

# ACANFD\_FeatherM4CAN Arduino library, for Adafruit Feather M4 CAN Version 2.0.0

Pierre Molinaro

May 24, 2024

## Contents

<b>1</b>	<b>Versions</b>	<b>4</b>
<b>2</b>	<b>Features</b>	<b>4</b>
<b>3</b>	<b>CAN Interfaces</b>	<b>5</b>
3.1	CAN0 . . . . .	5
3.2	CAN1 . . . . .	6
<b>4</b>	<b>FDCAN clock selection</b>	<b>6</b>
4.1	FDCAN0 clock selection . . . . .	7
4.2	FDCAN1 clock selection . . . . .	7
4.3	Selecting a clock . . . . .	8
4.4	Usefull FDCAN clocks frequencies . . . . .	8
4.5	Modifying DPLL1 frequency . . . . .	9
<b>5</b>	<b>Data flow</b>	<b>9</b>
<b>6</b>	<b>A sample sketch: LoopBackDemoCANFD_CAN1</b>	<b>11</b>
<b>7</b>	<b>The CANMessage class</b>	<b>13</b>
<b>8</b>	<b>The CANFDMessage class</b>	<b>14</b>
8.1	Properties . . . . .	15
8.2	The default constructor . . . . .	15
8.3	Constructor from CANMessage . . . . .	15
8.4	The type property . . . . .	16
8.5	The len property . . . . .	17

## CONTENTS

---

8.6	The <code>idx</code> property	17
8.7	The <code>pad</code> method	17
8.8	The <code>isValid</code> method	17
<b>9</b>	<b>Transmit FIFO</b>	<b>18</b>
9.1	The <code>driverTransmitFIFOSize</code> method	18
9.2	The <code>driverTransmitFIFOCount</code> method	18
9.3	The <code>driverTransmitFIFOPeakCount</code> method	18
<b>10</b>	<b>Transmit buffers (<code>TxBuffer<sub>i</sub></code>)</b>	<b>19</b>
<b>11</b>	<b>Transmit Priority</b>	<b>19</b>
<b>12</b>	<b>Receive FIFOs</b>	<b>19</b>
<b>13</b>	<b>Payload size</b>	<b>19</b>
13.1	The <code>ACANFD_FeatherM4CAN_Settings::wordCountForPayload</code> static method	20
13.2	The <code>ACANFD_FeatherM4CAN_Settings::frameDataByteCountForPayload</code> static method	20
13.3	Changing the default payloads	20
<b>14</b>	<b>Message RAM</b>	<b>21</b>
<b>15</b>	<b>Sending frames: the <code>tryToSendReturnStatusFD</code> method</b>	<b>22</b>
15.1	Testing a send buffer: the <code>sendBufferNotFullForIndex</code> method	23
15.2	Usage example	23
<b>16</b>	<b>Retrieving received messages using the <code>receiveFD<sub>i</sub></code> method</b>	<b>24</b>
16.1	Driver receive FIFO <i>i</i> size	26
16.2	The <code>driverReceiveFIFO<sub>i</sub>Size</code> method	26
16.3	The <code>driverReceiveFIFO<sub>i</sub>Count</code> method	26
16.4	The <code>driverReceiveFIFO<sub>i</sub>PeakCount</code> method	26
16.5	The <code>resetDriverReceiveFIFO<sub>i</sub>PeakCount</code> method	26
<b>17</b>	<b>Acceptance filters</b>	<b>27</b>
17.1	Acceptance filters for standard frames	27
17.1.1	Defining standard frame filters	27
17.1.2	Add single filter	27
17.1.3	Add dual filter	28
17.1.4	Add range filter	29
17.1.5	Add classic filter	29
17.2	Acceptance filters for extended frames	30
17.2.1	Defining extended frame filters	30
17.2.2	Add single filter	31
17.2.3	Add dual filter	32
17.2.4	Add range filter	32
17.2.5	Add classic filter	32

