ACANFD_STM32 Arduino library, for NUCLEO-G431KB, NUCLEO-G474RE and NUCLEO-H743ZI2 boards
Version 1.0.0

Pierre Molinaro

August 2, 2023

Contents

1 Versions

Version	Date	Comment
1.0.1	August 2, 2023	Bug Fix: correct setting of transceiver delay compensation.
1.0.0	July 19, 2023	Initial release.

2 Features

The `ACANFD_STM32` library is a CANFD (*Controller Area Network with Flexible Data*) Controller driver for the *NUCLEO-G431KB*[1], the *NUCLEO-G474RE*[2] and the *NUCLEO-H743ZI2*[3] boards running STM32duino. 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 all 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;
- standard reception filters can be easily defined;
- 128 extended reception filters can be easily defined;
- reception filters accept callback functions;
- hardware transmit buffer sizes are customisable (only `NUCLEO-H743ZI2`);
- hardware receive buffer sizes are customisable (only `NUCLEO-H743ZI2`);
- driver transmit buffer size is customisable;
- driver receive buffer size is customisable;
- the message RAM allocation is customizable and the driver checks no overflow occurs (only `NUCLEO-H743ZI2`);
- *internal loop back*, *external loop back* controller modes are selectable.

[1] https://www.st.com/en/evaluation-tools/nucleo-g431kb.html
[2] https://www.st.com/en/evaluation-tools/nucleo-g474re.html
[3] https://www.st.com/resource/en/user_manual/um2407-stm32h7-nucleo144-boards-mb1364-stmicroelectronics.pdf

For handled microcontrollers, CANFD modules are quite similar. The difference concerns the message RAM. It contains several sections for standard filters, extended filters, receive FIFOs, Tx Buffers.

For NUCLEO–G431KB and NUCLEO–G474RE, the sections sizes are statically allocated, for a total word size equal to 212 (848 bytes).

For NUCLEO–H743ZI2, the sections sizes are programmable, the two CANFD modules share a common 2560 words message RAM (10,240 bytes). The driver hides the details of the allocation, the user has just to specify the amount attributed to each CANFD module.

2.1 NUCLEO-G431KB

The NUCLEO–G431KB contains one CANFD module `canfd1` (table 2).

Name	**fdcan1**
Default FDCAN Clock	170 MHz
Default TxPin	PA_12
Alternate TxPin	PB_9
Default RxPin	PA_11
Alternate RxPin	PB_8
Message RAM Size	212 words
Standard Receive filters	28 elements (28 words)
Extended Receive filters	8 elements (16 words)
Rx FIFO0	3 elements (54 words)
Rx FIFO1	3 elements (54 words)
Tx Buffers	3 elements (54 words)

Table 2 – The CANFD module of NUCLEO–G431KB

2.2 NUCLEO-G474RE

The NUCLEO–G474RE contains three CANFD modules `canfd1`, `canfd2` and `canfd3` (table 3).

2.3 NUCLEO-H743ZI2

The NUCLEO–H743ZI2 contains two CANFD modules `canfd1` and `canfd2` (table 4).

3 Data flow

3.1 NUCLEO–G431KB, NUCLEO–G474RE

The data flow in given in figure 1.

Name	fdcan1	fdcan2	fdcan3
Default FDCAN Clock	*168 MHz, common to the three CANFD modules*		
Default TxPin	PA_12	PB_6	PA_15
Alternate TxPin	PB_9	PB_13	PB_4
Default RxPin	PA_11	PB_5	PA_8
Alternate RxPin	PB_8	PB_12	PB_3
Message RAM Size	212 words	212 words	212 words
Standard Receive filters	28 elements (28 words)	28 elements (28 words)	28 elements (28 words)
Extended Receive filters	8 elements (16 words)	8 elements (16 words)	0-8 elements (16 words)
Rx FIFO0	3 elements (54 words)	3 elements (54 words)	3 elements (54 words)
Rx FIFO1	3 elements (54 words)	3 elements (54 words)	3 elements (54 words)
Tx Buffers	3 elements (54 words)	3 elements (54 words)	3 elements (54 words)

Table 3 – The three CANFD modules of NUCLEO-G474RE

Name	fdcan1	fdcan2
Default FDCAN Clock	*120 MHz, common to the two CANFD modules*	
Default TxPin	PD_1	PB_6
Alternate TxPin	PB_9, PA_12	PB_13
Default RxPin	PD_0	PB_5
Alternate RxPin	PB_8, PA_11	PB_12
Message RAM Size	*2560 words, shared between the two CANFD modules*	
Standard Receive filters	0-128 elements (0-128 words)	0-128 elements (0-128 words)
Extended Receive filters	0-64 elements (0-128 words)	0-64 elements (0-128 words)
Rx FIFO0	0-64 elements (0-1152 words)	0-64 elements (0-1152 words)
Rx FIFO1	0-64 elements (0-1152 words)	0-64 elements (0-1152 words)
Tx Buffers	0-32 elements (0-576 words)	0-32 elements (0-576 words)

Table 4 – The two CANFD modules of NUCLEO-H743ZI2

Sending messages. The `ACANFD_STM32` driver defines a *driver transmit FIFO* (default size: 20 messages). The module *hardware transmit FIFO* has a fixed a size of 3 messages.

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 14 page 27 for details, and the `idx` property of the sent message should be equal to 0 (default value). If the `idx` property is greater than 0, the message is lost.

You can call the `sendBufferNotFullForIndex` method (section 14.1 page 27) for testing if a send buffer is not full.

Receiving messages. The *CANFD Protocol Engine* transmits all correct frames to the *reception filters*. By default, they are configured as pass-all to `FIFO0`, see section 16 page 31 for configuring them. Messages that pass the filters are stored in the *Hardware Reception FIFO0* (fixed size: 3) or in the *Hardware Reception FIFO1* (fixed size: 3). 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, the size of the *Driver Receive FIFO 1* is 0 by default – see section 15.1 page 30 for changing the default value. Two user methods are available:

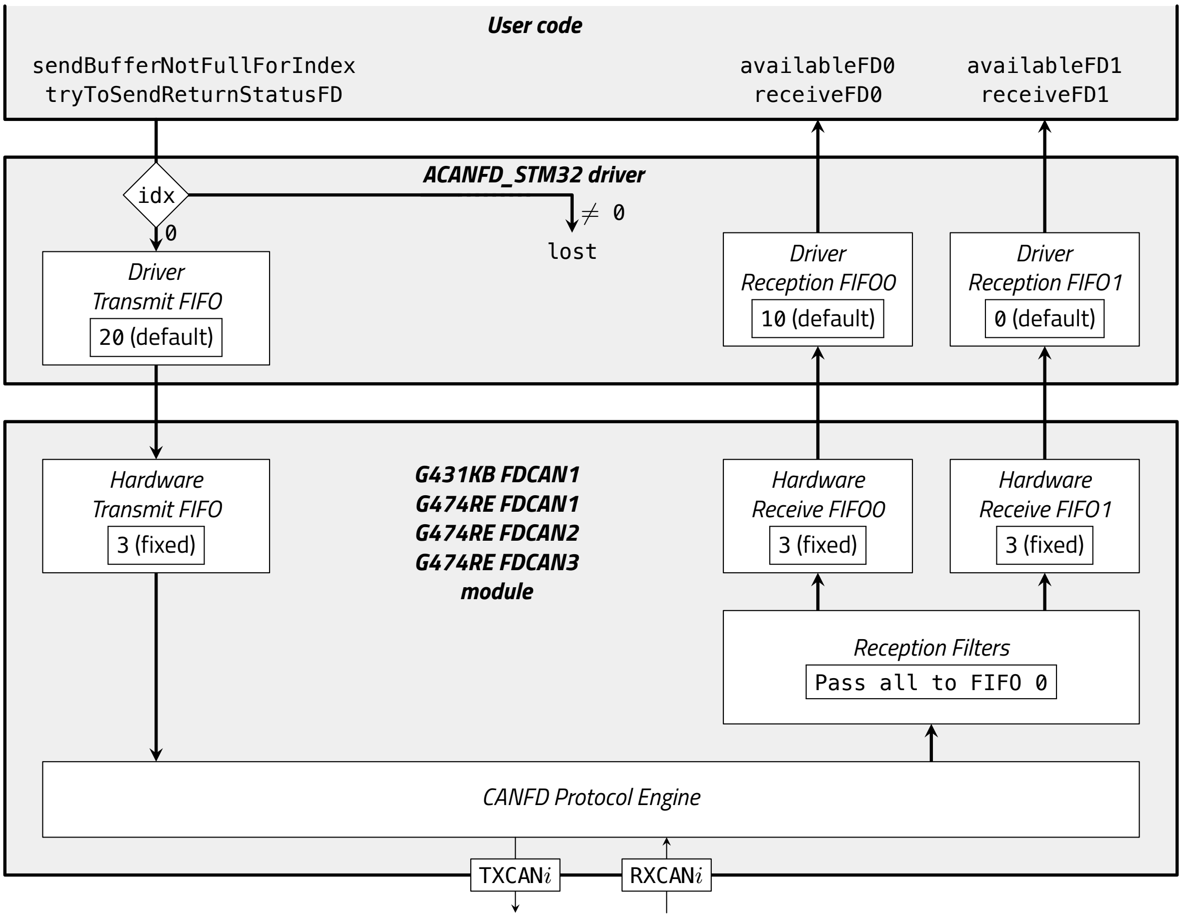

Figure 1 – NUCLEO-G431KB, NUCLEO-G474RE: message flow in ACANFD_STM32 driver and FDCANi module

- 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 15 page 29;
- 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 15 page 29.

3.2 NUCLEO-H743ZI2

The data flow in given in figure 2.

Sending messages. The ACANFD_STM32 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 14 page 27 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*;

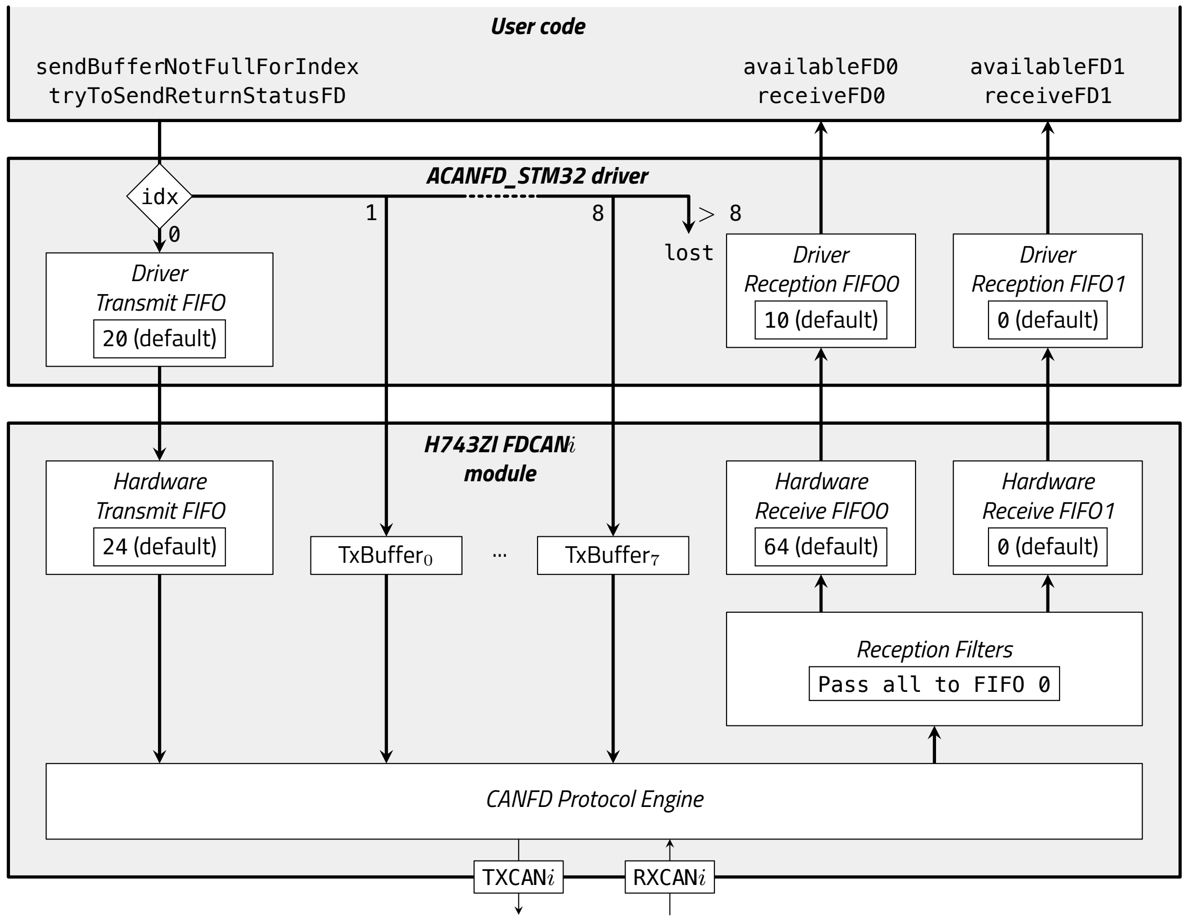

Figure 2 – NUCLEO-H743ZI2: message flow in ACANFD_STM32 driver and FDCANi module

- 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 14.1 page 27) 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 16 page 31 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 FIFOi*. The size of the *Driver Receive FIFO 0* is 10 by default – see section 15.1 page 30 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 15 page 29;
- 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 15 page 29.

4 A sample sketch: $board$–LoopBackDemo

The `G431KB-LoopBackDemo`, `G474RE-LoopBackDemo` and `H743ZI2-LoopBackDemo` are sample codes for introducing the `ACANFD_STM32` library. They demonstrate how to configure the library, to send a CANFD message, and to receive a CANFD message.

Note. Theses codes run without any additional CAN hardware, as the `FDCAN`i modules are configured in `EXTERNAL_LOOP_BACK` mode (see section 21.10.1 page 49); the `FDCAN`i module receives every CANFD frame it sends, and emitted frames can be observed on its `TxPin`.

4.1 Including <ACANFD_STM32.h>

You should include the `ACANFD_STM32.h` header only once in your sketch. If some other C++ files require access to `fdcan`i, include `ACANFD_STM32_from_cpp.h` header.

If you include `<ACANFD_STM32.h>` from several files, the `fdcan`i variables are multiply-defined, therefore you get a link error.

The `NUCLEO-H743ZI2` is a special case. As the message RAM is programmable, you should define the size allocated to each FDCAN module (the total should not exceed 2,560):

- the `FDCAN1_MESSAGE_RAM_WORD_SIZE` constant define the word size allocated to `fdcan1`;
- the `FDCAN2_MESSAGE_RAM_WORD_SIZE` constant define the word size allocated to `fdcan2`.

