

ACANFD_STM32 **Arduino library, for** NUCLEO-G431KB, NUCLEO-G474RE, WeActStudio-G474CE, NUCLEO-H723ZG NUCLEO-H743ZI2 **boards** **Version 1.1.2-rc1**

Pierre Molinaro

March 16, 2025

Contents

1	Versions	5
2	Features	5
3	Supported boards	6
3.1	NUCLEO-G0B1RE	6
3.2	NUCLEO-G431KB	6
3.3	NUCLEO-G474RE	6
3.4	WeActStudio G474CE	8
3.5	NUCLEO-H723ZG	8
3.6	NUCLEO-H743ZI2	8
4	Data flow	9
4.1	NUCLEO-G431KB, NUCLEO-G474RE, WeActStudio-G474CE	9
4.2	NUCLEO-H723ZG, NUCLEO-H743ZI2	10
5	A sample sketch: <i>board-LoopBackDemo</i>	12
5.1	Including <ACANFD_STM32.h>	12
5.2	The setup function	13
5.3	The global variables	14
5.4	The loop function	14
6	The CANMessage class	15

CONTENTS

7	The CANFDMessage class	16
7.1	Properties	17
7.2	The default constructor	17
7.3	Constructor from CANMessage	17
7.4	The type property	18
7.5	The len property	19
7.6	The idx property	19
7.7	The pad method	19
7.8	The isValid method	19
8	Modifying FDCAN Clock	20
8.1	Why define custom system clock configuration?	20
8.2	Define custom system clock configuration	20
8.2.1	Find the SystemClock_Config function for your board	21
8.2.2	Understand the original settings	22
8.2.3	Adapt the SystemClock_Config function settings	23
9	Transmit FIFO	24
9.1	The driverTransmitFIFOSize method	25
9.2	The driverTransmitFIFOCount method	25
9.3	The driverTransmitFIFOPeakCount method	25
10	Transmit buffers (TxBuffer_i)	26
11	Transmit Priority	26
12	Receive FIFOs	26
13	Payload size	26
13.1	The ACANFD_STM32_Settings::wordCountForPayload static method	27
13.2	The ACANFD_STM32_Settings::frameDataByteCountForPayload static method	27
13.3	Changing the default payloads	28
14	Message RAM	28
15	The poll method	30
16	Sending frames: the tryToSendReturnStatusFD method	30
16.1	Testing a send buffer: the sendBufferNotFullForIndex method	31
16.2	Usage example	31
17	Retrieving received messages using the receiveFD_i method	32
17.1	Driver receive FIFO <i>i</i> size	33
17.2	The driverReceiveFIFO _i Size method	33
17.3	The driverReceiveFIFO _i Count method	34
17.4	The driverReceiveFIFO _i PeakCount method	34
17.5	The resetDriverReceiveFIFO _i PeakCount method	34

18 Acceptance filters	34
18.1 Acceptance filters for standard frames	34
18.1.1 Defining standard frame filters	35
18.1.2 Add single filter	36
18.1.3 Add dual filter	36
18.1.4 Add range filter	36
18.1.5 Add classic filter	37
18.2 Acceptance filters for extended frames	38
18.2.1 Defining extended frame filters	38
18.2.2 Add single filter	39
18.2.3 Add dual filter	40
18.2.4 Add range filter	40
18.2.5 Add classic filter	40
19 The dispatchReceivedMessage method	41
19.1 Dispatching non matching standard frames	42
19.2 Dispatching non matching extended frames	42
20 The dispatchReceivedMessageFIFO0 method	43
21 The dispatchReceivedMessageFIFO1 method	44
22 The ACANFD_STM32::beginFD method reference	44
22.1 The prototypes	44
22.2 The error codes	45
22.2.1 The kTxBufferCountGreaterThan32 error code	45
23 ACANFD_STM32_Settings class reference	46
23.1 The ACANFD_STM32_Settings constructors: computation of the CAN bit settings	46
23.1.1 5 arguments constructor	46
23.1.2 3-arguments constructor	47
23.1.3 Exact bit rates	47
23.2 The CANBitSettingConsistency method	50
23.3 The actualArbitrationBitRate method	51
23.4 The exactArbitrationBitRate method	51
23.5 The exactDataBitRate method	52
23.6 The ppmFromDesiredArbitrationBitRate method	52
23.7 The ppmFromDesiredDataBitRate method	52
23.8 The arbitrationSamplePointFromBitStart method	52
23.9 The dataSamplePointFromBitStart method	53
23.10 Properties of the ACANFD_STM32_Settings class	53
23.10.1 The mModuleMode property	53
23.10.2 The mEnableRetransmission property	54
23.10.3 The mTransceiverDelayCompensation property	54

24 Other ACANFD_STM32 methods	55
24.1 The <code>getStatus</code> method	55
24.1.1 The <code>txErrorCount</code> method	55
24.1.2 The <code>rxErrorCount</code> method	55
24.1.3 The <code>isBusOff</code> method	55
24.1.4 The <code>transceiverDelayCompensationOffset</code> method	55
24.1.5 The <code>hardwareTxBufferPayload</code> method	55
24.1.6 The <code>hardwareRxFIFO0Payload</code> method	56
24.1.7 The <code>hardwareRxFIFO1Payload</code> method	56
25 Porting guide	56
25.1 Microcontroller with fixed size message RAM CANFD modules	56
25.1.1 The new <code>ACANFD_STM32_NUCLEO_GOB1RE-objects.h</code> file	57
25.1.2 The new <code>ACANFD_STM32_NUCLEO_GOB1RE-settings.h</code> file	60
25.1.3 Modify the <code>ACANFD_STM32_from_cpp.h</code> file	61
25.1.4 Modify the <code>ACANFD_STM32.h</code> file	61
25.1.5 Modify the <code>ACANFD_STM32_Settings.h</code> file	62
25.1.6 The <code>GOB1RE-LoopBackDemo</code> sample sketch	63
25.1.7 Checking bit rate	64
25.1.8 Adding interrupt handling	64
25.1.9 Checking pin assignments	66
25.2 Microcontroller with programmable size message RAM CANFD modules . .	66

1 Versions

Version	Date	Comment
1.1.2-rc1	March 16, 2025	Experimental support WeActStudio G474 board.
1.1.1	May 25, 2024	Removed <code>mTripleSampling</code> property of <code>ACANFD_STM32_Settings</code> , as triple sampling is not implemented by the hardware.
1.1.0	October 2, 2023	Added NUCLEO-H723ZG board.
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-G0B1RE¹, NUCLEO-G431KB², the NUCLEO-G474RE³, the NUCLEO-H723ZG⁴ and the NUCLEO-H743ZI2⁵ boards running STM32duino. It handles CANFD frames.

This library is compatible with other ACAN libraries 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;

¹<https://www.st.com/en/evaluation-tools/nucleo-g0b1re.html>

²<https://www.st.com/en/evaluation-tools/nucleo-g431kb.html>

³<https://www.st.com/en/evaluation-tools/nucleo-g474re.html>

⁴<https://www.st.com/en/evaluation-tools/nucleo-h723zg.html>

⁵https://www.st.com/resource/en/user_manual/um2407-stm32h7-nucleo144-boards-mb1364-stmicroelectronics.pdf

-
- the message RAM allocation is customizable and the driver checks no overflow occurs (only NUCLEO-H723ZG, NUCLEO-H743ZI2);
 - *internal loop back, external loop back* controller modes are selectable.

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.

3 Supported boards

The currently supported boards are:

- NUCLEO-G0B1RE ([section 3.1 page 6](#));
- NUCLEO-G431KB ([section 3.2 page 6](#));
- NUCLEO-G474RE ([section 3.3 page 6](#));
- NUCLEO-H723ZG ([section 3.5 page 8](#));
- NUCLEO-H743ZI2 ([section 3.6 page 8](#)).

For NUCLEO-G0B1RE, NUCLEO-G431KB and NUCLEO-G474RE, the message RAM sections are statically allocated, for a total word size equal to 212 (848 bytes).

For NUCLEO-H723ZG and NUCLEO-H743ZI2, the message RAM section 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.

3.1 NUCLEO-G0B1RE

The NUCLEO-G0B1RE contains two CANFD modules `canfd1` and `canfd2` ([table 2](#)). **Note FDCAN interrupts are not handled for this microcontroller, so you should call the `poll` method, see [section 15 page 30](#) and demo sketch `G0B1RE-LoopBackDemo`.**

3.2 NUCLEO-G431KB

The NUCLEO-G431KB contains one CANFD module `canfd1` ([table 3](#)).

3.3 NUCLEO-G474RE

The NUCLEO-G474RE contains three CANFD modules `canfd1`, `canfd2` and `canfd3` ([table 4](#)).