<b>18</b>	<b>The <code>dispatchReceivedMessage</code> method</b>	<b>33</b>
18.1	Dispatching non matching standard frames . . . . .	34
18.2	Dispatching non matching extended frames . . . . .	34
<b>19</b>	<b>The <code>dispatchReceivedMessageFIF00</code> method</b>	<b>35</b>
<b>20</b>	<b>The <code>dispatchReceivedMessageFIF01</code> method</b>	<b>35</b>
<b>21</b>	<b>The <code>ACANFD_FeatherM4CAN::beginFD</code> method reference</b>	<b>36</b>
21.1	The prototypes . . . . .	36
21.2	The error codes . . . . .	37
21.2.1	The <code>kTxBufferCountGreaterThan32</code> error code . . . . .	37
<b>22</b>	<b><code>ACANFD_FeatherM4CAN_Settings</code> class reference</b>	<b>37</b>
22.1	The <code>ACANFD_FeatherM4CAN_Settings</code> constructors: computation of the CAN bit settings	38
22.1.1	6 arguments constructor . . . . .	38
22.1.2	4-arguments constructor . . . . .	38
22.1.3	Exact bit rates . . . . .	39
22.2	The <code>CANBitSettingConsistency</code> method . . . . .	43
22.3	The <code>actualArbitrationBitRate</code> method . . . . .	44
22.4	The <code>exactArbitrationBitRate</code> method . . . . .	45
22.5	The <code>exactDataBitRate</code> method . . . . .	45
22.6	The <code>ppmFromDesiredArbitrationBitRate</code> method . . . . .	45
22.7	The <code>ppmFromDesiredDataBitRate</code> method . . . . .	45
22.8	The <code>arbitrationSamplePointFromBitStart</code> method . . . . .	45
22.9	The <code>dataSamplePointFromBitStart</code> method . . . . .	46
22.10	Properties of the <code>ACANFD_FeatherM4CAN_Settings</code> class . . . . .	46
22.10.1	The <code>mModuleMode</code> property . . . . .	47
22.10.2	The <code>mEnableRetransmission</code> property . . . . .	47
22.10.3	The <code>mTransceiverDelayCompensation</code> property . . . . .	47
<b>23</b>	<b>Other <code>ACANFD_FeatherM4CAN</code> methods</b>	<b>48</b>
23.1	The <code>getStatus</code> method . . . . .	48
23.1.1	The <code>txErrorCount</code> method . . . . .	48
23.1.2	The <code>rxErrorCount</code> method . . . . .	48
23.1.3	The <code>isBusOff</code> method . . . . .	48
23.1.4	The <code>transceiverDelayCompensationOffset</code> method . . . . .	48
23.1.5	The <code>hardwareTxBufferPayload</code> method . . . . .	48
23.1.6	The <code>hardwareRxFIF00Payload</code> method . . . . .	48
23.1.7	The <code>hardwareRxFIF01Payload</code> method . . . . .	49

---

## 1 Versions

Version	Date	Comment
2.0.0	May 24, 2024	Added CANFD clock selection (see <a href="#">section 4 page 6</a> ). Removed the <code>mTripleSampling</code> property of <code>ACANFD_FeatherM4CAN_Settings</code> class, it is useless, the CANFD modules do not use it.
1.2.3	March 7, 2024	Ensuring message RAM is in the first 64kB RAM.
1.2.2	April 6, 2022	Added <code>BUS_MONITORING</code> mode ( <a href="#">section 22.10.1 page 47</a> ). Added 5-arguments <code>ACANFD_FeatherM4CAN_Settings</code> constructor ( <a href="#">section 22.1.1 page 38</a> ).
1.2.1	March 17, 2022	Added <a href="#">section 11 page 19</a> on transmit priority. Added <code>LoopBackDemoCANFD_CAN1_payload</code> sample sketch.
1.2.0	March 12, 2022	Added <code>dispatchReceivedMessage</code> method. Added <code>dispatchReceivedMessageFIFO0</code> method. Added <code>dispatchReceivedMessageFIFO1</code> method. Added <code>LoopBackDemoCANFD_CAN1_dispatch</code> sample sketch.
1.1.0	March 10, 2022	Added handling Rx FIFO 1. Added receive standard filters. Added receive extended filters. Added <code>LoopBackDemoCANFD_CAN1_StandardFilters</code> sample sketch. Added <code>LoopBackDemoCANFD_CAN1_ExtendedFilters</code> sample sketch.
1.0.1	March 9, 2022	Added constraint <code>settings.mHardwareTransmitTxFIFOSize ≥ 2</code> . Added constraint <code>settings.mHardwareDedicatedTxBufferCount ≤ 30</code> . Fixed <code>tryToSendReturnStatusFD</code> , this method was returning error 1 in release 1.0.0.
1.0.0	March 8, 2022	Initial release (buggy, do not use).

## 2 Features

The `ACANFD_FeatherM4CAN` library is a CANFD (*Controller Area Network with Flexible Data*) Controller driver for the *Adafruit Feather M4 CAN*<sup>1</sup> board running Arduino. 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 the `CAN0` and `CAN1` CANFD modules;
- default configuration sends and receives any frame – no default filter to provide;
- efficient built-in CAN bit settings computation from arbitration and data bit rates;

---

<sup>1</sup><https://www.adafruit.com/product/4759>

- user can fully define its own CAN bit setting values;
- up to 128 standard reception filters can be easily defined;
- up to 128 extended reception filters can be easily defined;
- reception filters accept callback functions;
- driver and controller transmit buffer sizes are customisable;
- driver and controller receive buffer size is customisable;
- overflow of the driver receive buffer is detectable;
- the message RAM allocation is customizable and the driver checks no overflow occurs;
- *internal loop back, external loop back* controller modes are selectable.