For example:

```
static const uint32_t FDCAN1_MESSAGE_RAM_WORD_SIZE = 1000 ;
static const uint32_t FDCAN2_MESSAGE_RAM_WORD_SIZE = 1000 ;

#include <ACANFD_STM32.h>
```

If you do not use a module, it is safe to allocate a zero size (see `H743ZI2-LoopBackDemoIntensive-CAN1` demo sketch for example).

4.2 The setup function

```
void setup () {
//--- Switch on builtin led
  pinMode (LED_BUILTIN, OUTPUT) ;
  digitalWrite (LED_BUILTIN, HIGH) ;
//--- Start serial
  Serial.begin (9600) ;
//--- 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_STM32_Settings settings (500 * 1000, DataBitRateFactor::x4) ;
...
```

Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACANFD_STM32_Settings` class. The constructor has two parameters: the desired CAN arbitration bit rate (here, 500 kbit/s), and the data bit rate, given by a multiplicative factor of the arbitration bit rate; here, the data bit rate is 500 kbit/s * 4 = 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_STM32_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 21.10 page 48 lists all properties you can override.

```
...
const uint32_t errorCode = fdcan1.beginFD (settings) ;
...
```

This is the third step, configuration of the `FDCAN1` driver with `settings` values. The driver is configured for being able to send any (base / extended, data / remote, CAN / CANFD) frame, and to receive all (base / extended, data / remote, CAN / CANFD) frames. If you want to define reception filters, see section 16 page 31.

```
...
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 20.2 page 41.

As the `beginFD` does not modify the `settings`, you can use the same object for the other modules (if any):

```
...
const uint32_t errorCode2 = fdcan2.beginFD (settings) ;
if (errorCode2 != 0) {
  Serial.print ("Configuration error 0x") ;
```

```
    Serial.println (errorCode2, HEX) ;
  }
  ...
  const uint32_t errorCode3 = fdcan3.beginFD (settings) ;
  if (errorCode3 != 0) {
    Serial.print ("Configuration␣error␣0x") ;
    Serial.println (errorCode3, HEX) ;
  }
  ...
```

4.3 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.

4.4 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 = fdcan1.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 && fdcan1.receiveFD0 (frame)) {
    bool sameFrames = ... compare frame and gSentFrame ... ;
    if (sameFrames) {
      gReceiveCount += 1 ;
      Serial.print ("Received␣") ;
      Serial.println (gReceiveCount) ;
    }else{
      gOk = false ;
      ... Print error ...
    }
  }
}
```

5 The CANMessage class

Note. The CANMessage class is declared in the CANMessage.h header file. The class declaration is protected by an include guard that causes the macro GENERIC_CAN_MESSAGE_DEFINED to be defined. The ACAN2515 driver[4], the ACAN2517 driver[5] and the ACAN2517FD driver[6] contain an identical CANMessage.h header file, enabling using the ACANFD_STM32 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 6.3 page 15).

```
class CANMessage {
  public : uint32_t id = 0 ;  // Frame identifier
  public : bool ext = false ; // false -> standard frame, true -> extended frame
  public : bool rtr = false ; // false -> data frame, true -> remote frame
  public : uint8_t idx = 0 ;  // 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} ;
```

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

```
  } ;
} ;
```

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 (STM32 processors are little-endian).

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

- for a received message, it contains the acceptance filter index (see section 17 page 37) or 255 if it does not correspond to any filter;
- on sending messages, it is used for selecting the transmit buffer (see section 14 page 27).

6 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[7] contains an identical CANFDMessage.h header file, enabling using the ACANFD_STM32 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
```

6.1 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)
```

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

```
  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 (STM32 processors are little-endian).

6.2 The default constructor

All properties are initialized by default, and represent a base data frame, with an identifier equal to 0, and without any data (table 5).

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 5 – `CANFDMessage` default constructor initialization

6.3 Constructor from `CANMessage`

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

All properties are initialized from the `inCANMessage` (table 6). Note that only `data64[0]` is initialized from `inCANMessage.data64`.

6.4 The `type` property

The `type` property value is an instance of an enumerated type:

```
class CANFDMessage {
```

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 6 – CANFDMessage constructor CANMessage

```
...
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 7.

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 7 – CANFDMessage type property

6.5 The len property

Note that len property contains the actual length, not its encoding in CANFD frames. So valid values are: 0, 1, ..., 8, 12, 16, 20, 24, 32, 48, 64. Having other values is an error that prevents frame to be sent by the ACANFD_STM32::tryToSendReturnStatusFD method. You can use the pad method (see section 6.7 page 16) for padding with 0x00 bytes to the next valid length.

6.6 The idx property

The idx property is not used in CANFD frames, but it is used for selecting the transmit buffer (see section 14 page 27).

6.7 The pad method

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

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

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

6.8 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 7 page 16, and false otherwise.

7 Modifying FDCAN Clock

7.1 Why define custom system clock configuration?

In short: because default FDCAN clock can make a given bit rate unavailable.

For example, I want with a NUCLEO-G474RE the 5 Mbit/s a data bit rate (arbitration bit rate is not significant for getting correct bit rate: if data bit rate is correct, so is the arbitration bit rate). Default FDCAN clock is 168 MHz (see table 3 page 7).

From the G474RE-LoopBackDemo.ino sketch, the settings object is instanciated by (we choose arbitration bit rate equal to 500 kbit/s):

```
  ACANFD_STM32_Settings settings (500 * 1000, DataBitRateFactor::x10) ;
```

Running the sketch prints the data and arbitration bits decomposition:

```
...
Bit Rate prescaler: 1
Arbitration Phase segment 1: 254
Arbitration Phase segment 2: 85
Arbitration SJW: 85
Actual Arbitration Bit Rate: 494117 bit/s
Arbitration sample point: 75%
Exact Arbitration Bit Rate ? no
Data Phase segment 1: 24
```

```
Data Phase segment 2: 9
Data SJW: 9
Actual Data Bit Rate: 4941176 bit/s
...
```

The closest data bit rate is 4.941 MHz, the closest arbitration bit rate is 494.117 kbit/s, the difference is 1.2% from expected bit rate.

Getting exactly 5 Mbit/s data bit rate is not possible because 5 is not a divisor of 168.

The only solution is to change the FDCAN clock.

We need a FDCAN clock frequency multiple of 5 MHz, minimum 10 times 5 MHz, for allowing correct bit timing. Note setting a custom FDCAN clock frequency affect also CPU speed. Note also STM32G474RE max CPU frequency is 170 MHz. Some valid frequencies are 160 MHz, 165 MHz or 170 MHz.

7.2 Define custom system clock configuration

For an example, see the `G474RE-LoopBackDemo-customSystemClock.ino` demo sketch.

For any board, System Clock can be overriden. The mechanism is described in:

https://github.com/stm32duino/Arduino_Core_STM32/wiki/Custom-definitions#systemclock_config

You have to define a custom `SystemClock_Config` function, with values adapted to the FDCAN clock you want.