Name	fdcan1	fdcan2
Default FDCAN Clock	<i>64 MHz, common to the two CANFD modules</i>	
Default TxPin	PA12	PB1
Alternate TxPins	PB9, PC5, PD1	PB6, PB13, PC3
Default RxPin	PA11	PB0
Alternate RxPins	PB8, PC4, PD0	PB5, PB12, PC2
Message RAM Size	212 words	212 words
Standard Receive filters	28 (28 words)	28 (28 words)
Extended Receive filters	8 (16 words)	8 (16 words)
Rx FIFO0	3 (54 words)	3 (54 words)
Rx FIFO1	3 (54 words)	3 (54 words)
Tx Buffers	3 (54 words)	3 (54 words)

Table 2 – The two CANFD modules of NUCLEO-G0B1RE

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 (28 words)
Extended Receive filters	8 (16 words)
Rx FIFO0	3 (54 words)
Rx FIFO1	3 (54 words)
Tx Buffers	3 (54 words)

Table 3 – The CANFD module of NUCLEO-G431KB

Name	fdcan1	fdcan2	fdcan3
Default FDCAN Clock	<i>168 MHz, common to the three CANFD modules</i>		
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 (28 words)	28 (28 words)	28 (28 words)
Extended Receive filters	8 (16 words)	8 (16 words)	0-8 (16 words)
Rx FIFO0	3 (54 words)	3 (54 words)	3 (54 words)
Rx FIFO1	3 (54 words)	3 (54 words)	3 (54 words)
Tx Buffers	3 (54 words)	3 (54 words)	3 (54 words)

Table 4 – The three CANFD modules of NUCLEO-G474RE

3.4 WeActStudio G474CE

The WeActStudio G474RE board contains three CANFD modules `canfd1`, `canfd2` and `canfd3` ([table 5](#)).

Name	fdcan1	fdcan2	fdcan3
Default FDCAN Clock	<i>170 MHz, common to the three CANFD modules</i>		
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 (28 words)	28 (28 words)	28 (28 words)
Extended Receive filters	8 (16 words)	8 (16 words)	0-8 (16 words)
Rx FIFO0	3 (54 words)	3 (54 words)	3 (54 words)
Rx FIFO1	3 (54 words)	3 (54 words)	3 (54 words)
Tx Buffers	3 (54 words)	3 (54 words)	3 (54 words)

Table 5 – The three CANFD modules of WeActStudio G474CE

3.5 NUCLEO-H723ZG

The NUCLEO-H723ZG contains three CANFD modules `canfd1`, `canfd2` and `canfd3` ([table 6](#)).

Name	fdcan1	fdcan2	fdcan3
Default FDCAN Clock	<i>120 MHz, common to the two CANFD modules</i>		
Default TxPin	PD_1	PB_6	PF_7
Alternate TxPin	PB_9, PA_12	PB_13	PD_13, PG_9
Default RxPin	PD_0	PB_5	PF_6
Alternate RxPin	PB_8, PA_11	PB_12	PD_12, PG_10
Message RAM Size	<i>2560 words, shared between the two CANFD modules</i>		
Standard Receive filters	0-128 (0-128 words)	0-128	0-128 (0-128 words)
Extended Receive filters	0-64 (0-128 words)	0-64	0-64 (0-128 words)
Rx FIFO0	0-64 (0-1152 words)	0-64	0-64 (0-1152 words)
Rx FIFO1	0-64 (0-1152 words)	0-64	0-64 (0-1152 words)
Tx Buffers	0-32 (0-576 words)	0-32	0-32 (0-576 words)

Table 6 – The three CANFD modules of NUCLEO-H723ZG

3.6 NUCLEO-H743ZI2

The NUCLEO-H743ZI2 contains two CANFD modules `canfd1` and `canfd2` ([table 7](#)).

Name	fdcan1	fdcan2
Default FDCAN Clock	<i>120 MHz, common to the two CANFD modules</i>	
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	<i>2560 words, shared between the two CANFD modules</i>	
Standard Receive filters	0-128 (0-128 words)	0-128 (0-128 words)
Extended Receive filters	0-64 (0-128 words)	0-64 (0-128 words)
Rx FIFO0	0-64 (0-1152 words)	0-64 (0-1152 words)
Rx FIFO1	0-64 (0-1152 words)	0-64 (0-1152 words)
Tx Buffers	0-32 (0-576 words)	0-32 (0-576 words)

Table 7 – The two CANFD modules of NUCLEO-H743ZI2

4 Data flow

4.1 NUCLEO-G431KB, NUCLEO-G474RE, WeActStudio-G474CE

The data flow is given in [figure 1](#).

Sending messages. The `ACANFD_STM32` driver defines a *driver transmit FIFO* (default size: 20 messages). The module *hardware transmit FIFO* has a fixed 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 16 page 30](#) 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 16.1 page 31](#)) 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 18 page 34](#) 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 FIFOi*. 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 17.1 page 33](#) 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 17 page 32](#);
- the `availableFD1` method returns `false` if the *Driver Receive FIFO1* is empty, and `true` otherwise;

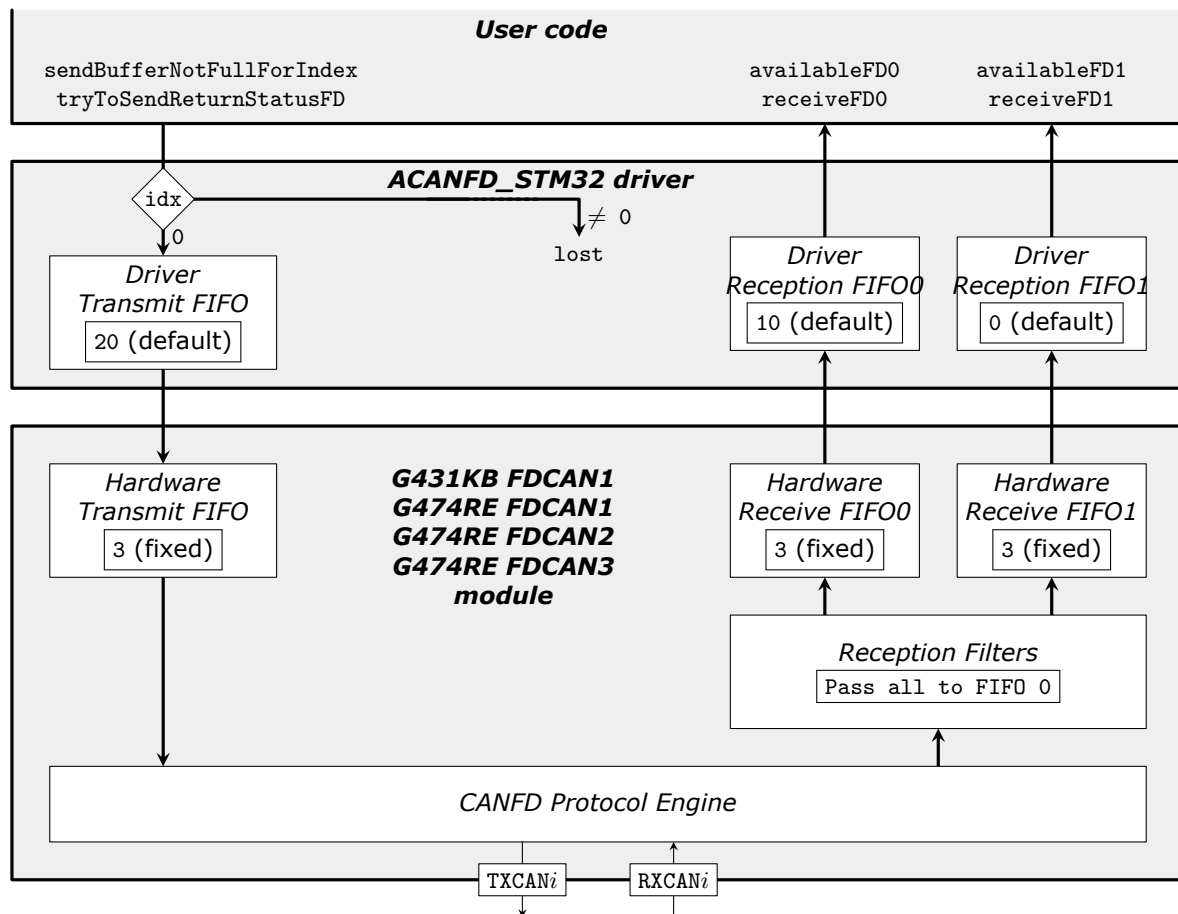


Figure 1 – NUCLEO-G431KB, NUCLEO-G474RE: message flow in ACANFD_STM32 driver and FDCAN_i module

otherwise;

- the `receiveFD1` method retrieves messages from the *Driver Receive FIFO1* – see [section 17 page 32](#).