### 3 CAN Interfaces

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

#### 3.1 CAN0

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

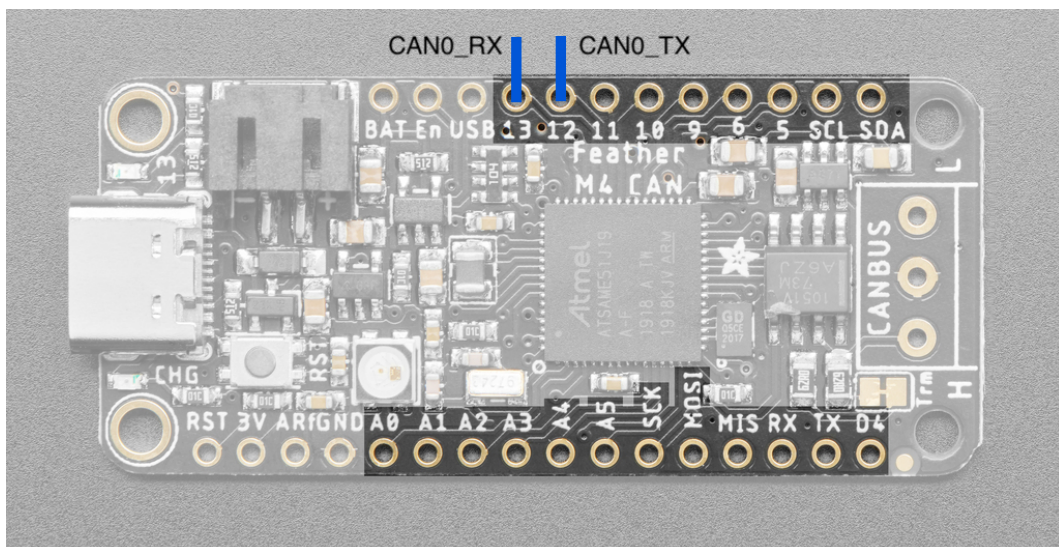


Figure 1 – CAN0 pins

### 3.2 CAN1

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

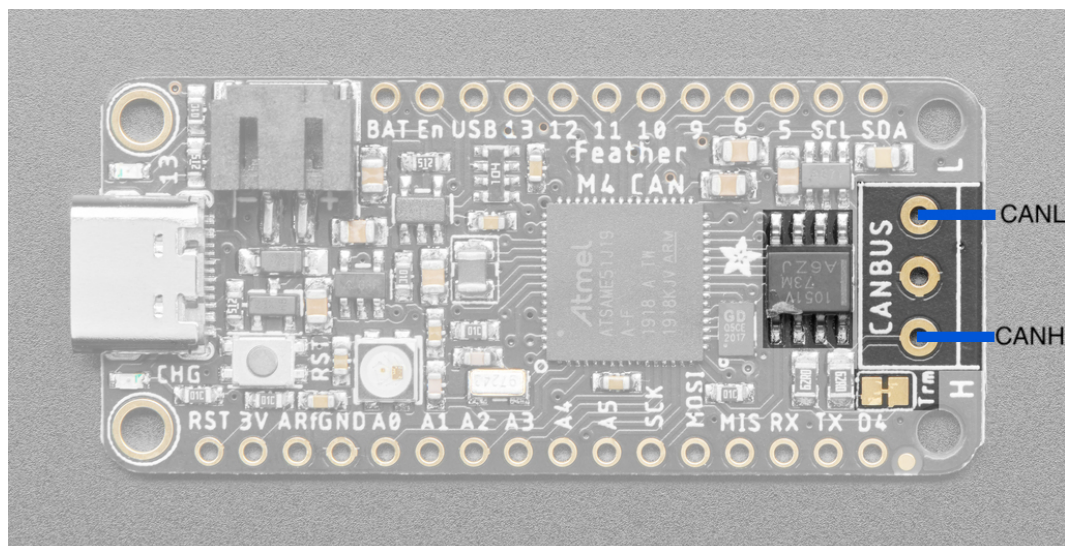


Figure 2 – CAN1 pins

## 4 FDCAN clock selection

The reference document is *SAM D5x/E5x Family Data Sheet*<sup>3</sup>.

Every FDCAN module should be driven by a dedicated clock, the *FDCAN clock*. A FDCAN clock setting is independent from the settings of the other CANFD module. The maximum *FDCAN clock* frequency is approximately 100 MHz<sup>4</sup>. At 125°C, the limit is approximately 90 MHz<sup>5</sup>.

For driving the *FDCAN clock*, there are three interesting sources:

- a 48 MHz source; its frequency is independent from CPU speed setting; this source was the implicit source available in the 1.x library releases;
- a DPLL0 source; its frequency is exactly the CPU speed setting;
- a DPLL1 source; its frequency is set to 100 MHz, independently from CPU speed setting.

An integer divisor from 1 to 256 can be applied to DPLL0 and DPLL1 sources. No divisor is available for the 48 MHz source.

<sup>2</sup><https://learn.adafruit.com/adafruit-feather-m4-can-express/pinouts>

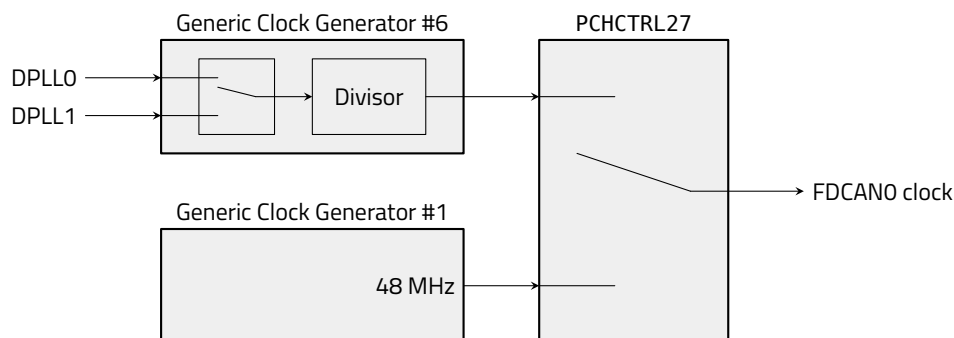
<sup>3</sup>Microchip Technology Inc, *SAM D5x/E5x Family Data Sheet*, DS60001507G, 2021

<sup>4</sup>DS60001507G, table 54-6, page 1789.