7.2.1 Find the `SystemClock_Config` function for your board

For the `NUCLEO_G474RE`, the `SystemClock_Config` function file is defined in STM32duino package. On my Mac, it is in the file:

`~/Library/Arduino15/packages/STMicroelectronics/hardware/stm32/2.6.0/variants/`
`STM32G4xx/G473R(B-C-E)T_G474R(B-C-E)T_G483RET_G484RET/variant_NUCLEO_G474RE.cpp`

The found `SystemClock_Config` function is:

```
WEAK void SystemClock_Config(void) {
  RCC_OscInitTypeDef RCC_OscInitStruct = {};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {};
#ifdef USBCON
  RCC_PeriphCLKInitTypeDef PeriphClkInit = {};
#endif

  /* Configure the main internal regulator output voltage */
  HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1_BOOST);
  /* Initializes the CPU, AHB and APB busses clocks */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI48 | RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.HSI48State = RCC_HSI48_ON;
```

```
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = RCC_PLLM_DIV2;
  RCC_OscInitStruct.PLL.PLLN = 28;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
    Error_Handler();
  }
  /* Initializes the CPU, AHB and APB busses clocks */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK
                              | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_8) != HAL_OK) {
    Error_Handler();
  }

#ifdef USBCON
  /* Initializes the peripherals clocks */
  PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USB;
  PeriphClkInit.UsbClockSelection = RCC_USBCLKSOURCE_HSI48;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK) {
    Error_Handler();
  }
#endif
}
```

Note the function is declared `WEAK`, allowing it to be overridden. The original function is duplicated in the sketch, and will be modified.

7.2.2 Understand the original settings

We have to understand how the original settings provide a FDCLAN clock of 168 MHz. A very very simplified explaination of the clock tree is (for a full understanding of the clock tree, consider using `STM32CubeMX`):

- FDCAN clock is PCLK1;
- `PCLK1 = HSE_CLOCK * PLLN / PLLM / PLLP`.

For the `NUCLEO-G474RE`, `HSE_CLOCK` = 24 MHz, we cannot change that, it is given by STLink.

In the original file (see above):

- RCC_OscInitStruct.PLL.PLLM = RCC_PLLM_DIV2 → PLLM=2;
- RCC_OscInitStruct.PLL.PLLN = 28 → PLLN=28;
- RCC_OscInitStruct.PLL.PLLP = RCC_PLLM_DIV2 → PLLP=2.

So we can check that: PCLK1 = 24 MHz * 28 / 2 / 2 = 168 MHz.

Note STM32Duino provides (use STM32CubeMX for understanding the role of each clock):

- the F_CPU constant equal to CPU speed (here, 168 MHz);
- the HAL_RCC_GetPCLK1Freq() function that returns the PCLK1 frequency (here, 168 MHz);
- the HAL_RCC_GetPCLK2Freq() function that returns the PCLK2 frequency (here, 168 MHz);
- the HAL_RCC_GetHCLKFreq() function that returns the HCLK frequency (here, 168 MHz);
- the HAL_RCC_GetSysClockFreq() function that returns the SysClock frequency (here, 168 MHz).

The ACANFD_STM32 library provides the fdcanClock() function, that returns the frequency used for bit timing computations (here, 168 MHz).

7.2.3 Adapt the SystemClock_Config function settings

For setting a system clock, I suggest to first try to adapt PLLM and PLLN values. **Caution:** any value is not valid, I strongly suggest using STM32CubeMX for checking a given setting. Some valid settings:

PCLK1=160 MHz. Choose PLLM=3 and PLLN=40: PCLK1 = 24 MHz * 40 / 3 / 2 = 160 MHz

PCLK1=165 MHz. Choose PLLM=8 and PLLN=110: PCLK1 = 24 MHz * 110 / 8 / 2 = 165 MHz

PCLK1=170 MHz. Choose PLLM=6 and PLLN=85: PCLK1 = 24 MHz * 85 / 6 / 2 = 170 MHz

Note: for setting PLLM, use the RCC_PLLM_DIVi symbols.

Always validate your setting by checking actual CAN clock (call fdcanClock function), and examine actual Data Bit Rate (call settings.actualDataBitRate function, see below). For PCLK1=160 MHz, the SystemClock_Config function is:

```
extern "C" void SystemClock_Config (void) { // extern "C" IS REQUIRED!
  RCC_OscInitTypeDef RCC_OscInitStruct = {};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {};
#ifdef USBCON
  RCC_PeriphCLKInitTypeDef PeriphClkInit = {};
#endif

  /* Configure the main internal regulator output voltage */
  HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1_BOOST);
  /* Initializes the CPU, AHB and APB busses clocks */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI48 | RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
```

```
  RCC_OscInitStruct.HSI48State = RCC_HSI48_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = RCC_PLLM_DIV8 ; // Original value: RCC_PLLM_DIV2
  RCC_OscInitStruct.PLL.PLLN = 110 ; // Original value: 28
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
    Error_Handler();
  }
  /* Initializes the CPU, AHB and APB busses clocks */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK
                              | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_8) != HAL_OK) {
    Error_Handler();
  }

#ifdef USBCON
  /* Initializes the peripherals clocks */
  PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USB;
  PeriphClkInit.UsbClockSelection = RCC_USBCLKSOURCE_HSI48;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK) {
    Error_Handler();
  }
#endif
}
```

Now, the sketch can be run in order to print the serial monitor output:

```
...
Bit Rate prescaler: 1
Arbitration Phase segment 1: 246
Arbitration Phase segment 2: 83
Arbitration SJW: 83
Actual Arbitration Bit Rate: 500000 bit/s
Arbitration sample point: 74%
Exact Arbitration Bit Rate ? yes
Data Phase segment 1: 23
Data Phase segment 2: 9
Data SJW: 9
Actual Data Bit Rate: 5000000 bit/s
...
```

```
CPU frequency: 165000000 Hz
PCLK1 frequency: 165000000 Hz
PCLK2 frequency: 165000000 Hz
HCLK frequency: 165000000 Hz
SysClock frequency: 165000000 Hz
CAN Clock: 165000000 Hz
...
```

The can clock is 165 MHz, the data bit rate is exactly 5 Mbit/s, and the arbitration bit rate exactly 500 kbit/s.

8 Transmit FIFO

The transmit FIFO (see figure 1 page 8 and figure 2 page 9) is composed by:

- the *driver transmit FIFO*, whose size is positive or zero; you can change the default size by setting the `mDriverTransmitFIFOSize` property of your `settings` object;
- the *hardware transmit FIFO*, whose size is:
  - for `NUCLEO-G431KB`, `NUCLEO-G474RE`: 3, you cannot change this size;
  - for `NUCLEO-H743ZI2`: 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 ($\texttt{TxBuffer}_i$)

Transmit buffers are only available for `NUCLEO-H743ZI2`. There are `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 14.1 page 27) 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 Transmit Priority

Pending dedicaced $\texttt{TxBuffer}_i$ and oldest pending Tx FIFO buffer are scanned, and buffer with lowest message identifier gets highest priority and is transmitted next.

11 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 2 page 9 and figure 1 page 8) is composed by:

- the *hardware receive FIFO$_i$* (in the Message RAM, see section 13 page 25), whose size is:
  - for NUCLEO-G431KB, NUCLEO-G474RE: 3, you cannot change this size;
  - for NUCLEO-H743ZI2: 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.

12 Payload size

This section is only relevant for NUCLEO-H743ZI2. NUCLEO-H431KB and NUCLEO-H474RE payload size is always 72 bytes.

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 13 page 25. 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_STM32_Settings class, as described in table 8. The type of theses properties is the ACANFD_STM32_Settings::Payload enumeration type, and defines 8 values (table 9).