4.2 NUCLEO-H723ZG, NUCLEO-H743ZI2

The data flow is 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 16 page 30](#) 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*;

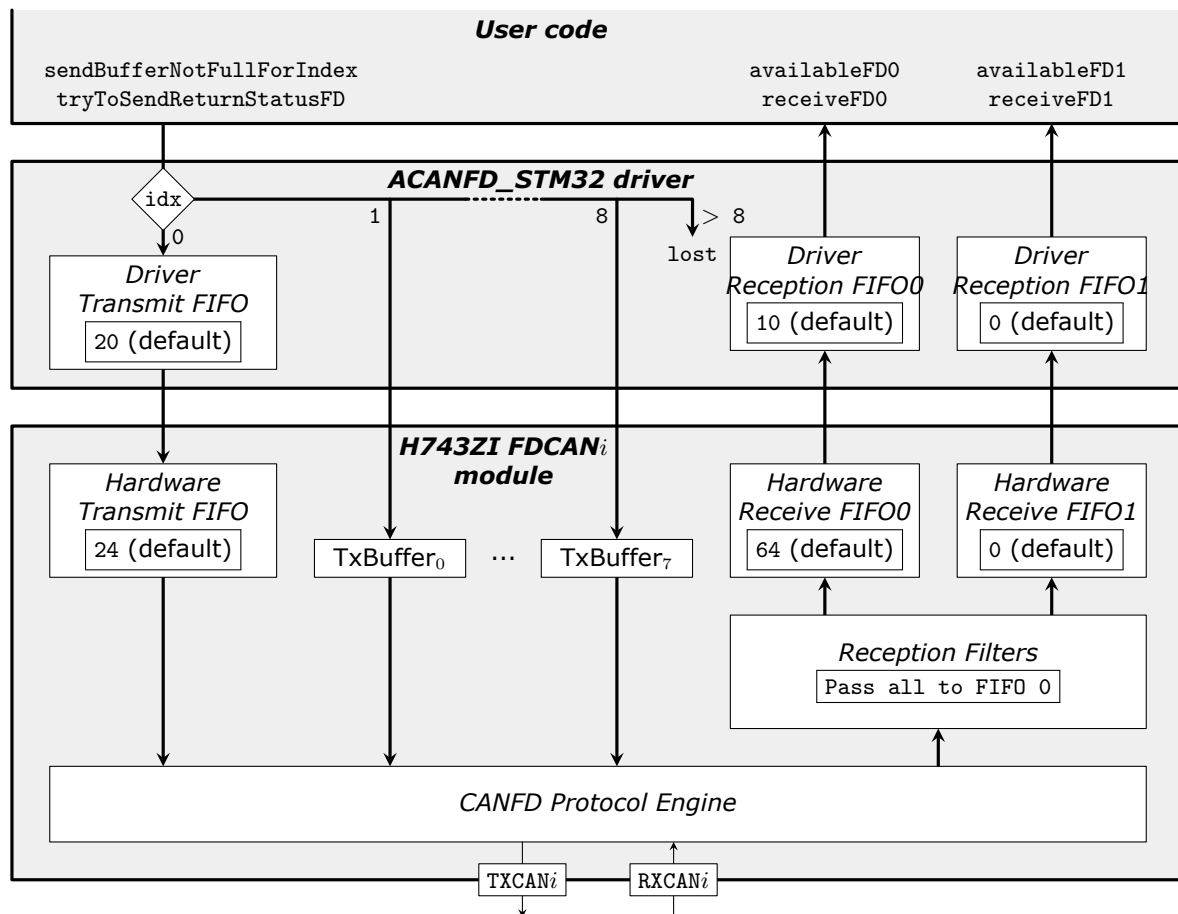


Figure 2 – NUCLEO-H723ZG, NUCLEO-H743ZI2: message flow in ACANFD_STM32 driver and FDCAN_i module

- 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 16.1 page 31](#)) 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 18 page 34](#) 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 FIFO_i to the *Driver Receive FIFO*_i. The size of the *Driver Receive FIFO 0* is 10 by default – see [section 17.1 page 33](#) 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 17 page 32](#);
- 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 17 page 32](#).

5 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. These codes run without any additional CAN hardware, as the `FDCANi` modules are configured in `EXTERNAL_LOOP_BACK` mode (see [section 23.10.1 page 53](#)); the `FDCANi` module receives every CANFD frame it sends, and emitted frames can be observed on its `TxPin`.

5.1 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 `fdcani`, include `ACANFD_STM32_from_cpp.h` header.

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

The `NUCLEO-H723ZG` and `NUCLEO-H743ZI2` are 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`;
- (`NUCLEO-H723ZG` only) the `FDCAN3_MESSAGE_RAM_WORD_SIZE` constant define the word size allocated to `fdcan3`.

For example for the `NUCLEO-H743ZI2`:

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

#include <ACANFD_STM32.h>
```

5.2 The setup function

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

5.2 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 instantiates 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 if you have no connection to a physical CAN network. The [section 23.10 page 53](#) 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 18 page 34](#).

5.3 The global variables

```
...
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 22.2 page 45](#).

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) ;
}
...
```

5.3 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 successfully received messages.

5.4 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.receiveFDO (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 driver⁶, the ACAN2517 driver⁷ and the ACAN2517FD driver⁸ 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

⁶The ACAN2515 driver is a CAN driver for the MCP2515 CAN controller, <https://github.com/pierremolinaro/acan2515>.

⁷The ACAN2517 driver is a CAN driver for the MCP2517FD CAN controller in CAN 2.0B mode, <https://github.com/pierremolinaro/acan2517>.

⁸The ACAN2517FD driver is a CANFD driver for the MCP2517FD CAN controller in CANFD mode, <https://github.com/pierremolinaro/acan2517FD>.

0, and without any data. In this library, the `CANMessage` class is only used by a `CANFDMessage` constructor (section 7.3 page 17).

```
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} ;
} ;
} ;
```

Note the message datas are defined by an `union`. So message datas can be seen as eight 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 19 page 41) or 255 if it does not correspond to any filter;
- on sending messages, it is used for selecting the transmit buffer (see section 16 page 30).

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` driver⁹ 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.

⁹The `ACAN2517FD` driver is a CANFD driver for the `MCP2517FD` CAN controller in CANFD mode, <https://github.com/pierremolinaro/acan2517FD>.

7.1 Properties

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

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

7.1 Properties

```
class CANFDMessage {
    ...
    public : uint32_t id; // Frame identifier
    public : bool ext ; // false -> base frame, true -> extended frame
    public : Type type ;
    public : uint8_t idx ; // Used by the driver
    public : uint8_t len ; // Length of data (0 ... 64)
    public : union {
        uint64_t data64 [ 8] ; // Caution: subject to endianness
        uint32_t data32 [16] ; // Caution: subject to endianness
        uint16_t data16 [32] ; // Caution: subject to endianness
        float dataFloat [16] ; // Caution: subject to endianness
        uint8_t data [64] ;
    } ;
    ...
} ;
```

Note the message datas are defined by an `union`. So message datas can be seen as 64 bytes, 32 x 16-bit unsigned integers, 16 x 32-bit, 8 x 64-bit or 16 x 32-bit floats. Be aware that multi-byte integers are subject to endianness (STM32 processors 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 8](#)).

7.3 Constructor from CANMessage

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

7.4 The type property

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	-	<i>uninitialized</i>

Table 8 – CANFDMessage default constructor initialization

```
...  
};
```

All properties are initialized from the `inCANMessage` ([table 9](#)). Note that only `data64[0]` is initialized from `inCANMessage.data64`.

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

Table 9 – 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 10](#).

7.5 The len property

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 10 – 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_STM32::tryToSendReturnStatusFD` method. You can use the `pad` method (see [section 7.7 page 19](#)) 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 16 page 30](#)).

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 constraints on the `len` property are checked, as indicated the [table 10 page 19](#), and `false` otherwise.

8 Modifying FDCAN Clock

8.1 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 4 page 7](#)).

From the G474RE-LoopBackDemo.ino sketch, the settings object is instantiated 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.

8.2 Define custom system clock configuration

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

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

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

8.2 Define custom system clock configuration

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

8.2.1 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_SYCLK  
                                | 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();  
    }  
}
```

8.2 Define custom system clock configuration

```
#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.

8.2.2 Understand the original settings

We have to understand how the original settings provide a `FDCLAN` clock of 168 MHz. A very very simplified explanation 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: $PCLK1 = 24\text{ MHz} * 28 / 2 / 2 = 168\text{ MHz}$.

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).

8.2 Define custom system clock configuration

8.2.3 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: $PCLK1 = 24 \text{ MHz} * 40 / 3 / 2 = 160 \text{ MHz}$

PCLK1=165 MHz. Choose PLLM=8 and PLLN=110: $PCLK1 = 24 \text{ MHz} * 110 / 8 / 2 = 165 \text{ MHz}$

PCLK1=170 MHz. Choose PLLM=6 and PLLN=85: $PCLK1 = 24 \text{ MHz} * 85 / 6 / 2 = 170 \text{ MHz}$

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.

9 Transmit FIFO

The transmit FIFO (see [figure 1 page 10](#) and [figure 2 page 11](#)) 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:

9.1 The driverTransmitFIFOSize method

- 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 through 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.

9.1 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 ();
```

9.2 The driverTransmitFIFOCount method

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

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

9.3 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 `kTransmitBufferOverflow`. In such case, the following calls of `driverTransmitBufferPeakCount()` will return `driverTransmitFIFOSize ()+1`.