<sup>5</sup>DS60001507G, table 56-2, page 1851-1852.

## 4.1 FDCAN0 clock selection

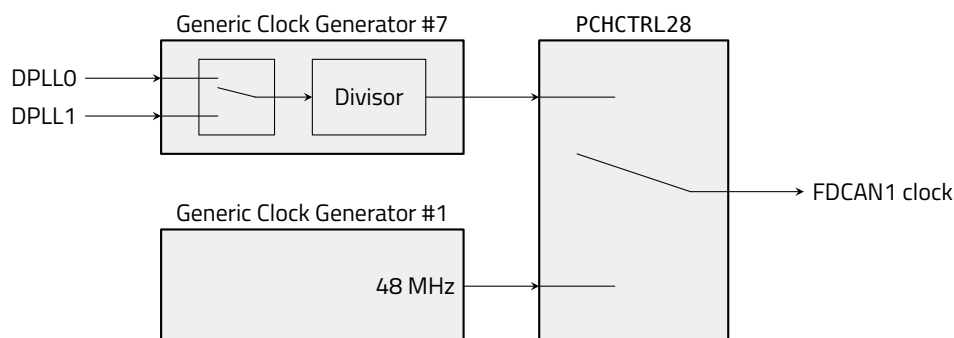
The FDCAN0 clock selection is summarised by the [figure 3](#). The PCHCTRL27 register (DS60001507G, section 14.8.4, page 155) drives the FDCAN0 clock. It is a multiplexer that selects one of 12 possible sources. These sources are provided by the *Generic Clock Generators*. 12 *Generic Clock Generators* are implemented, from #0 to #11 (DS60001507G, chapter 14, from page 142). #0 to #5 are already used by system, #6 to #11 are free and available for any use. The arduino sketch `generic-clcks.ino` in the `sample-code` directory displays the configuration of the 12 *Generic Clock Generators*. The Generic Clock Generator #1 output is the 48 MHz clock; as it used for some modules (as USB), it is advisable not to change its settings. **The library uses the first available Generic Clock Generator (#6) for an alternate clock generator for FDCAN0 clock.** It implements a source multiplexer (two sources are useful, DPLL0 and DPLL1), followed by a divisor.



**Figure 3** – FDCAN0 clock selection

## 4.2 FDCAN1 clock selection

The FDCAN1 clock selection is summarised by the [figure 4](#). It is similar to FDCAN0 clock selection. The PCHCTRL28 register drives the FDCAN1 clock. **The library uses the second available Generic Clock Generator (#7) for an alternate clock generator for FDCAN1 clock.**



**Figure 4** – FDCAN1 clock selection

### 4.3 Selecting a clock

The `ACANFD_FeatherM4CAN_Settings::ClockSelection` enumerated type defines the 11 available clock sources (table 2). The clock selection is the first argument of the `ACANFD_FeatherM4CAN_Settings` constructors.

	Source	Frequency
<code>ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz</code>		48 MHz
<code>ACANFD_FeatherM4CAN_Settings::CPU_CLOCK</code>		CPU clock frequency
<code>ACANFD_FeatherM4CAN_Settings::CPU_CLOCK_DIVIDED_BY_2</code>		half of CPU clock frequency
<code>ACANFD_FeatherM4CAN_Settings::CPU_CLOCK_DIVIDED_BY_3</code>		third of CPU clock frequency
<code>ACANFD_FeatherM4CAN_Settings::CPU_CLOCK_DIVIDED_BY_4</code>		quarter of CPU clock frequency
<code>ACANFD_FeatherM4CAN_Settings::CPU_CLOCK_DIVIDED_BY_5</code>		fifth of CPU clock frequency
<code>ACANFD_FeatherM4CAN_Settings::CPU_CLOCK_DIVIDED_BY_6</code>		sixth of CPU clock frequency
<code>ACANFD_FeatherM4CAN_Settings::DPLL1_CLOCK</code>		100 MHz
<code>ACANFD_FeatherM4CAN_Settings::DPLL1_CLOCK_DIVIDED_BY_2</code>		50 MHz
<code>ACANFD_FeatherM4CAN_Settings::DPLL1_CLOCK_DIVIDED_BY_3</code>		33.33 MHz
<code>ACANFD_FeatherM4CAN_Settings::DPLL1_CLOCK_DIVIDED_BY_4</code>		25 MHz

**Table 2** – Available FDCAN clock sources

**Important note.** When selecting a source from the CPU clock, not all combinations are valid depending on the actual speed of the CPU. The FDCAN clock max frequency is approximately 100 MHz, as indicated above. The table 3 gives the available settings for the FDCAN clock of the FDCAN modules. Values above this limit are noted in red in this table, values near this limit in orange.

