Valid Data Rate factors	Arbitration Bit Rate	Valid Data Rate factors
5 000	x8 x10	6 000	x8 x10
6 250	x5 x6 x8 x10	6 400	x10
7 500	x5 x8 x10	7 680	x10
8 000	x5 x6 x8 x10	9 375	x4 x5 x8 x10
9 600	x5 x8 x10	10 000	x4 x5 x6 x8 x10
12 000	x4 x5 x8 x10	12 500	x3 x4 x5 x6 x8 x10
12 800	x5 x6 x10	15 000	x4 x5 x8 x10
15 360	x5	15 625	x2 x3 x4 x6 x8
16 000	x3 x4 x5 x6 x8 x10	18 750	x2 x4 x5 x8 x10
19 200	x4 x5 x10	20 000	x2 x3 x4 x5 x6 x8 x10
24 000	x2 x4 x5 x8 x10	25 000	x2 x3 x4 x5 x6 x8 x10
25 600	x3 x5	30 000	x2 x4 x5 x8 x10
31 250	x1 x2 x3 x4 x6 x8	32 000	x2 x3 x4 x5 x6 x10
37 500	x1 x2 x4 x5 x8 x10	38 400	x2 x5 x10
40 000	x1 x2 x3 x4 x5 x6 x8 x10	46 875	x1 x2 x4 x8
48 000	x1 x2 x4 x5 x8 x10	50 000	x1 x2 x3 x4 x5 x6 x8 x10
60 000	x1 x2 x4 x5 x8 x10	62 500	x1 x2 x3 x4 x6 x8
64 000	x1 x2 x3 x5 x6 x10	75 000	x1 x2 x4 x5 x8 x10
76 800	x1 x5	80 000	x1 x2 x3 x4 x5 x6 x8 x10
93 750	x1 x2 x4 x8	96 000	x1 x2 x4 x5 x10
100 000	x1 x2 x3 x4 x5 x6 x8 x10	120 000	x1 x2 x4 x5 x8 x10
125 000	x1 x2 x3 x4 x6 x8	128 000	x1 x3 x5
150 000	x1 x2 x4 x5 x8 x10	160 000	x1 x2 x3 x4 x5 x6 x10
187 500	x1 x2 x4 x8	192 000	x1 x2 x5 x10
200 000	x1 x2 x3 x4 x5 x6 x8 x10	240 000	x1 x2 x4 x5 x8 x10
250 000	x1 x2 x3 x4 x6 x8	300 000	x1 x2 x4 x5 x8 x10
320 000	x1 x2 x3 x5 x6 x10	375 000	x1 x2 x4 x8
384 000	x1 x5	400 000	x1 x2 x3 x4 x5 x6 x8 x10
480 000	x1 x2 x4 x5 x10	500 000	x1 x2 x3 x4 x6 x8
600 000	x1 x2 x4 x5 x8 x10	640 000	x1 x3 x5
750 000	x1 x2 x4 x8	800 000	x1 x2 x3 x4 x5 x6 x10
960 000	x1 x2 x5 x10	1 000 000	x1 x2 x3 x4 x6 x8

Table 9 – The 313 exact bit rates

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 () {
  ...
  ACANFD_FeatherM4CAN_Settings settings (571428, DataBitRateFactor::x7) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (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 third argument of ACANFD_FeatherM4CAN_Settings constructor. For example, with an arbitration bit rate equal to 727 kbit/s:

```
void setup () {
  ...
  ACANFD_FeatherM4CAN_Settings settings (727 * 1000,
                                         DataBitRateFactor::x1,
                                         100) ; // 100 ppm
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (false)
  Serial.print ("actual arbitration bit rate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; // 727272 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 375 ppm
  ...
}
```