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

10 Transmit buffers (TxBuffer_i)

Transmit buffers are only available for NUCLEO-H743ZI2. There are `settings.mHardwareDedicacedTxBuffer` 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 16.1 page 31](#)) for testing if a TxBuffer is empty or full.

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

11 Transmit Priority

Pending dedicaced TxBuffer_i and oldest pending Tx FIFO buffer are scanned, and buffer with lowest message identifier gets highest priority and is transmitted next.

12 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 ($0 \leq i \leq 1$, see [figure 2 page 11](#) and [figure 1 page 10](#)) is composed by:

- the *hardware receive FIFO_i* (in the Message RAM, see [section 14 page 28](#)), 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 `mHardwareRxFIFOiSize` 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 `mDriverReceiveFIFOiSize` property of your `settings` object.

The receive FIFO mechanism ensures sequentiality of reception.

13 Payload size

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

13.1 The ACANFD_STM32_Settings::wordCountForPayload static method

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 14 page 28](#). 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 11](#). The type of these properties is the ACANFD_STM32_Settings::Payload enumeration type, and defines 8 values ([table 12](#)).

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 11 – 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 12 – ACANFD_STM32_Settings object size from payload size specification

13.1 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 12](#).

13.2 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 12](#).

13.3 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 retrieved 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.

14 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 contains¹⁰:

- standard filters (0-128 elements, 0-128 words);

¹⁰See DS60001507G, section 39.9.1 page 1177.

-
- 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 12](#));
- 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 12](#));
- 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 12](#)).

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`.

15 The poll method

If the library does not implement FDCAN interrupt handling (as on the STM320G1RE), you should call the `poll` method as often as possible, typically at the start of the `loop` function. See the `GOB1RE-LoopBackDemo` sketch.

For other microcontrollers, the library implements FDCAN interrupt handling, calling the `poll` method has no effect.

16 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 7.8 page 19](#));
- `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 9 page 24](#)) ;
- 1 ... `settings.mHardwareDedicatedTxBufferCount` for a dedicated `TxBuffer` ([section 10 page 26](#)).

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_xn` 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.

16.1 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 13).

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 13 – Value returned by the `sendBufferNotFullForIndex` method

16.2 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 ;
    }
}
...
}

```

17 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 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:

17.1 Driver receive FIFO *i* size

```
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.

17.1 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`.

17.2 The `driverReceiveFIFOiSize` method

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

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

17.3 The driverReceiveFIFO*i*Count method

17.3 The driverReceiveFIFO*i*Count method

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

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

17.4 The driverReceiveFIFO*i*PeakCount 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*PeakCount ()+1.

17.5 The resetDriverReceiveFIFO*i*PeakCount method

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

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

18 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.

18.1 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_FeatherM4-CAN_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 18.1.1 page 35](#)), up to 128, none by default.

18.1 Acceptance filters for standard frames

The standard frame filtering is illustrated by [figure 3](#).

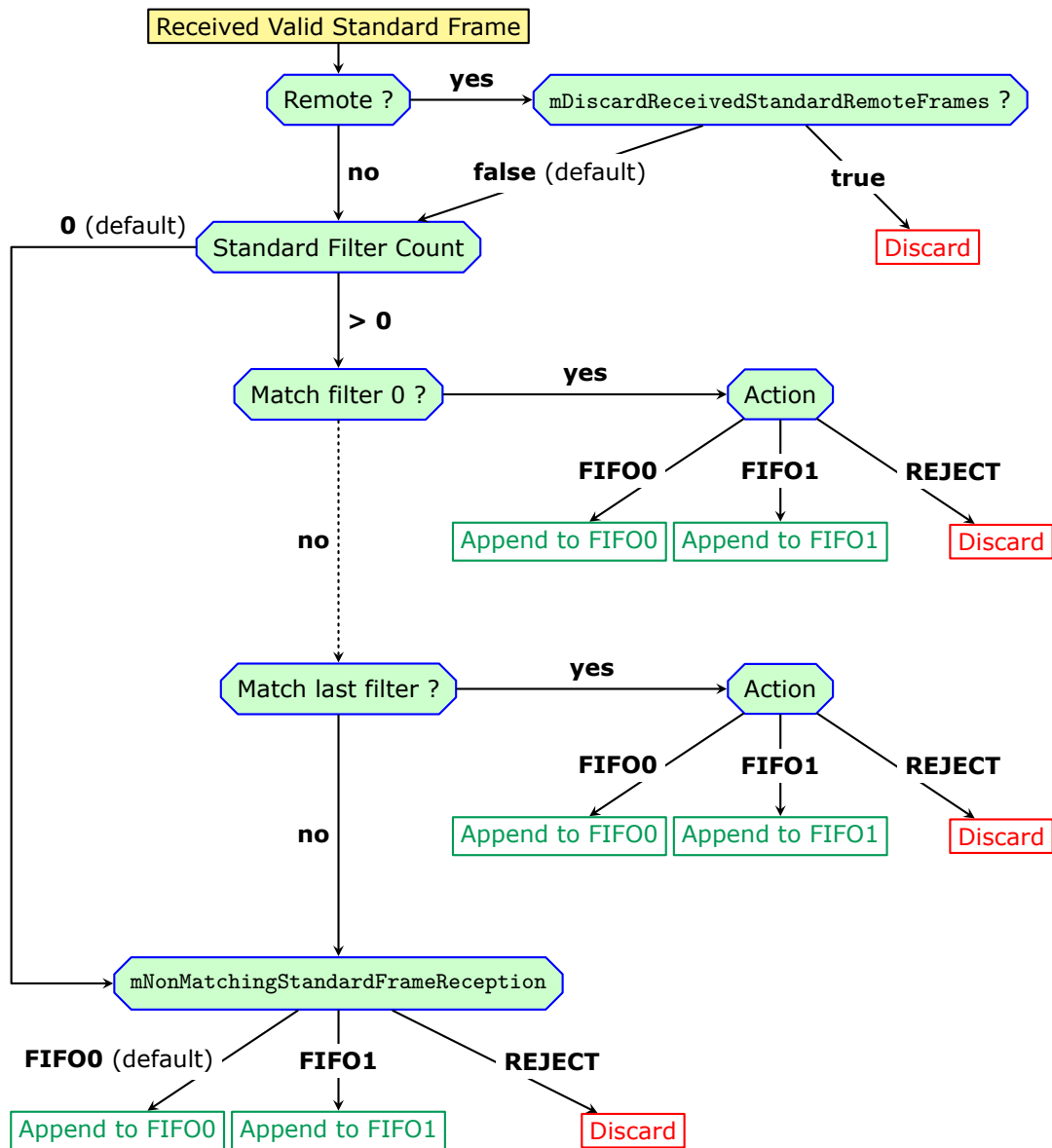


Figure 3 – Standard frame filtering

18.1.1 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;
...
```

18.1 Acceptance filters for standard frames

```
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 18.1.2 page 36](#)), `addDual` ([section 18.1.3 page 36](#)), `addRange` ([section 18.1.4 page 36](#)) or `addClassic` ([section 18.1.5 page 37](#)) 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).

18.1.2 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 19 page 41](#).

18.1.3 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 19 page 41](#).

18.1.4 Add range filter

18.1 Acceptance filters for standard frames

```
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 19 page 41](#).

18.1.5 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 19 page 41](#).

For example:

```
standardFilters.addClassic (0x405, 0x7D5, ACANFD_STM32_FilterAction::FIF00) ;
```

18.2 Acceptance filters for extended frames

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.

18.2 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 18.2.1 page 38](#)), up to 128, none by default.

The extended frame filtering is illustrated by [figure 4](#).

18.2.1 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 18.2.2 page 39](#)), `addDual` ([section 18.2.3 page 40](#)), `addRange` ([section 18.2.4 page 40](#)) or `addClassic` ([section 18.2.5 page 40](#)) methods. Then, add the `ACANFD_STM32_ExtendedFilters` as second argument of `beginFD` call.

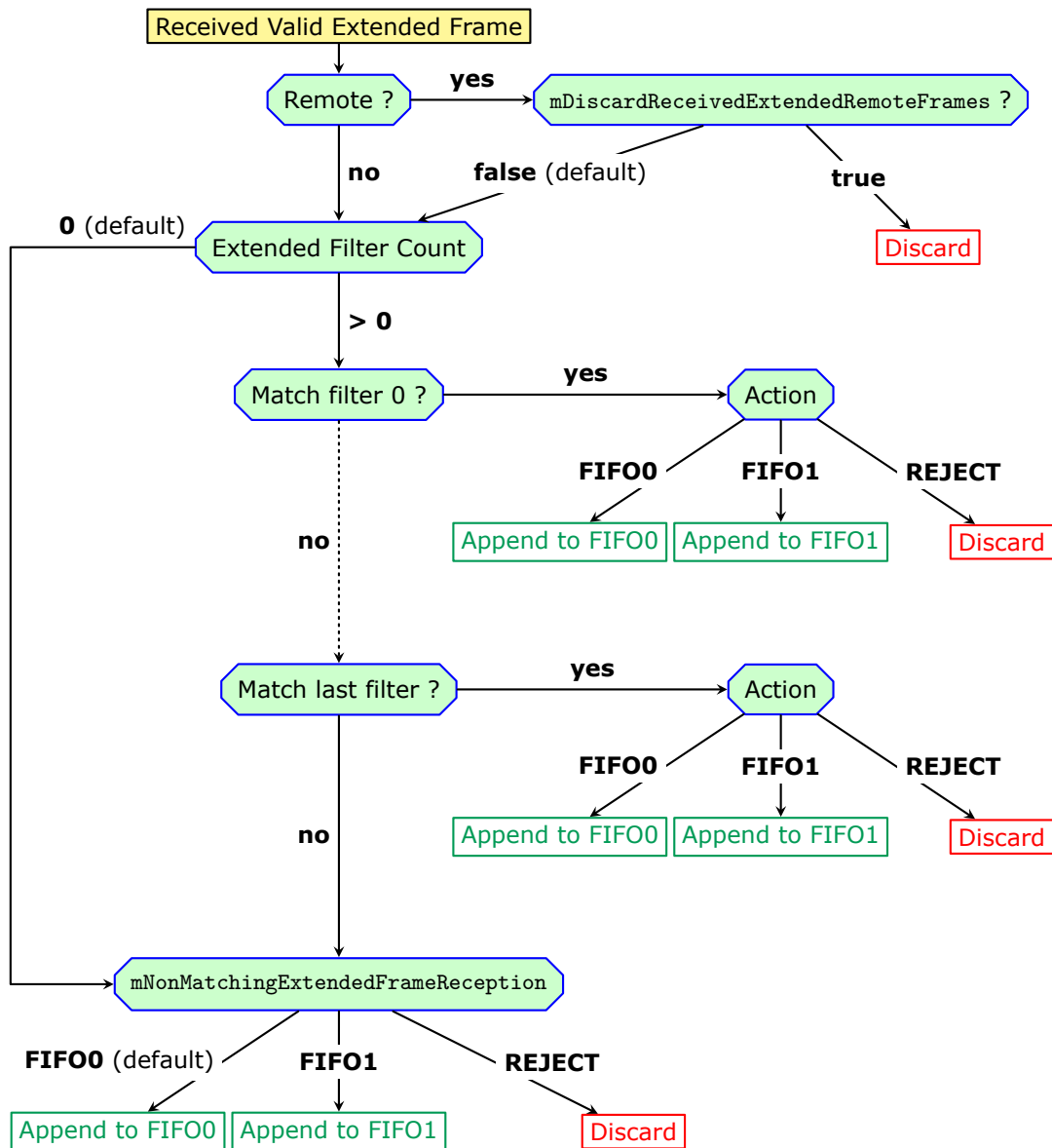


Figure 4 – Extended frame filtering

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).

18.2.2 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

18.2 Acceptance filters for extended frames

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 19 page 41](#).

18.2.3 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 19 page 41](#).

18.2.4 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 19 page 41](#).

18.2.5 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 19 page 41](#).

For example:

```
extendedFilters.addClassic (0x6789, 0x1FFF67BD, ACANFD_STM32_FilterAction::FIF00) ;
```

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
<code>inIdentifier: 0x6789</code>	0		0	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1
<code>inMask: 0x1FFF67BD</code>	1		0	1	1	0	0	1	1	1	1	0	1	1	1	1	0	1
Matching identifiers	0		<i>x</i>	1	1	<i>x</i>	<i>x</i>	1	1	1	1	<i>x</i>	1	1	1	0	<i>x</i>	1

Therefore there are 32 matching identifiers.

19 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 `FIF00` and `FIF01`, 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
    ...
}
```

19.1 Dispatching non matching standard frames

The `dispatchReceivedMessage` method handles one FIF00 message and one FIF01 message on each call. Specifically:

- if FIF00 and FIF01 are both empty, it returns `false`;
- if FIF00 is not empty, its oldest message is extracted and its associated callback is called; then, if FIF01 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 be used for emptying and dispatching all received messages:

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

19.1 Dispatching non matching standard frames

Following the [figure 3 page 35](#), non matching standard frames are stored in FIF00 if `mNonMatchingStandardFrameReception` is equal to FIF00, or in FIF01 if `mNonMatchingStandardFrameReception` is equal to FIF01. As these 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 the `dispatchReceivedMessage` method. By default, `mNonMatchingStandardMessageCallBack` value is `nullptr`.

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

19.2 Dispatching non matching extended frames

Following the [figure 4 page 39](#), non matching extended frames are stored in FIF00 if `mNonMatchingExtendedFrameReception` is equal to FIF00, or in FIF01 if `mNonMatchingExtendedFrameReception` is equal to FIF01. As these 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 ex-

tended 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.

20 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 ...
    }
    ...
}
```

21 The dispatchReceivedMessageFIFO1 method

The `dispatchReceivedMessageFIFO1` method dispatches the messages stored in the FIFO1. The messages stored in FIFO0 are retrieved using the `receiveFIFO0` method.

```
void loop () {
    fdcan1.dispatchReceivedMessageFIFO1 () ; // Do not call fdcan1.receiveFIFO1 any more
    CANFDMessage ;
    if (can1.receiveFIFO0 (message)) {
        ... handle FIFO0 message ...
    }
    ...
}
```

Instead of calling the `receiveFIFO1` 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 `receiveFIFO1` 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 be used for emptying and dispatching all received messages:

```
void loop () {
    while (can1.dispatchReceivedMessageFIFO1 ()) {
    }
    CANFDMessage ;
    if (can1.receiveFIFO0 (message)) {
        ... handle FIFO0 message ...
    }
    ...
}
```

22 The ACANFD_STM32::beginFD method reference

22.1 The prototypes

22.2 The error codes

```
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 18.1 page 34](#)). By default, the standard filter list is empty.

The third one is optional, and specifies the extended filter list (see [section 18.2 page 38](#)). By default, the extended filter list is empty.

22.2 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 14](#). 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 15 page 51](#).

Bit	Code	Static constant Name	Comment
0	0x1	kBitRatePrescalerIsZero	See table 15 page 51
...	See table 15 page 51
16	0x1_0000	kDataSJWIsGreaterThanOrEqualToPhaseSegment2	See table 15 page 51
20	0x10_0000	kMessageRamTooSmall	See section 14 page 28
21	0x20_0000	kMessageRamNotInFirst64kio	See section 14 page 28
22	0x40_0000	kHardwareRxFIFOSizeGreaterThan64	settings.mHardwareRxFIFOSize > 64
23	0x80_0000	kHardwareTransmitFIFOSizeGreaterThan32	settings.mHardwareTransmitTxFIFOSize > 32
24	0x100_0000	kDedicatedTransmitTxBufferCountGreaterThan30	settings.mHardwareDedicatedTxBufferCount > 30
25	0x200_0000	kTxBufferCountGreaterThan32	See section 22.2.1 page 45
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 18.1 page 34
29	0x2000_0000	kExtendedFilterCountGreaterThan128	More than 128 extended filters, see section 18.2 page 38

Table 14 – The `ACANFD_STM32::beginFD` method error code bits

22.2.1 The `kTxBufferCountGreaterThan32` error code

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

23 ACANFD_STM32_Settings class reference

23.1 The ACANFD_STM32_Settings constructors: computation of the CAN bit settings

23.1.1 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 49](#).

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.

23.1 The ACANFD_STM32_Settings constructors: computation of the CAN bit settings

23.1.2 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) ;
```

23.1.3 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
    ...
}
```

23.1 The ACANFD_STM32_Settings constructors: computation of the CAN bit settings

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:_") ;
}
```


23.1 The ACANFD_STM32_Settings constructors: computation of the CAN bit settings

```
Serial.println (settings.dataSamplePointFromBitStart ()) ; // 69, meaning 59%
Serial.print ("Consistency:␣") ;
Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning 0k
...
}
```

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 0k
    ...
}
```

Figure 5 – Adapting property values

Be aware to always respect CAN bit timing consistency! The NUCLE0-H743ZI2 constraints are:

23.2 The CANBitSettingConsistency method

$$\begin{aligned}1 &\leq \text{mBitRatePrescaler} \leq 32 \\1 &\leq \text{mArbitrationPhaseSegment1} \leq 256 \\2 &\leq \text{mArbitrationPhaseSegment2} \leq 128 \\1 &\leq \text{mArbitrationSJW} \leq \text{mArbitrationPhaseSegment2} \\1 &\leq \text{mDataPhaseSegment1} \leq 32 \\2 &\leq \text{mDataPhaseSegment2} \leq 16 \\1 &\leq \text{mDataSJW} \leq \text{mDataPhaseSegment2}\end{aligned}$$

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

Resulting actual bit rates are given by:

$$\begin{aligned}\text{Actual Arbitration Bit Rate} &= \frac{\text{FDCAN_CLOCK}}{\text{mBitRatePrescaler} \cdot (1 + \text{mArbitrationPhaseSegment1} + \text{mArbitrationPhaseSegment2})} \\ \text{Actual Data Bit Rate} &= \frac{\text{FDCAN_CLOCK}}{\text{mBitRatePrescaler} \cdot (1 + \text{mDataPhaseSegment1} + \text{mDataPhaseSegment2})}\end{aligned}$$

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

$$\begin{aligned}\text{Arbitration Sampling Point} &= 100 \cdot \frac{1 + \text{mArbitrationPhaseSegment1}}{1 + \text{mArbitrationPhaseSegment1} + \text{mArbitrationPhaseSegment2}} \\ \text{Data Sampling Point} &= 100 \cdot \frac{1 + \text{mDataPhaseSegment1}}{1 + \text{mDataPhaseSegment1} + \text{mDataPhaseSegment2}}\end{aligned}$$

23.2 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: ") ;
    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 15](#).