CPU Speed	80 MHz	120 MHz	150 MHz	160 MHz	180 MHz	200 MHz
		(standard)	(overclock)	(overclock)	(overclock)	(overclock)
<b>CPU_CLOCK</b>	80 MHz	120 MHz	150 MHz	160 MHz	180 MHz	200 MHz
<b>CPU_CLOCK_DIVIDED_BY_2</b>	40 MHz	60 MHz	75 MHz	80 MHz	90 MHz	100 MHz
<b>CPU_CLOCK_DIVIDED_BY_3</b>	26.67 MHz	40 MHz	50 MHz	53.33 MHz	60 MHz	66.67 MHz
<b>CPU_CLOCK_DIVIDED_BY_4</b>	20 MHz	30 MHz	37.5 MHz	40 MHz	45 MHz	50 MHz
<b>CPU_CLOCK_DIVIDED_BY_5</b>	16 MHz	24 MHz	30 MHz	32 MHz	36 MHz	40 MHz
<b>CPU_CLOCK_DIVIDED_BY_6</b>	13.33 MHz	20 MHz	25 MHz	26.67 MHz	30 MHz	33.33 MHz

**Table 3** – FDCAN clock frequencies, from CPU speed and CAN CLOCK selection

### 4.4 Usefull FDCAN clocks frequencies

The CiA 601-2 CAN controller interface specification recommends using 20 MHz, 40 MHz, or 80 MHz<sup>6</sup>.

**80 MHz.** Currently only available overclocking CPU clock to 160 MHz and selecting `CPU_CLOCK_DIVIDED_BY_2`, or slown down CPU to 80 MHz and select `CPU_CLOCK`.

**40 MHz.** Standard 120 MHz CPU clock and `CPU_CLOCK_DIVIDED_BY_3`.

<sup>6</sup>CAN Newsletter 1/2018, Recommendation for the CAN FD bit-timing, <https://can-newsletter.org/uploads/media/raw/5a08588dc5eef8bbdbf7113ab5537251.pdf>



**20 MHz.** Standard 120 MHz CPU clock and CPU\_CLOCK\_DIVIDED\_BY\_6.

As we can see, the problem that persists is the impossibility of obtaining 80 MHz with a standard CPU frequency of 120 MHz. The 100 MHz output of DPLL1 is useless for this purpose.

### 4.5 Modifying DPLL1 frequency

It is possible to set the DPLL1 output to 80 MHz.

**Caution! Making this modification may alter some peripheral as USART, I2C, SPI, QSPI...** I think it does not alter USB as it uses the 48 MHz clock.

The trick is to consider the start up code in file:

Arduino15/packages/adafruit/hardware/samd/1.7.16/cores/arduino/startup.c

Observe line 178:

```
OSCCTRL->DPLL[1].DPLLATIO.reg = OSCCTRL_DPLLATIO_LDRFRAC(0x00) | OSCCTRL_DPLLATIO_LDR(99); //100 Mhz
```

The "99" at end of the line means the DPLL1 output frequency is (99 + 1) MHz. So changing this value to 79 provides a 80 MHz DPLL1 output frequency:

```
OSCCTRL->DPLL[1].DPLLATIO.reg = OSCCTRL_DPLLATIO_LDRFRAC(0x00) | OSCCTRL_DPLLATIO_LDR(79); //80 Mhz
```

You can check the DPLL1 output frequency is actually modified running the `oscillators-controller.ino` sketch. Therefore, the `ACANFD_FeatherM4CAN_Settings::DPLL1_CLOCK` setting provides a 80 MHz FD-CAN clock, so the [table 2](#) should be adapted accordingly.