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

```
void setup () {
  ...
  ACANFD_FeatherM4CAN_Settings settings (500 * 1000,
                                         DataBitRateFactor::x1,
                                         0) ; // Max distance is 0 ppm
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (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 () {
  ...
  ACANFD_FeatherM4CAN_Settings settings (423 * 1000, DataBitRateFactor::x3) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (false)
  Serial.print ("Actual Arbitration Bit Rate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  421 052 bit/s
  Serial.print ("Actual Data Bit Rate: ") ;
  Serial.println (settings.actualDataBitRate ()) ; //  1 263 157 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 4 603 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 () {
  ...
  ACANFD_FeatherM4CAN_Settings settings (423 * 1000, DataBitRateFactor::x3) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (false)
  Serial.print ("Actual Arbitration Bit Rate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  421 052 bit/s
  Serial.print ("Actual Data Bit Rate: ") ;
  Serial.println (settings.actualDataBitRate ()) ; //  1 263 157 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 4 603 ppm
  Serial.print ("Bit rate prescaler: ") ;
  Serial.println (settings.mBitRatePrescaler) ; // BRP = 1
  Serial.print ("Arbitration Phase segment 1: ") ;
  Serial.println (settings.mArbitrationPhaseSegment1) ; // PS1 = 22
  Serial.print ("Arbitration Phase segment 2: ") ;
  Serial.println (settings.mArbitrationPhaseSegment2) ; // PS2 = 10
  Serial.print ("Arbitration Resynchronization Jump Width: ") ;
  Serial.println (settings.mArbitrationSJW) ; // SJW = 10
  Serial.print ("Arbitration Sample Point: ") ;
  Serial.println (settings.arbitrationSamplePointFromBitStart ()) ; // 69, meaning 69%
  Serial.print ("Data Phase segment 1: ") ;
  Serial.println (settings.mDataPhaseSegment1) ; // PS1 = 22
  Serial.print ("Data Phase segment 2: ") ;
  Serial.println (settings.mDataPhaseSegment2) ; // PS2 = 10
  Serial.print ("Data Resynchronization Jump Width: ") ;
  Serial.println (settings.mDataSJW) ; // SJW = 10
  Serial.print ("Data Sample Point: ") ;
  Serial.println (settings.dataSamplePointFromBitStart ()) ; // 69, meaning 59%
  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 () {
  ...
  ACANFD_FeatherM4CAN_Settings settings (500 * 1000, DataBitRateFactor::x1) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (true)
  settings.mArbitrationPhaseSegment1 -= 4 ; // 32 -> 28: safe, 1 <= PS1 <= 256
  settings.mArbitrationPhaseSegment2 += 4 ; // 15 -> 19: safe, 1 <= PS2 <= 128
  settings.mArbitrationSJW += 4           ; // 15 -> 19: safe, 1 <= SJW <= PS2
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 58, meaning 58%
  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 ATSAME51G19A constraints are:

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

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

Resulting actual bit rates are given by (SYSCLK = 48 MHz):

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

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

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

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 () {
  ...
  ACANFD_FeatherM4CAN_Settings settings (500 * 1000, DataBitRateFactor::x2) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (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 10.

The `ACANFD_FeatherM4CAN_Settings` class defines static constant properties that can be used as mask error. For example:

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

20.3 The `actualArbitrationBitRate` method

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

```
void setup () {
  ...
  ACANFD_FeatherM4CAN_Settings settings (440 * 1000, DataBitRateFactor::x1) ;
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (false)
  Serial.print ("actual arbitration bit rate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  444,444 bit/s
  ...
}
```

Bit	Code	Error Name	Error
0	0x1	kBitRatePrescalerIsZero	mBitRatePrescaler == 0
1	0x2	kBitRatePrescalerIsGreaterThan32	mBitRatePrescaler > 32
2	0x4	kArbitrationPhaseSegment1IsZero	mArbitrationPhaseSegment1 == 0
3	0x8	kArbitrationPhaseSegment1IsGreaterThan256	mArbitrationPhaseSegment1 > 256
4	0x10	kArbitrationPhaseSegment2IsLowerThan2	mArbitrationPhaseSegment2 < 2
5	0x20	kArbitrationPhaseSegment2IsGreaterThan128	mArbitrationPhaseSegment2 > 128
6	0x40	kArbitrationSJWIsZero	mArbitrationSJW == 0
7	0x80	kArbitrationSJWIsGreaterThan128	mArbitrationSJW > 128
8	0x100	kArbitrationSJWIsGreaterThanPhaseSegment2	mArbitrationSJW > mArbitrationPhaseSegment2
9	0x200	kArbitrationPhaseSegment1Is1AndTripleSampling	(mArbitrationPhaseSegment1 == 1) and triple sampling
10	0x400	kDataPhaseSegment1IsZero	mDataPhaseSegment1 == 0
11	0x800	kDataPhaseSegment1IsGreaterThan32	mDataPhaseSegment1 > 32
12	0x1000	kDataPhaseSegment2IsLowerThan2	mDataPhaseSegment2 < 2
13	0x2000	kDataPhaseSegment2IsGreaterThan16	mDataPhaseSegment2 > 16
14	0x4000	kDataSJWIsZero	mDataSJW == 0
15	0x8000	kDataSJWIsGreaterThan16	mDataSJW > 16
16	0x1_0000	kDataSJWIsGreaterThanPhaseSegment2	mDataSJW > mDataPhaseSegment2

Table 10 – The ACANFD_FeatherM4CAN_Settings::CANBitSettingConsistency method error codes

Note. If CAN bit settings are not consistent (see section 20.2 page 38), the returned value is irrelevant.

20.4 The exactArbitrationBitRate method

```
bool ACANFD_FeatherM4CAN_Settings::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 38), the returned value is irrelevant.

20.5 The exactDataBitRate method

```
bool ACANFD_FeatherM4CAN_Settings::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 38), the returned value is irrelevant.

20.6 The ppmFromDesiredArbitrationBitRate method

```
uint32_t ACANFD_FeatherM4CAN_Settings::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 38), the returned value is irrelevant.

20.7 The `ppmFromDesiredDataBitRate` method

```
uint32_t ACANFD_FeatherM4CAN_Settings::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 38), the returned value is irrelevant.

20.8 The `arbitrationSamplePointFromBitStart` method

```
uint32_t ACANFD_FeatherM4CAN_Settings::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 38), the returned value is irrelevant.