23.3 The actualArbitrationBitRate method

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 ;
```

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	-	<i>unused</i>
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 15 – The ACANFD_STM32_Settings::CANBitSettingConsistency method error codes

23.3 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) ;  
    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 23.2 page 50](#)), the returned value is irrelevant.

23.4 The exactArbitrationBitRate method

```
bool ACANFD_STM32_Settings::exactArbitrationBitRate (void) const ;
```

23.5 The exactDataBitRate method

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 23.2 page 50](#)), the returned value is irrelevant.

23.5 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 23.2 page 50](#)), the returned value is irrelevant.

23.6 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 23.2 page 50](#)), the returned value is irrelevant.

23.7 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 23.2 page 50](#)), the returned value is irrelevant.

23.8 The arbitrationSamplePointFromBitStart method

```
float 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%.

23.9 The dataSamplePointFromBitStart method

Note. If CAN bit settings are not consistent (see [section 23.2 page 50](#)), the returned value is irrelevant.

23.9 The dataSamplePointFromBitStart method

```
float 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 ppc = 1% = 10^{-2} . It is a good practice to get sample point from 65% to 80%.

Note. If CAN bit settings are not consistent (see [section 23.2 page 50](#)), the returned value is irrelevant.

23.10 Properties of the ACANFD_STM32_Settings class

All properties of the ACANFD_STM32_Settings class are declared public and are initialized ([table 16](#)).

23.10.1 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

23.10 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 23.1 page 46
mArbitrationPhaseSegment1	uint16_t	256	See section 23.1 page 46
mArbitrationPhaseSegment2	uint8_t	128	See section 23.1 page 46
mArbitrationSJW	uint8_t	128	See section 23.1 page 46
mDataPhaseSegment1	uint8_t	32	See section 23.1 page 46
mDataPhaseSegment2	uint8_t	16	See section 23.1 page 46
mDataSJW	uint8_t	16	See section 23.1 page 46
mBitSettingOk	bool	true	See section 23.1 page 46
mModuleMode	ModuleMode	NORMAL_FD	See section 23.10.1 page 53
mDriverReceiveFIFO0Size	uint16_t	10	See section 17.1 page 33
mHardwareRxFIFO0Size	uint8_t	64	See section 14 page 28
mHardwareRxFIFO0Payload	Payload	PAYLOAD_64_BYTES	See section 14 page 28
mDriverReceiveFIFO1Size	uint16_t	0	See section 17.1 page 33
mHardwareRxFIFO1Size	uint8_t	0	See section 14 page 28
mHardwareRxFIFO1Payload	Payload	PAYLOAD_64_BYTES	See section 14 page 28
mEnableRetransmission	bool	true	See section 23.10.2 page 54
mDiscardReceivedStandardRemoteFrames	bool	false	See section 18 page 34
mDiscardReceivedExtendedRemoteFrames	bool	false	See section 18 page 34
mNonMatchingStandardFrameReception	FilterAction	FIF00	See section 18 page 34
mNonMatchingExtendedFrameReception	FilterAction	FIF00	See section 18 page 34
mTransceiverDelayCompensation	uint8_t	5	See section 23.10.3 page 54
mDriverTransmitFIFOSize	uint8_t	20	See section 9 page 24
mHardwareTransmitTxFIFOSize	uint8_t	24	See section 9 page 24
mHardwareDedicatedTxBufferCount	uint8_t	8	See section 10 page 26
mHardwareTransmitBufferPayload	Payload	PAYLOAD_64_BYTES	See section 13 page 26
mNonMatchingStandardMessageCallBack	ACANFDCallBackRoutine	nullptr	See section 19.1 page 42
mNonMatchingExtendedMessageCallBack	ACANFDCallBackRoutine	nullptr	See section 19.2 page 42

Table 16 – Properties of the ACANFD_STM32_Settings class

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.

23.10.2 The mEnableRetransmission property

By default, a frame is automatically retransmitted if 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.

23.10.3 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.

24 Other ACANFD_STM32 methods

24.1 The `getStatus` method

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

24.1.1 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.

24.1.2 The `rxErrorCount` method

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

This method returns the *Receive Error Counter* value.

24.1.3 The `isBusOff` method

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

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

24.1.4 The `transceiverDelayCompensationOffset` method

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

This method returns *Transceiver Delay Compensation Offset* value.

24.1.5 The `hardwareTxBufferPayload` method

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

This method returns the payload of transmit TxBuffers.

24.1.6 The hardwareRxFIFO0Payload method

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

This method returns the payload of hardware receive FIFO 0.

24.1.7 The hardwareRxFIFO1Payload method

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

This method returns the payload of hardware receive FIFO 1.

25 Porting guide

This section shows how to port the library to other STM32 microcontrollers.

You need the *reference manual* and the *datasheet* of the microcontroller, and the *datasheet* of the board.

Check the microcontroller contains CANFD modules, not CAN modules. CAN modules are completely different and the library cannot support them.

There are two CANFD module variants:

- those with fixed size message RAM (for example: NUCLEO_G431KB, NUCLEO_G474RE), see [section 25.1 page 56](#);
- those with programmable size message RAM (for example: NUCLEO_H723ZG, NUCLEO_H743ZI2), see [section 25.2 page 66](#).

25.1 Microcontroller with fixed size message RAM CANFD modules

Here, we explain how the library has been ported to the NUCLEO-G0B1RE, a Nucleo-64 development board with STM32G0B1RE microcontroller:

- the last release of the microcontroller *datasheet* is the document DS13560 Rev 4, december 2022;
- the last release of the microcontroller *reference manual* is the document RM0444 Rev 5, december 2020;
- the last release of the corresponding STM32 Nucleo-64 board user manual is UM2324 Rev 4, march 2021.

25.1 Microcontroller with fixed size message RAM CANFD modules

Take a look to the chapter 36 “FD controller area network (FDCAN)” (page 1196) of the RM0444 Rev 5 reference manual: the sentence “A 0.8-Kbyte message RAM per FDCAN instance implements filters, receive FIFOs, transmit event FIFOs and transmit FIFOs.” in the introduction means the FDCAN module has fixed size message RAM.

You can find the same sentence in chapter 44 of STM32G4 reference manual (RM0440 Rev 7, page 1938).

So we consider the source files relative to NUCLEO_G474RE as starting point. Porting to this microcontroller consists of the following steps:

1. create a new `ACANFD_STM32_NUCLEO_G0B1RE-objects.h` file ([section 25.1.1 page 57](#));
2. create a new `ACANFD_STM32_NUCLEO_G0B1RE-settings.h` file ([section 25.1.2 page 60](#));
3. modify the `ACANFD_STM32_from_cpp.h` file ([section 25.1.3 page 61](#));
4. modify the `ACANFD_STM32.h` file ([section 25.1.4 page 61](#));
5. modify the `ACANFD_STM32_Settings.h` file ([section 25.1.5 page 62](#));
6. write the `G0B1RE-LoopBackDemo` sample sketch ([section 25.1.6 page 63](#));
7. check bit rate ([section 25.1.7 page 64](#));
8. add interrupt handling ([section 25.1.8 page 64](#));
9. check pin assignments ([section 25.1.9 page 66](#)).

25.1.1 The new `ACANFD_STM32_NUCLEO_G0B1RE-objects.h` file

Duplicate the `ACANFD_STM32_NUCLEO_G474RE-objects.h` file and rename it `ACANFD_STM32_NUCLEO_G0B1RE-objects.h`.

This file should contain the Tx and Tx pin assignments for every CANFD module, its base address, its message RAM base address, its interrupt configuration.

Open the DS13560 Rev 4 *datasheet* and find all Tx and Rx pins. The first page specifies there are two FDCAN controllers, so you can delete FDCAN3 settings in the `ACANFD_STM32_NUCLEO_G0B1RE-objects.h` file.

Then search for `FDCAN1_RX`, `FDCAN1_TX`, `FDCAN2_RX`, `FDCAN2_TX` strings in the *datasheet*.

It is important to get pin and alternate function mapping. For example, `FDCAN1_RX` is found in table 13 page 56 (PA11, AF3), table 15 page 58 (PB8, AF3), table 17 page 60 (PC4, AF3) and table 18 page 61 (PD0, AF3).

Check theses pins are actually available on your NUCLEO board. In the STM32 Nucleo-64 board user manual (UM2324 Rev 4), table 13 from page 37 lists available pins: PD14 and PD15 are not available for `FDCAN2_RX` and `FDCAN2_TX`.

You can insert the result as comment in the `ACANFD_STM32_NUCLEO_G0B1RE-objects.h` file:

25.1 Microcontroller with fixed size message RAM CANFD modules