Object Size specification	Default value	Applies to
mHardwareTransmitBufferPayload	PAYLOAD_64_BYTES	Hardware transmit FIFO, TxBuffers
mHardwareRxFIFO0Payload	PAYLOAD_64_BYTES	Hardware receive FIFO 0
mHardwareRxFIFO1Payload	PAYLOAD_64_BYTES	Hardware receive FIFO 1

Table 8 – Payload properties of ACANFD_STM32_Settings class

Object Size specification	Handles frames up to	Object Size
ACANFD_STM32_Settings::PAYLOAD_8_BYTES	8 bytes	4 words = 16 bytes
ACANFD_STM32_Settings::PAYLOAD_12_BYTES	12 bytes	5 words = 20 bytes
ACANFD_STM32_Settings::PAYLOAD_16_BYTES	16 bytes	6 words = 24 bytes
ACANFD_STM32_Settings::PAYLOAD_20_BYTES	20 bytes	7 words = 28 bytes
ACANFD_STM32_Settings::PAYLOAD_24_BYTES	24 bytes	8 words = 32 bytes
ACANFD_STM32_Settings::PAYLOAD_32_BYTES	32 bytes	10 words = 40 bytes
ACANFD_STM32_Settings::PAYLOAD_48_BYTES	48 bytes	14 words = 56 bytes
ACANFD_STM32_Settings::PAYLOAD_64_BYTES	64 bytes	18 words = 72 bytes

Table 9 – ACANFD_STM32_Settings object size from payload size specification

12.1 The ACANFD_STM32_Settings::wordCountForPayload static method

```
uint32_t ACANFD_STM32_Settings::wordCountForPayload (const Payload inPayload);
```

This static method returns the object word size for a given payload specification, following table 9.

12.2 The ACANFD_STM32_Settings::frameDataByteCountForPayload static method

```
uint32_t ACANFD_STM32_Settings::frameDataByteCountForPayload (const Payload inPayload);
```

This static method returns the handled data byte count for a given payload specification, following table 9.

12.3 Changing the default payloads

See LoopBackDemoCANFDIntensive_CAN1_payload sample sketch.

Overriding the default payloads enables saving Message RAM size.

mHardwareTransmitBufferPayload. Setting the mHardwareTransmitBufferPayload property limits the size of TxBuffers. Data bytes beyond this limit are not stored in the TxBuffers. The transmitted frame does not contain this data bytes, but 0xCC bytes instead. For example, if it is set to ACANFD_STM32_Settings::PAYLOAD_24_BYTES, and a 32-byte data frame is submitted:

- for indexes from 0 to 23, the transmitted data are those of the message;
- for indexes from 24 to 31, 0xCC data bytes are sent.

If you submit a frame with 24 bytes of data or less, all message bytes are sent.

mHardwareRxFIFO0Payload. Setting the mHardwareTransmitBufferPayload property limits the size of hardware FIFO 0 elements. Received frame data bytes beyond this limit are not stored in the hardware FIFO 0. The retrived frame does not contain this data bytes, but 0xCC bytes instead. For example, if it is set to ACANFD_STM32_Settings::PAYLOAD_24_BYTES, and a 32-byte data frame is received:

- for indexes from 0 to 23, the message contains the received frame corresponding data bytes;
- for indexes from 24 to 31, the message contains 0xCC data bytes.

If a frame with 24 bytes of data or less is received, all message bytes are received.

mHardwareRxFIFO1Payload. Same for hardware FIFO 1 elements.

13 Message RAM

This section is only relevant for NUCLEO-H743ZI2. NUCLEO-H431KB and NUCLEO-H474RE Message RAM sections are fixed and not programmable.

Each CANFD module uses *Message RAM* for storing TxBuffers, hardware transmit FIFO, hardware receives FIFO, and reception filters.

The `NUCLEO-H743ZI2` two FDCAN modules share 2,560 words space.

A message RAM contains[8]:

- 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 2,560 words.

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);

There are five properties of `ACANFD_STM32_Settings` class that affect the actual message RAM size:

- the `mHardwareRxFIFO0Size` property sets the hardware receive FIFO 0 element count (0-64);
- the `mHardwareRxFIFO0Payload` property sets the size of the hardware receive FIFO 0 element (table 9);
- 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 9);
- 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 9).

[8] See DS60001507G, section 39.9.1 page 1177.

The ACANFD_STM32::messageRamRequiredSize method returns the required word size.

The ACANFD_STM32::begin method checks the message RAM allocated size is greater or equal to the required size. Otherwise, it raises the error code kMessageRamTooSmall.

14 Sending frames: the tryToSendReturnStatusFD method

The ACANFD_STM32::tryToSendReturnStatusFD method sends CAN 2.0B and CANFD frames:

```
uint32_t ACANFD_STM32::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 6.8 page 17);
- 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 22) ;
- 1 … settings.mHardwareDedicacedTxBufferCount for a dedicaced TxBuffer (section 9 page 23).

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_STM32_Settings::DATA_BITRATE_x$_n$ setting;
- CANFD_WITH_BIT_RATE_SWITCH, with the ACANFD_STM32_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.

14.1 Testing a send buffer: the sendBufferNotFullForIndex method

```
bool ACANFD_STM32::sendBufferNotFullForIndex (const uint32_t inTxBufferIndex);
```

This method returns true if the corresponding transmit buffer is not full, and false otherwise (table 10).

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 10 – Value returned by the `sendBufferNotFullForIndex` method

14.2 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 ;
    }
  }
  ...
}
```

15 Retrieving received messages using the receiveFDi method

```
bool ACANFD_STM32::receiveFD0 (CANFDMessage & outMessage) ;
bool ACANFD_STM32::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.

15.1 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_STM32_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.

15.2 The driverReceiveFIFOiSize method

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

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

15.3 The driverReceiveFIFOiCount method

The driverReceiveFIFOiCount method returns the current number of messages in the driver receive FIFO i.

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

15.4 The driverReceiveFIFOiPeakCount method

The driverReceiveFIFOiPeakCount 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.driverReceiveFIFOiPeakCount ()` return `can0.driverReceiveFIFOiSize ()+1`.

15.5 The `resetDriverReceiveFIFOiPeakCount` method

The `resetDriverReceiveFIFOiPeakCount` method assign the current count to the peak value.

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

16 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.

16.1 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 16.1.1 page 31), up to 128, none by default.

The standard frame filtering is illustrated by figure 3.

16.1.1 Defining standard frame filters