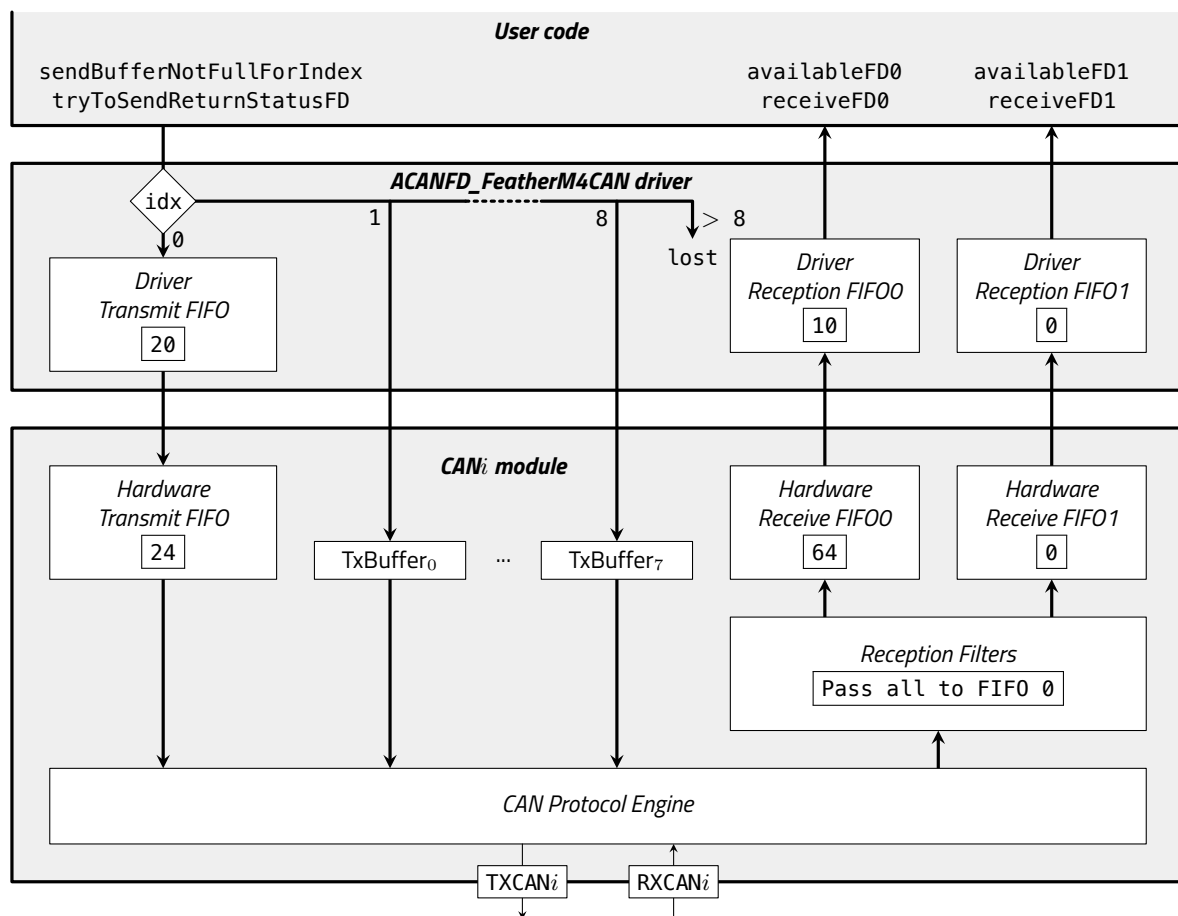
## 5 Data flow

The [figure 5](#) illustrates default message flow of sending and receiving CANFD messages for CAN0 and CAN1 modules.

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

A message is defined by an instance of the `CANFDMessage` or `CANMessage` class. For sending a message, user code calls the `tryToSendReturnStatusFD` method – see [section 15 page 22](#) for details, and the `idx` property of the sent message should be:

- 0 (default value), for sending via *driver transmit FIFO* and *hardware transmit FIFO*;
- 1, for sending via *TxBuffer<sub>0</sub>*;
- ...
- 8, for sending via *TxBuffer<sub>7</sub>*.



**Figure 5** – Message flow in ACANFD\_FeatherM4CAN driver and CAN $i$  module, default configuration

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

You can call the `sendBufferNotFullForIndex` method ([section 15.1 page 23](#)) 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 17 page 27](#) 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 16.1 page 26](#) 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 16 page 24](#);
- 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 16 page 24](#).

---

## 6 A sample sketch: LoopBackDemoCANFD\_CAN1

The LoopBackDemoCANFD\_CAN1 sketch is a sample code for introducing the ACANFD\_FeatherM4CAN library. It demonstrates how to configure the library, to send a CANFD message, and to receive a CANFD message.

Note: this code runs without any CAN connection, the CAN1 module is configured in EXTERNAL\_LOOP\_BACK mode (see [section 22.10.1 page 47](#)); the CAN1 module receives every CANFD frame it sends, and emitted frames can be observed on CANH/CANL pins.

### ACANFD\_FeatherM4CAN configuration.

```
#define CAN0_MESSAGE_RAM_SIZE (0)
#define CAN1_MESSAGE_RAM_SIZE (1728)

#include <ACANFD_FeatherM4CAN.h>
```

Before including the ACANFD\_FeatherM4CAN library, you should define the CAN0\_MESSAGE\_RAM\_SIZE and the CAN1\_MESSAGE\_RAM\_SIZE macro names.

Each CANFD module uses a private *Message RAM* ([section 14 page 21](#)) that is in the first 64 kio of the microcontroller SRAM. Its size depends from the current module configuration, and cannot exceed 4,352 32-bits words (17,408 bytes). Here, CAN0\_MESSAGE\_RAM\_SIZE value is 0, meaning that the CAN0 module is not configured; its TxCAN and RxCAN pins can be freely used for an other function. CAN1\_MESSAGE\_RAM\_SIZE value is 1728, therefore CAN1 module Message RAM has a capacity of 1,728 32-bit words (6,912 bytes), that corresponds to the default configuration.

Note you should include <ACANFD\_FeatherM4CAN.h> only once, from the .ino source file. From an other C++ file, you should include <ACANFD\_FeatherM4CAN-from-cpp.h>.

If you include <ACANFD\_FeatherM4CAN.h> from several files, the can0 and / or can1 variables are multiply-defined, therefore you get a link error.