```
// NUCLEO-G0B1RE: STM32G0B1RE (DS13560 Rev 4, december 2022)
//   FDCAN1_RX : PA11, PB8, PC4, PD0
//   FDCAN1_TX : PA12, PB9, PC5, PD1
//   FDCAN2_RX : PB0, PB5, PB12, PC2; not available on Nucleo-64 board: PD14
//   FDCAN2_TX : PB1, PB6, PB13, PC3; not available on Nucleo-64 board: PD15
```

Now, write the `fdcan1_tx_pin_array` that handles all available pins for FDCAN1_TX:

- **use the underscore in pin naming**, for example `PA_12`; there are subtle differences between `PA_12` and `PA12` that make them incompatible;
- after each pin, write the alternate function (for this microcontroller, it is always 3);
- the first item defines the default pin.

So the `fdcan1_tx_pin_array` is:

```
const std::initializer_list <ACANFD_STM32::PinPort> fdcan1_tx_pin_array {
    ACANFD_STM32::PinPort (PA_12, 3),
    ACANFD_STM32::PinPort (PB_9,  3),
    ACANFD_STM32::PinPort (PC_5,  3),
    ACANFD_STM32::PinPort (PD_1,  3)
} ;
```

And for the `fdcan1_rx_pin_array` is:

```
const std::initializer_list <ACANFD_STM32::PinPort> fdcan1_rx_pin_array {
    ACANFD_STM32::PinPort (PA_11, 3),
    ACANFD_STM32::PinPort (PB_8,  3),
    ACANFD_STM32::PinPort (PC_4,  3),
    ACANFD_STM32::PinPort (PD_0,  3)
} ;
```

Now define the `fdcan1` object that is the instance of the FDCAN1 driver. It has five arguments:

1. `FDCAN1`, the base address of the FDCAN1 control registers; this symbol is defined by ST system files;
2. `SRAMCAN_BASE`, the base address of the FDCAN1 message RAM; this symbol is defined by ST system files;
3. `std::nullopt`, indicating that CANFD interrupts are not used;
4. `fdcan1_tx_pin_array`, the Tx pins and their alternate function settings, defined above;
5. `fdcan1_rx_pin_array`, the Rx pins and their alternate function settings, defined above.

```
ACANFD_STM32 fdcan1 (
    FDCAN1, // CAN Peripheral base address
```

25.1 Microcontroller with fixed size message RAM CANFD modules

```
SRAMCAN_BASE,  
std::nullopt, // No interrupt  
fdcan1_tx_pin_array,  
fdcan1_rx_pin_array  
);
```

Handling interrupts can be tricky, so it's a good idea to start without the CANFD module's interrupt handling. So remove `FDCAN1_IT0_IRQHandler`, `FDCAN1_IT1_IRQHandler`, `FDCAN2_IT0_IRQHandler` and `FDCAN2_IT1_IRQHandler` interrupt routines declaration and implementation. Handling interrupts will be discussed in [section 25.1.8 page 64](#).

Follow the same procedure to provide information for `FDCAN2_TX_PIN_ARRAY`, `FDCAN2_RX_PIN_ARRAY` and `fdcan2`.

Note the message RAM address is different for every module:

- for `fdcan1`, it is `SRAMCAN_BASE + 212 * 4 * 0` (that is `SRAMCAN_BASE`);
- for `fdcan2`, it is `SRAMCAN_BASE + 212 * 4 * 1`;
- if the microcontroller has 3 FDCAN modules, for `fdcan3`, `SRAMCAN_BASE + 212 * 4 * 2`.

That's it for the `ACANFD_STM32_NUCLEO_G0B1RE-objects.h` file. Here is the full listing (comments have been removed):

```
#pragma once  
  
#include <ACANFD_STM32_from_cpp.h>  
  
const std::initializer_list <ACANFD_STM32::PinPort> fdcan1_tx_pin_array {  
    ACANFD_STM32::PinPort (PA_12, 3),  
    ACANFD_STM32::PinPort (PB_9, 3),  
    ACANFD_STM32::PinPort (PC_5, 3),  
    ACANFD_STM32::PinPort (PD_1, 3)  
};  
  
const std::initializer_list <ACANFD_STM32::PinPort> fdcan1_rx_pin_array {  
    ACANFD_STM32::PinPort (PA_11, 3),  
    ACANFD_STM32::PinPort (PB_8, 3),  
    ACANFD_STM32::PinPort (PC_4, 3),  
    ACANFD_STM32::PinPort (PD_0, 3)  
};  
  
ACANFD_STM32 fdcan1 (  
    FDCAN1, // CAN Peripheral base address  
    SRAMCAN_BASE,  
    std::nullopt, // No interrupt  
    fdcan1_tx_pin_array,  
    fdcan1_rx_pin_array
```

25.1 Microcontroller with fixed size message RAM CANFD modules

```
) ;

const std::initializer_list <ACANFD_STM32::PinPort> FDCAN2_TX_PIN_ARRAY {
    ACANFD_STM32::PinPort (PB_1,  3),
    ACANFD_STM32::PinPort (PB_6,  3),
    ACANFD_STM32::PinPort (PB_13, 3),
    ACANFD_STM32::PinPort (PC_3,  3)
} ;

const std::initializer_list <ACANFD_STM32::PinPort> FDCAN2_RX_PIN_ARRAY {
    ACANFD_STM32::PinPort (PB_0,  3),
    ACANFD_STM32::PinPort (PB_5,  3),
    ACANFD_STM32::PinPort (PB_12, 3),
    ACANFD_STM32::PinPort (PC_2,  3)
} ;

ACANFD_STM32 fdcan2 (
    FDCAN2, // CAN Peripheral base address
    SRAMCAN_BASE + 212 * 4,
    std::nullopt, // No interrupt
    FDCAN2_TX_PIN_ARRAY,
    FDCAN2_RX_PIN_ARRAY
) ;
```

25.1.2 The new ACANFD_STM32_NUCLEO_G0B1RE-settings.h file

Duplicate the ACANFD_STM32_NUCLEO_G474RE-settings.h and rename it ACANFD_STM32_NUCLEO_G0B1RE-settings.h.

This file implements the fdcanClock function that relies on functions defined in a microcontroller specific file, here stm32g0xx_ll_rcc.h for the STM32G0B1RE microcontroller. The only change to do is to replace the inclusion of stm32g4xx_ll_rcc.h (specific to STM32G4xxx) by the inclusion of stm32g0xx_ll_rcc.h. Here is the full listing (comments have been removed):

```
#pragma once

#include <stm32g0xx_ll_rcc.h>

inline uint32_t fdcanClock (void) {
    if (!__HAL_RCC_FDCAN_IS_CLK_ENABLED ()) { // Is not enabled ?
        //--- Enable CAN clock
        __HAL_RCC_FDCAN_CLK_ENABLE () ;
        //--- Reset CAN peripherals
        __HAL_RCC_FDCAN_FORCE_RESET () ;
        __HAL_RCC_FDCAN_RELEASE_RESET () ;
        //--- Select CAN clock
        LL_RCC_SetFDCANClockSource (RCC_FDCANCLKSOURCE_PCLK1) ;
    }
```

25.1 Microcontroller with fixed size message RAM CANFD modules

```
}  
    return HAL_RCC_GetPCLK1Freq ();  
}
```

The `fdcanClock` function is called by the `ACANFD_STM32_Settings` constructor.

By default, after startup, the FDCAN clock (common to the two modules) is disabled. The first call of the function enables FDCAN clock, performs a reset of all FDCAN peripherals, and selects PCLK1 as FDCAN clock. Then, the PCLK1 frequency (in Hz) is returned.

The following calls do not perform any action on the clock, they just return the PCLK1 frequency.

25.1.3 Modify the `ACANFD_STM32_from_cpp.h` file

This file redirects to the header for the module whose RAM message is of fixed or variable size.

The STM32G0B1RE microcontroller has fixed size FDCAN message RAM, as the STM32G474RE, so we insert:

```
#elif defined (ARDUINO_NUCLEO_G0B1RE)  
    #include "ACANFD-STM32-fixed-ram-sections.h"
```

Here is the full listing (comments have been removed):