```
  ACANFD_STM32_Settings settings (..., ...) ;
  ...
  ACANFD_STM32_StandardFilters standardFilters ;
  standardFilters.addSingle (0x55, ACANFD_STM32_FilterAction::FIFO0) ;
  ...
//--- Reject standard frames that do not match any filter
  settings.mNonMatchingStandardFrameReception = ACANFD_STM32_FilterAction::REJECT;
  ...
```

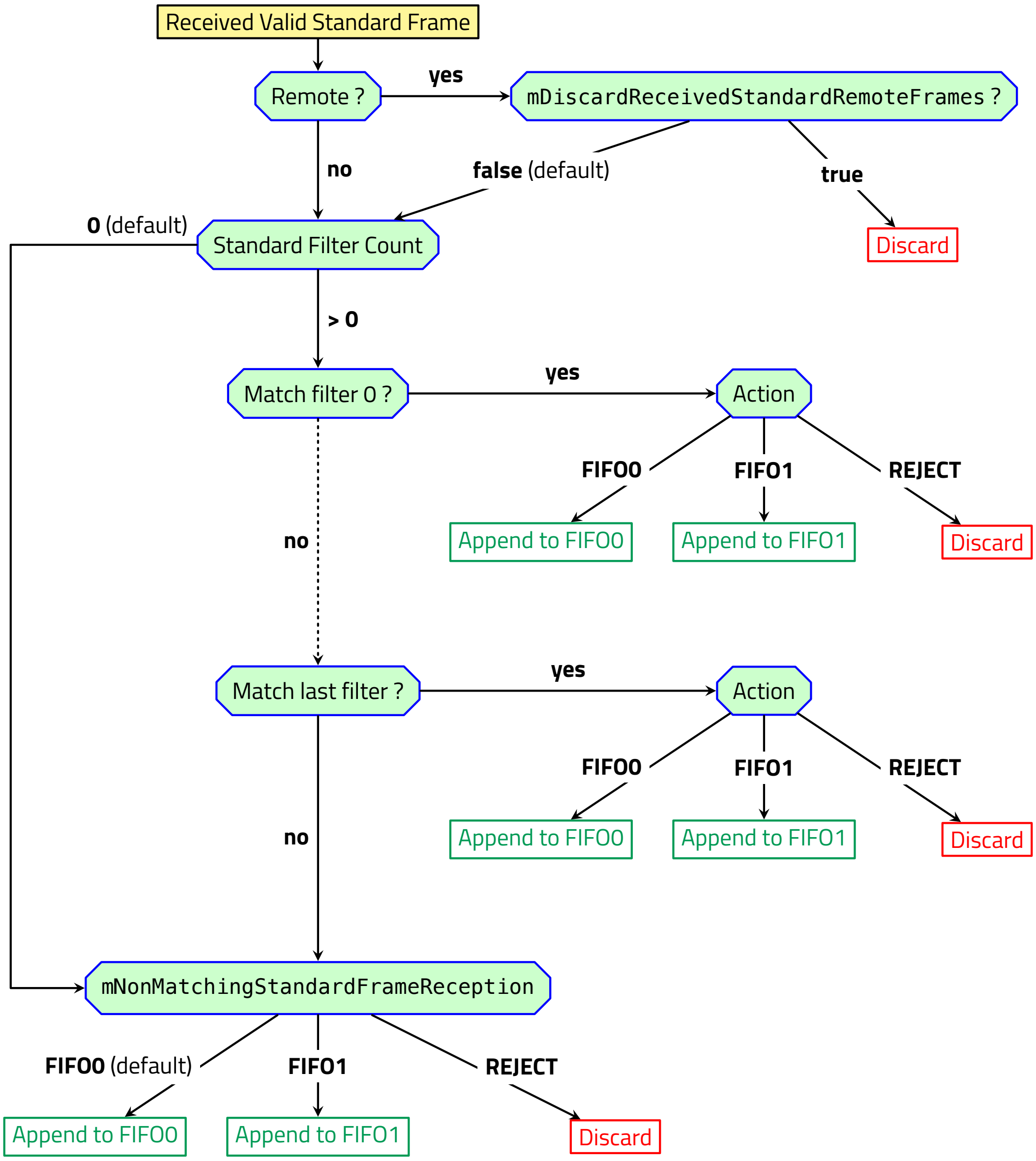

Figure 3 – Standard frame filtering

```
const uint32_t errorCode = fdcan1.beginFD (settings, standardFilters) ;
...
```

The `ACANFD_STM32_StandardFilters` class handles a standard frame filter list. Default constructor constructs an empty list. For appending filters, use the `addSingle` (section 16.1.2 page 33), `addDual` (section 16.1.3 page 33), `addRange` (section 16.1.4 page 33) or `addClassic` (section 16.1.5 page 33) 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 3 for detail).

16.1.2 Add single filter

```
bool ACANFD_STM32_StandardFilters::addSingle (const uint16_t inIdentifier,
                                   const ACANFD_STM32_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 17 page 37.

16.1.3 Add dual filter

```
bool ACANFD_STM32_StandardFilters::addDual (const uint16_t inIdentifier1,
                                 const uint16_t inIdentifier2,
                                 const ACANFD_STM32_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 17 page 37.

16.1.4 Add range filter

```
bool ACANFD_STM32_StandardFilters::addRange (const uint16_t inIdentifier1,
                                  const uint16_t inIdentifier2,
                                  const ACANFD_STM32_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 17 page 37.

16.1.5 Add classic filter

```
bool ACANFD_STM32_StandardFilters::addClassic (const uint16_t inIdentifier,
                                    const uint16_t inMask,
                                    const ACANFD_STM32_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 17 page 37.

For example:

```
standardFilters.addClassic (0x405, 0x7D5, ACANFD_STM32_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`.

16.2 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 16.2.1 page 35), up to 128, none by default.

The extended frame filtering is illustrated by figure 4.

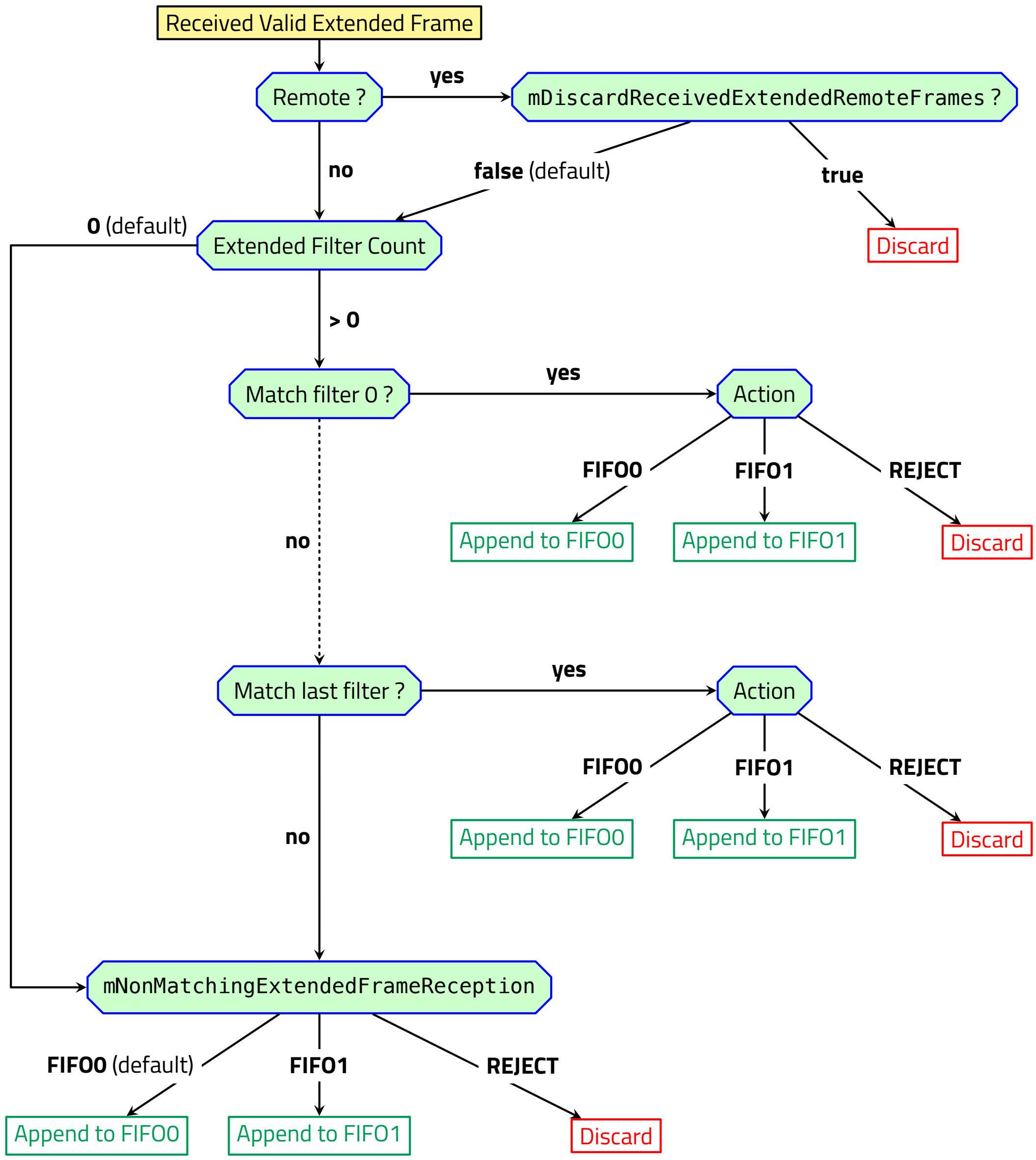

Figure 4 – Extended frame filtering

16.2.1 Defining extended frame filters

```
ACANFD_STM32_Settings settings (..., ...) ;
...
ACANFD_STM32_ExtendedFilters extendedFilters ;
extendedFilters.addSingle (0x55, ACANFD_STM32_FilterAction::FIFO0) ;
...
//--- Reject extended frames that do not match any filter
settings.mNonMatchingExtendedFrameReception = ACANFD_STM32_FilterAction::REJECT;
...
const uint32_t errorCode = fdcan1.beginFD (settings, extendedFilters) ;
...
```