### The setup function.

```
void setup () {
  //--- Switch on builtin led
  pinMode (LED_BUILTIN, OUTPUT) ;
  digitalWrite (LED_BUILTIN, HIGH) ;
  //--- Start serial
  Serial.begin (115200) ;
  //--- Wait for serial (blink led at 10 Hz during waiting)
  while (!Serial) {
    delay (50) ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
  }
}
```

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

```
ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,
                                         1000 * 1000,
                                         DataBitRateFactor::x2) ;
```

---

Configuration is a four-step operation. This line is the first step. It instantiates the `settings` object of the `ACANFD_FeatherM4CAN_Settings` class. The constructor has three parameters: the clock selection, the desired CAN arbitration bit rate (here, 1 Mbit/s), and the data bit rate, given by a multiplicative factor of the arbitration bit rate; here, the data bit rate is  $1 \text{ Mbit/s} * 2 = 2 \text{ Mbit/s}$ . It returns a `settings` object fully initialized with CAN bit settings for the desired arbitration and data bit rates, and default values for other configuration properties.

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

This is the second step. You can override the values of the properties of `settings` object. Here, the `mModuleMode` property is set to `EXTERNAL_LOOP_BACK` – its value is `NORMAL_FD` by default. Setting this property enables *external loop back*, that is you can run this demo sketch even if you have no connection to a physical CAN network. The [section 22.10 page 46](#) lists all properties you can override.

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

This is the third step, configuration of the `CAN1` driver with `settings` values (for configuring the `CAN0` module, use the `can0` variable). The driver is configured for being able to send any (base / extended, data / remote, CAN / CANFD) frame, and to receive all (base / extended, data / remote, CAN / CANFD) frames. If you want to define reception filters, see [section 17 page 27](#).

```
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 21.2 page 37](#).

### The `pseudoRandomValue` function.

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

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

### The global variables.

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

The `gBlinkDate` global variable is used for sending a CAN message every second. The `gSentCount` global variable counts the number of sent messages. The sent message is stored in the `gSentFrame` variable. While `gOk` is true, the received message is compared to the sent message. If they are different, `gOk` is set to false, and

---

no more message is sent. The `gReceivedCount` global variable counts the number of successfully received messages.

### The Loop function.

```
void loop () {
  if (gBlinkDate <= millis ()) {
    gBlinkDate += PERIOD ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
    if (gOk) {
      ... build random CANFD frame ...
      const uint32_t sendStatus = can1.tryToSendReturnStatusFD (gSentFrame) ;
      if (sendStatus == 0) {
        gSentCount += 1 ;
        Serial.print ("Sent_") ;
        Serial.println (gSentCount) ;
      }else{
        Serial.print ("Sent_error_0x") ;
        Serial.println (sendStatus) ;
      }
    }
  }
}

//--- Receive frame
CANFDMessage frame ;
if (gOk && can1.receiveFD0 (frame)) {
  bool sameFrames = ... compare frame and gSentFrame ... ;
  if (sameFrames) {
    gReceiveCount += 1 ;
    Serial.print ("Received_") ;
    Serial.println (gReceiveCount) ;
  }else{
    gOk = false ;
    ... Print error ...
  }
}
}
```

## 7 The CANMessage class

**Note.** The `CANMessage` class is declared in the `ACANFD_FeatherM4CAN_CANMessage.h` header file. The class declaration is protected by an include guard that causes the macro `GENERIC_CAN_MESSAGE_DEFINED` to be defined. The `ACAN2515` driver<sup>7</sup>, the `ACAN2517` driver<sup>8</sup> and the `ACAN2517FD` driver<sup>9</sup> contain an header file

<sup>7</sup>The `ACAN2515` driver is a CAN driver for the MCP2515 CAN controller, <https://github.com/pierremolinaro/acan2515>.

<sup>8</sup>The `ACAN2517` driver is a CAN driver for the MCP2517FD CAN controller in CAN 2.0B mode, <https://github.com/pierremolinaro/acan2517>.

<sup>9</sup>The `ACAN2517FD` driver is a CANFD driver for the MCP2517FD CAN controller in CANFD mode, <https://github.com/pierremolinaro/acan2517fd>.

---

that declares an identical `CANMessage` class, enabling using the `ACANFD_FeatherM4CAN` driver, the `ACAN2515` driver, `ACAN2517` driver and `ACAN2517FD` driver in a same sketch.

A *CAN message* is an object that contains all CAN 2.0B frame user informations. All properties are initialized by default, and represent a base data frame, with an identifier equal to 0, and without any data. In this library, the `CANMessage` class is only used by a `CANFDMessage` constructor ([section 8.3 page 15](#)).