```
#pragma once  
  
#ifdef ARDUINO_NUCLEO_H743ZI2  
    #include "ACANFD-STM32-programmable-ram-sections.h"  
#elif defined (ARDUINO_NUCLEO_G431KB)  
    #include "ACANFD-STM32-fixed-ram-sections.h"  
#elif defined (ARDUINO_NUCLEO_G474RE)  
    #include "ACANFD-STM32-fixed-ram-sections.h"  
#elif defined (ARDUINO_NUCLEO_G0B1RE)  
    #include "ACANFD-STM32-fixed-ram-sections.h"  
#elif defined (ARDUINO_NUCLEO_H723ZG)  
    #include "ACANFD-STM32-programmable-ram-sections.h"  
#else  
    #error "Unhandled_Board"  
#endif
```

25.1.4 Modify the `ACANFD_STM32.h` file

This file redirects to the corresponding `xxx-objects.h` header file. Insert:

```
#elif defined (ARDUINO_NUCLEO_G0B1RE)  
    #include "ACANFD_STM32_NUCLEO_G0B1RE-objects.h"
```

25.1 Microcontroller with fixed size message RAM CANFD modules

Here is the full listing (comments have been removed):

```
#pragma once

#ifdef ARDUINO_NUCLEO_H743ZI2
    #include "ACANFD_STM32_NUCLEO_H743ZI2-objects.h"
#elif defined (ARDUINO_NUCLEO_G431KB)
    #include "ACANFD_STM32_NUCLEO_G431KB-objects.h"
#elif defined (ARDUINO_NUCLEO_G474RE)
    #include "ACANFD_STM32_NUCLEO_G474RE-objects.h"
#elif defined (ARDUINO_NUCLEO_G0B1RE)
    #include "ACANFD_STM32_NUCLEO_G0B1RE-objects.h"
#elif defined (ARDUINO_NUCLEO_H723ZG)
    #include "ACANFD_STM32_NUCLEO_H723ZG-objects.h"
#else
    #error "Unhandled_Nucleo_Board"
#endif
```

25.1.5 Modify the ACANFD_STM32_Settings.h file

At the beginning of this file, the corresponding *xxx*-settings.h header file should be included. Insert:

```
#elif defined (ARDUINO_NUCLEO_G0B1RE)
    #include "ACANFD_STM32_NUCLEO_G0B1RE-settings.h"
```

Here is the beginning of the listing (comments have been removed):

```
#pragma once

#include <Arduino.h>

#ifdef ARDUINO_NUCLEO_H743ZI2
    #include "ACANFD_STM32_NUCLEO_H743ZI2-settings.h"
#elif defined (ARDUINO_NUCLEO_G431KB)
    #include "ACANFD_STM32_NUCLEO_G431KB-settings.h"
#elif defined (ARDUINO_NUCLEO_G474RE)
    #include "ACANFD_STM32_NUCLEO_G474RE-settings.h"
#elif defined (ARDUINO_NUCLEO_G0B1RE)
    #include "ACANFD_STM32_NUCLEO_G0B1RE-settings.h"
#elif defined (ARDUINO_NUCLEO_H723ZG)
    #include "ACANFD_STM32_NUCLEO_H723ZG-settings.h"
#else
    #error "Unhandled_Nucleo_Board"
#endif

...
```

25.1.6 The GOB1RE-LoopBackDemo sample sketch

Now, it is time to write a first demo sketch. The FDCAN modules are configured in *external loopback mode* ([section 23.10.1 page 53](#)), so no external hardware is required. In `sample-code` directory, duplicate the `G474RE-LoopBackDemo` sketch and rename it `GOB1RE-LoopBackDemo` (rename the directory and the `.ino` file).

Perform the following changes.

1. Update the comments at the beginning of the file.
2. Change board checking. Instead of:

```
#ifndef ARDUINO_NUCLEO_G474RE
#error This sketch runs on NUCLEO-G474RE Nucleo-64 board
#endif
```

Write:

```
#ifndef ARDUINO_NUCLEO_GOB1RE
#error This sketch runs on NUCLEO-GOB1RE Nucleo-64 board
#endif
```

3. STM32G474RE has 3 FDCAN modules, STM32GOB1RE has 2. So remove all `fdcan3` relative lines in the `setup` and `loop` functions. Remove also `gSentCount3` and `gReceivedCount3` global variables.
4. Remember that no FDCAN interrupt is handled. So it is required to add a call of the `poll` method ([section 15 page 30](#)) at the beginning of the `loop` function:

```
void loop () {
  //--- As the library does not implement FDCAN interrupt handling for
  // the STM32GOB1RE, you should poll each module.
  fdcan1.poll () ;
  fdcan2.poll () ;
  //---
  if (gSendDate < millis ()) {
    ...
  }
}
```

5. Now, you can compile and and run the demo sketch. You can observe in the monitor:

```
CPU frequency: 64000000 Hz
PCLK1 frequency: 64000000 Hz
HCLK frequency: 64000000 Hz
SysClock frequency: 64000000 Hz
FDCAN Clock: 64000000 Hz
Bit Rate prescaler: 2
Arbitration Phase segment 1: 47
Arbitration Phase segment 2: 16
Arbitration SJW: 16
Actual Arbitration Bit Rate: 500000 bit/s
```

25.1 Microcontroller with fixed size message RAM CANFD modules

```
Arbitration sample point: 75.00%
Exact Arbitration Bit Rate ? yes
Data Phase segment 1: 23
Data Phase segment 2: 8
Data SJW: 8
Actual Data Bit Rate: 1000000 bit/s
Data sample point: 75.00%
Exact Data Bit Rate ? yes
fdcan1 configuration ok
fdcan2 configuration ok
fdcan1 sent: 1
fdcan2 sent: 1
fdcan1 received: 1
fdcan2 received: 1
fdcan1 sent: 2
fdcan2 sent: 2
fdcan1 received: 2
fdcan2 received: 2
...
```

If you observe successive sent / received messages, it is ok. Note the FDCAN clock frequency, here 64 MHz.

25.1.7 Checking bit rate

It is important to check the bit rates are correct. As the FDCAN modules are configured in *external loopback mode*, the CAN FD frames can be observed on their default Tx pins, PA12 for fdcan1 and PB1 for fdcan2.

The easy way is to use a logic analyzer with a CANFD protocol decoder. Here, we observe valid frames and correct bit rates on PA12 and PB1 pins.

If your logic analyzer only supports CAN 2.0B protocol analyzer, modify the demo sketch for sending CAN 2.0B frames.

25.1.8 Adding interrupt handling

Using FDCAN interrupts makes it unnecessary to call the `poll` method, all transfers between the driver and the FDCAN module are carried out by the interrupt service routines.

But, depending on the microcontroller, this can be very easy or impossible without modifying the system files supplied by ST.

A FDCAN has two interrupt lines, IT0 and IT1. Ideally, these two lines correspond to dedicated microcontroller interrupts.

STM32G474RE. Consider the STM32G474RE; it has 3 FDCAN modules, and six dedicated

25.1 Microcontroller with fixed size message RAM CANFD modules

interrupts¹¹: FDCAN1_IT0 (#21), FDCAN1_IT1 (#22), FDCAN2_IT0 (#86), FDCAN2_IT1 (#87), FDCAN3_IT0 (#88), FDCAN3_IT1 (#89). This makes very easy to assign interrupt to `fdcanx` objects. Open the `ACANFD_STM32_NUCLEO_G474RE-objects.h` file and observe how `fdcan1` is defined:

```
void loop () {
  ACANFD_STM32 fdcan1 (
    FDCAN1, // CAN Peripheral base address
    SRAMCAN_BASE,
    ACANFD_STM32::IRQs (FDCAN1_IT0_IRQn, FDCAN1_IT1_IRQn), // Interrupts
    fdcan1_tx_pin_array,
    fdcan1_rx_pin_array
  ) ;

  extern "C" void FDCAN1_IT0_IRQHandler (void) ;
  extern "C" void FDCAN1_IT1_IRQHandler (void) ;

  void FDCAN1_IT0_IRQHandler (void) {
    fdcan1.isr0 () ;
  }

  void FDCAN1_IT1_IRQHandler (void) {
    fdcan1.isr1 () ;
  }
}
```

Note the following points:

- for `fdcan1` object, interrupt lines are defined by `ACANFD_STM32::IRQs (FDCAN1_IT0_IRQn, FDCAN1_IT1_IRQn)`; no need to use the actual interrupt numbers, ST provides the `FDCAN1_IT0_IRQn` and `FDCAN1_IT1_IRQn` names;
- for interrupt services routines, the `FDCAN1_IT0_IRQHandler` and `FDCAN1_IT1_IRQHandler` names are **required**, they are defined as weak symbols by system files;
- they **should** be declared with C linkage by `extern "C"`.

The `fdcan2` and `fdcan3` are defined in a similar way.

STM32G0B1RE. Unfortunately, things aren't quite as simple with the STM32G0B1RE. It has two FDCAN modules, we expect 4 dedicated interrupts. But in RM0444 Rev 5, table 58, pages 314-316, we find only two interrupt lines:

- #21, shared between TIM16, FDCAN1_IT0 and FDCAN2_IT0;
- #22, shared between TIM17, FDCAN1_IT1 and FDCAN2_IT1.

¹¹RM0440 Rev 7, table 97, pages 441-444.

25.1.9 Checking pin assignments

25.2 Microcontroller with programmable size message RAM CANFD modules