The ACANFD_STM32_ExtendedFilters class handles an extended frame filter list. Default constructor constructs an empty list. For appending filters, use the addSingle (section 16.2.2 page 36), addDual (section 16.2.3 page 36), addRange (section 16.2.4 page 36) or addClassic (section 16.2.5 page 37) methods. Then, add the ACANFD_STM32_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 4 for detail).

16.2.2 Add single filter

```
bool ACANFD_STM32_ExtendedFilters::addSingle (const uint32_t inIdentifier,
                                              const ACANFD_STM32_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 17 page 37.

16.2.3 Add dual filter

```
bool ACANFD_STM32_ExtendedFilters::addDual (const uint32_t inIdentifier1,
                                            const uint32_t inIdentifier2,
                                            const ACANFD_STM32_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 17 page 37.

16.2.4 Add range filter

```
bool ACANFD_STM32_ExtendedFilters::addRange (const uint32_t inIdentifier1,
                                             const uint32_t inIdentifier2,
                                             const ACANFD_STM32_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 17 page 37.

16.2.5 Add classic filter

```
bool ACANFD_STM32_ExtendedFilters::addClassic (const uint32_t inIdentifier,
                                               const uint32_t inMask,
                                               const ACANFD_STM32_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 17 page 37.

For example:

```
extendedFilters.addClassic (0x6789, 0x1FFF67BD, ACANFD_STM32_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.

17 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 () {
  fdcan1.dispatchReceivedMessage () ; // Do not call fdcan1.receiveFD0, fdcan1.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 ()) {
  }
  ...
}
```

17.1 Dispatching non matching standard frames

Following the figure 3 page 32, 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_STM32_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.

17.2 Dispatching non matching extended frames

Following the figure 4 page 35, 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_STM32_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.

18 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 () {
  fdcan1.dispatchReceivedMessageFIFO0 () ; // Do not call fdcan1.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 ...
```

```
  }
  ...
}
```

19 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 () {
  fdcan1.dispatchReceivedMessageFIFO1 () ; // Do not call fdcan1.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 ...
  }
  ...
}
```

20 The ACANFD_STM32::beginFD method reference

20.1 The prototypes

```
uint32_t ACANFD_STM32::beginFD (const ACANFD_STM32_Settings & inSettings,
   const ACANFD_STM32_StandardFilters & inStandardFilters = ACANFD_STM32_StandardFilters (),
   const ACANFD_STM32_ExtendedFilters & inExtendedFilters = ACANFD_STM32_ExtendedFilters ()) ;

uint32_t ACANFD_STM32::beginFD (const ACANFD_STM32_Settings & inSettings,
                                const ACANFD_STM32_ExtendedFilters & inExtendedFilters) ;
```

The first argument is a ACANFD_STM32_Settings instance that defines the settings.

The second one is optional, and specifies the standard filter list (see section 16.1 page 31). By default, the standard filter list is empty.

The third one is optional, and specifies the extended filter list (see section 16.2 page 34). By default, the extended filter list is empty.

20.2 The error codes

The ACANFD_STM32::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 11. An error code could report several errors. The ACANFD_STM32 class defines static constants for naming errors. Bits 0 to 16 denote a bit configuration error, see table 12 page 47.

Bit	Code	Static constant Name	Comment
0	0x1	kBitRatePrescalerIsZero	See table 12 page 47
...	...	....	See table 12 page 47
16	0x1_0000	kDataSJWIsGreaterThanPhaseSegment2	See table 12 page 47
20	0x10_0000	kMessageRamTooSmall	See section 13 page 25
21	0x20_0000	kMessageRamNotInFirst64kio	See section 13 page 25
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 20.2.1 page 41
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 16.1 page 31
29	0x2000_0000	kExtendedFilterCountGreaterThan128	More than 128 extended filters, see section 16.2 page 34

Table 11 – The ACANFD_STM32::beginFD method error code bits

20.2.1 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.

21 ACANFD_STM32_Settings class reference

21.1 The ACANFD_STM32_Settings constructors: computation of the CAN bit settings

21.1.1 5 arguments constructor

```
ACANFD_STM32_Settings::
ACANFD_STM32_Settings (const uint32_t inDesiredArbitrationBitRate,
                       const uint32_t inDesiredArbitrationSamplePoint,
                       const DataBitRateFactor inDataBitRateFactor,
                       const uint32_t inDesiredDataSamplePoint,
                       const uint32_t inTolerancePPM = 1000) ;
```

The constructor of the `ACANFD_STM32_Settings` four mandatory arguments:

1. the desired arbitration bit rate,
2. the desired arbitration sample point (in per-cent),
3. the data bit rate factor,
4. the desired data sample point (in per-cent).

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`. The sample points are expressed in per-cent values, 60 to 80 are typical values. Note that the desired values of the sample points may not be achieved exactly, due to integer quantization. Very often the actual value is lower than the desired value. You can change the property values for be closer to the required values, see the listing in the figure 5 page 45.

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_STM32_Settings settings (1000 * 1000, 75, DataBitRateFactor::x8, 75) ;
  // 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.

21.1.2 3-arguments constructor

This constructor implicitly sets desired arbitration sample point and desired data sample point to 75.

```
ACANFD_STM32_Settings::
ACANFD_STM32_Settings (const uint32_t inDesiredArbitrationBitRate,
```

```
                         const DataBitRateFactor inDataBitRateFactor,
                         const uint32_t inTolerancePPM = 1000) ;
```

21.1.3 Exact bit rates

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_STM32_Settings` constructor. For example, with an arbitration bit rate equal to 727 kbit/s:

```
void setup () {
  ...
  ACANFD_STM32_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_STM32_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_STM32_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_STM32_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, and required sample points. By example, as shown in the figure 5, 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_STM32_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
  ...
}
```

Figure 5 – Adapting property values

Be aware to always respect CAN bit timing consistency! The `NUCLEO-H743ZI2` constraints are:

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

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

Resulting actual bit rates are given by:

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

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

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

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

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

21.2 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_STM32_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 12.

The `ACANFD_STM32_Settings` class defines static constant properties that can be used as mask error. For example:

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

21.3 The `actualArbitrationBitRate` method

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

```
void setup () {
  ...
  ACANFD_STM32_Settings settings (440 * 1000, DataBitRateFactor::x1) ;
```

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 12 – The ACANFD_STM32_Settings::CANBitSettingConsistency method error codes

```
  Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (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 21.2 page 46), the returned value is irrelevant.

21.4 The exactArbitrationBitRate method

```
bool ACANFD_STM32_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 21.2 page 46), the returned value is irrelevant.