20.9 The `dataSamplePointFromBitStart` method

```
uint32_t ACANFD_FeatherM4CAN_Settings::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 38), the returned value is irrelevant.

20.10 Properties of the `ACANFD_FeatherM4CAN_Settings` class

All properties of the `ACANFD_FeatherM4CAN_Settings` class are declared `public` and are initialized (table 11).

20.10.1 The `mModuleMode` property

This property defines the mode requested at this end of the configuration process: `NORMAL_FD` (default value), `INTERNAL_LOOP_BACK`, `EXTERNAL_LOOP_BACK`.

Property	Type	Initial value	Comment
mDesiredArbitrationBitRate	uint32_t	Constructor argument	
mDataBitRateFactor	DataBitRateFactor	Constructor argument	
mBitRatePrescaler	uint8_t	32	See section 20.1 page 33
mArbitrationPhaseSegment1	uint16_t	256	See section 20.1 page 33
mArbitrationPhaseSegment2	uint8_t	128	See section 20.1 page 33
mArbitrationSJW	uint8_t	128	See section 20.1 page 33
mDataPhaseSegment1	uint8_t	32	See section 20.1 page 33
mDataPhaseSegment2	uint8_t	16	See section 20.1 page 33
mDataSJW	uint8_t	16	See section 20.1 page 33
mTripleSampling	bool	true	See section 20.1 page 33
mBitSettingOk	bool	true	See section 20.1 page 33
mModuleMode	ModuleMode	NORMAL_FD	See section 20.10.1 page 40
mDriverReceiveFIFO0Size	uint16_t	10	See section 14.1 page 21
mHardwareRxFIFO0Size	uint8_t	64	See section 12 page 17
mHardwareRxFIFO0Payload	Payload	PAYLOAD_64_BYTES	See section 12 page 17
mDriverReceiveFIFO1Size	uint16_t	0	See section 14.1 page 21
mHardwareRxFIFO1Size	uint8_t	0	See section 12 page 17
mHardwareRxFIFO1Payload	Payload	PAYLOAD_64_BYTES	See section 12 page 17
mEnableRetransmission	bool	true	See section 20.10.2 page 41
mDiscardReceivedStandardRemoteFrames	bool	false	See section 15 page 22
mDiscardReceivedExtendedRemoteFrames	bool	false	See section 15 page 22
mNonMatchingStandardFrameReception	FilterAction	FIFO0	See section 15 page 22
mNonMatchingExtendedFrameReception	FilterAction	FIFO0	See section 15 page 22
mTransceiverDelayCompensation	uint8_t	5	See section 20.10.3 page 41
mDriverTransmitFIFOSize	uint8_t	20	See section 8 page 14
mHardwareTransmitTxFIFOSize	uint8_t	24	See section 8 page 14
mHardwareDedicacedTxBufferCount	uint8_t	8	See section 9 page 15
mHardwareTransmitBufferPayload	Payload	PAYLOAD_64_BYTES	See section 11 page 16
mNonMatchingStandardMessageCallBack	ACANFDCallBackRoutine	nullptr	See section 16.1 page 29
mNonMatchingExtendedMessageCallBack	ACANFDCallBackRoutine	nullptr	See section 16.2 page 30

Table 11 – Properties of the ACANFD_FeatherM4CAN_Settings class

20.10.2 The `mEnableRetransmission` property

By default, a trame is automatically retransmitted is an error occurs during its transmission, or if its transmission is preempted by a higher priority frame. You can turn off this feature by setting the `mEnableRetransmission` to `false`.

20.10.3 The `mTransceiverDelayCompensation` property

Setting the *Transmitter Delay Compensation* is required when data bit rate switch is enabled and data phase bit time that is shorter than the transceiver loop delay. The `mTransceiverDelayCompensation` property is by default set to 8 by the `ACANFD_FeatherM4CAN_Settings` constructor.

For more details, see `DS60001507G`, sections 39.6.2.4, pages 1095 and 1096.

21 Other ACANFD_FeatherM4CAN methods

21.1 The getStatus method

```
ACANFD_FeatherM4CAN::Status ACANFD_FeatherM4CAN::getStatus (void) const ;
```

21.1.1 The txErrorCount method

```
uint16_t ACANFD_FeatherM4CAN::Status::txErrorCount (void) const ;
```

This method returns 256 if the bus status is *Bus Off*, and the *Transmitter Error Counter* value otherwise.

21.1.2 The rxErrorCount method

```
uint8_t ACANFD_FeatherM4CAN::Status::rxErrorCount (void) const ;
```

This method returns the *Receive Error Counter* value.

21.1.3 The isBusOff method

```
bool ACANFD_FeatherM4CAN::Status::isBusOff (void) const ;
```

This method returns `true` if the bus status is *Bus Off*, and `false` otherwise.

21.1.4 The transceiverDelayCompensationOffset method

```
uint8_t ACANFD_FeatherM4CAN::Status::transceiverDelayCompensationOffset (void) const ;
```

This method returns *Transceiver Delay Compensation Offset* value.