21.5 The exactDataBitRate method

```
bool ACANFD_STM32_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 21.2 page 46), the returned value is irrelevant.

21.6 The `ppmFromDesiredArbitrationBitRate` method

```
uint32_t ACANFD_STM32_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): $1\ \text{ppm} = 10^{-6}$. In other words, $10,000\ \text{ppm} = 1\%$.

Note. If CAN bit settings are not consistent (see section 21.2 page 46), the returned value is irrelevant.

21.7 The `ppmFromDesiredDataBitRate` method

```
uint32_t ACANFD_STM32_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): $1\ \text{ppm} = 10^{-6}$. In other words, $10,000\ \text{ppm} = 1\%$.

Note. If CAN bit settings are not consistent (see section 21.2 page 46), the returned value is irrelevant.

21.8 The `arbitrationSamplePointFromBitStart` method

```
uint32_t ACANFD_STM32_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): $1\ \text{ppc} = 1\% = 10^{-2}$. 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 21.2 page 46), the returned value is irrelevant.

21.9 The `dataSamplePointFromBitStart` method

```
uint32_t ACANFD_STM32_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): $1\ \text{ppc} = 1\% = 10^{-2}$. 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 21.2 page 46), the returned value is irrelevant.

21.10 Properties of the `ACANFD_STM32_Settings` class

All properties of the `ACANFD_STM32_Settings` class are declared `public` and are initialized (table 13).

21.10 Properties of the ACANFD_STM32_Settings class

Property	Type	Initial value	Comment
mDesiredArbitrationBitRate	uint32_t	Constructor argument	
mDataBitRateFactor	DataBitRateFactor	Constructor argument	
mBitRatePrescaler	uint8_t	32	See section 21.1 page 42
mArbitrationPhaseSegment1	uint16_t	256	See section 21.1 page 42
mArbitrationPhaseSegment2	uint8_t	128	See section 21.1 page 42
mArbitrationSJW	uint8_t	128	See section 21.1 page 42
mDataPhaseSegment1	uint8_t	32	See section 21.1 page 42
mDataPhaseSegment2	uint8_t	16	See section 21.1 page 42
mDataSJW	uint8_t	16	See section 21.1 page 42
mTripleSampling	bool	true	See section 21.1 page 42
mBitSettingOk	bool	true	See section 21.1 page 42
mModuleMode	ModuleMode	NORMAL_FD	See section 21.10.1 page 49
mDriverReceiveFIFO0Size	uint16_t	10	See section 15.1 page 30
mHardwareRxFIFO0Size	uint8_t	64	See section 13 page 25
mHardwareRxFIFO0Payload	Payload	PAYLOAD_64_BYTES	See section 13 page 25
mDriverReceiveFIFO1Size	uint16_t	0	See section 15.1 page 30
mHardwareRxFIFO1Size	uint8_t	0	See section 13 page 25
mHardwareRxFIFO1Payload	Payload	PAYLOAD_64_BYTES	See section 13 page 25
mEnableRetransmission	bool	true	See section 21.10.2 page 50
mDiscardReceivedStandardRemoteFrames	bool	false	See section 16 page 31
mDiscardReceivedExtendedRemoteFrames	bool	false	See section 16 page 31
mNonMatchingStandardFrameReception	FilterAction	FIFO0	See section 16 page 31
mNonMatchingExtendedFrameReception	FilterAction	FIFO0	See section 16 page 31
mTransceiverDelayCompensation	uint8_t	5	See section 21.10.3 page 50
mDriverTransmitFIFOSize	uint8_t	20	See section 8 page 22
mHardwareTransmitTxFIFOSize	uint8_t	24	See section 8 page 22
mHardwareDedicacedTxBufferCount	uint8_t	8	See section 9 page 23
mHardwareTransmitBufferPayload	Payload	PAYLOAD_64_BYTES	See section 12 page 24
mNonMatchingStandardMessageCallBack	ACANFDCallBackRoutine	nullptr	See section 17.1 page 38
mNonMatchingExtendedMessageCallBack	ACANFDCallBackRoutine	nullptr	See section 17.2 page 38

Table 13 – Properties of the ACANFD_STM32_Settings class

21.10.1 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, BUS_MONITORING.

BUS_MONITORING mode. See DS60001507G datasheet, section 39.6.2.6 page 1096.

In Bus Monitoring Mode (see ISO 11898-1, 10.12 Bus monitoring), the CAN is able to receive valid data frames and valid remote frames, but cannot start a transmission. In this mode, it sends only recessive bits on the CAN bus. If the CAN is required to send a dominant bit (ACK bit, overload flag, active error flag), the bit is rerouted internally so that the CAN monitors this dominant bit, although the CAN bus may remain in recessive state. In Bus Monitoring Mode register TXBRP is held in reset state. The Bus Monitoring Mode can be used to analyze the traffic on a CAN bus without affecting it by the transmission of dominant bits. The figure below shows the connection of signals CAN_TX and CAN_RX to the CAN in Bus Monitoring Mode.

INTERNAL_LOOP_BACK mode. See DS60001507G datasheet, section 39.6.2.8 page 1098.

This mode can be used for a "Hot Selftest", meaning the CAN can be tested without affecting a running CAN system connected to the pins CAN_TX and CAN_RX. In this mode pin CAN_RX is disconnected from the CAN and pin CAN_TX *is held recessive.*

`EXTERNAL_LOOP_BACK` mode. See `DS60001507G` datasheet, section 39.6.2.8 page 1098.

In this Mode, the CAN treats its own transmitted messages as received messages and stores them (if they pass acceptance filtering) into an Rx Buffer or an Rx FIFO. This mode is provided for hardware self-test. To be independent from external stimulation, the CAN ignores acknowledge errors (recessive bit sampled in the acknowledge slot of a data/remote frame) in Loop Back Mode. In this mode the CAN performs an internal feedback from its Tx output to its Rx input. The actual value of the CAN_RX input pin is disregarded by the CAN. The transmitted messages can be monitored at the CAN_TX pin.

21.10.2 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`.

21.10.3 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_STM32_Settings` constructor.

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

22 Other `ACANFD_STM32` methods

22.1 The `getStatus` method

```
ACANFD_STM32::Status ACANFD_STM32::getStatus (void) const ;
```

22.1.1 The `txErrorCount` method

```
uint16_t ACANFD_STM32::Status::txErrorCount (void) const ;
```

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

22.1.2 The `rxErrorCount` method

```
uint8_t ACANFD_STM32::Status::rxErrorCount (void) const ;
```

This method returns the *Receive Error Counter* value.

22.1.3 The isBusOff method

```
bool ACANFD_STM32::Status::isBusOff (void) const ;
```

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

22.1.4 The transceiverDelayCompensationOffset method

```
uint8_t ACANFD_STM32::Status::transceiverDelayCompensationOffset (void) const ;
```

This method returns *Transceiver Delay Compensation Offset* value.

22.1.5 The hardwareTxBufferPayload method

```
ACANFD_STM32_Settings::Payload ACANFD_STM32::hardwareTxBufferPayload (void) const ;
```

This method returns the payload of transmit TxBuffers.

22.1.6 The hardwareRxFIFO0Payload method

```
ACANFD_STM32_Settings::Payload ACANFD_STM32::hardwareRxFIFO0Payload (void) const ;
```

This method returns the payload of hardware receive FIFO 0.

22.1.7 The hardwareRxFIFO1Payload method

```
ACANFD_STM32_Settings::Payload ACANFD_STM32::hardwareRxFIFO1Payload (void) const ;
```

This method returns the payload of hardware receive FIFO 1.