```
class CANMessage {
    public : uint32_t id = 0 ; // Frame identifier
    public : bool ext = false ; // false -> standard frame, true -> extended frame
    public : bool rtr = false ; // false -> data frame, true -> remote frame
    public : uint8_t idx = 0 ; // This field is used by the driver
    public : uint8_t len = 0 ; // Length of data (0 ... 8)
    public : union {
        uint64_t data64 ; // Caution: subject to endianness
        int64_t data_s64 ; // Caution: subject to endianness
        uint32_t data32 [2] ; // Caution: subject to endianness
        int32_t data_s32 [2] ; // Caution: subject to endianness
        float dataFloat [2] ; // Caution: subject to endianness
        uint16_t data16 [4] ; // Caution: subject to endianness
        int16_t data_s16 [4] ; // Caution: subject to endianness
        int8_t data_s8 [8] ;
        uint8_t data [8] = {0, 0, 0, 0, 0, 0, 0, 0} ;
    } ;
} ;
```

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 (Cortex M4F processor of the `ATSAME51G19A` is little-endian).

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

- for a received message, it contains the acceptance filter index (see [section 18 page 33](#)) or 255 if it does not correspond to any filter;
- on sending messages, it is used for selecting the transmit buffer (see [section 15 page 22](#)).

## 8 The `CANFDMessage` class

**Note.** The `CANFDMessage` class is declared in the `ACANFD_FeatherM4CAN_CANFDMessage.h` header file. The class declaration is protected by an include guard that causes the macro `GENERIC_CANFD_MESSAGE_DEFINED` to be defined. This allows an other library to freely include this file without any declaration conflict. The `ACAN2517FD` driver<sup>10</sup> contains an header file that declares an identical `CANFDMessage` class, enabling using the `ACANFD_FeatherM4CAN` driver and the `ACAN2517FD` driver in a same sketch.

---

<sup>10</sup>The `ACAN2517FD` driver is a CANFD driver for the `MCP2517FD` CAN controller in CANFD mode, <https://github.com/pierremolinaro/acan2517fd>.

## 8.1 Properties

---

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

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

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

## 8.1 Properties

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

Note the message datas are defined by an **union**. So message datas can be seen as 64 bytes, 32 x 16-bit unsigned integers, 16 x 32-bit, 8 x 64-bit or 16 x 32-bit floats. Be aware that multi-byte integers are subject to endianness (Cortex M4 processors of Teensy 3.x are little-endian).

## 8.2 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 4](#)).

## 8.3 Constructor from CANMessage

```
class CANFDMessage {
    ...
```

## 8.4 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>unitialized</i>

**Table 4** – CANFDMessage default constructor initialization

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

All properties are initialized from the `inCANMessage` ([table 5](#)). 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>
<code>data64[0]</code>	<code>inCANMessage.data64</code>

**Table 5** – CANFDMessage constructor `CANMessage`

## 8.4 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 6](#).



## 8.5 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 6** – CANFDMessage type property

## 8.5 The len property

Note that len property contains the actual length, not its encoding in CANFD frames. So valid values are: 0, 1, ..., 8, 12, 16, 20, 24, 32, 48, 64. Having other values is an error that prevents frame to be sent by the `ACANFD_FeatherM4CAN::tryToSendReturnStatusFD` method. You can use the `pad` method (see [section 8.7 page 17](#)) for padding with `0x00` bytes to the next valid length.

## 8.6 The idx property

The `idx` property is not used in CANFD frames, but it is used for selecting the transmit buffer (see [section 15 page 22](#)).

## 8.7 The pad method

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

The `CANFDMessage::pad` method appends zero bytes to `data` 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
```

## 8.8 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 6 page 17](#), and `false` otherwise.

---

## 9 Transmit FIFO

The transmit FIFO (see [figure 5 page 10](#)) is composed by:

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

For sending a message 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 (`TxBufferi`)

You can use `settings.mHardwareDedicatedTxBufferCount` `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 15.1 page 23](#)) for testing if a `TxBuffer` is empty or full.

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

## 11 Transmit Priority

Pending dedicated `TxBufferi` 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 `FIFOi` ( $0 \leq i \leq 1$ , see [figure 5 page 10](#)) is composed by:

- the *hardware receive `FIFOi`* (in the Message RAM, see [section 14 page 21](#)), whose size is 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 `FIFOi`* (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

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 21](#). The size of each object

### 13.1 The `ACANFD_FeatherM4CAN_Settings::wordCountForPayload` static method

depends on the setting applied to the corresponding FIFO or buffer.

By default, all objects accept frames up to 64 data bytes. The size of each object is 72 bytes. If your application sends and / or receives messages with less than 64 bytes, you can reduce Message RAM size by setting the payload properties of `ACANFD_FeatherM4CAN_Settings` class, as described in [table 7](#). The type of theses properties is the `ACANFD_FeatherM4CAN_Settings::Payload` enumeration type, and defines 8 values ([table 8](#)).

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

**Table 7** – Payload properties of `ACANFD_FeatherM4CAN_Settings` class

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

**Table 8** – `ACANFD_FeatherM4CAN_Settings` object size from payload size specification

### 13.1 The `ACANFD_FeatherM4CAN_Settings::wordCountForPayload` static method

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

This static method returns the object word size for a given payload specification, following [table 8](#).

### 13.2 The `ACANFD_FeatherM4CAN_Settings::frameDataByteCountForPayload` static method

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

This static method returns the handled data byte count for a given payload specification, following [table 8](#).

### 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_FeatherM4CAN_Settings::PAYLOAD_24_BYTES`, and a 32-byte data frame is submitted:

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

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

**mHardwareRxFIFO0Payload.** Setting the `mHardwareTransmitBufferPayload` property limits the size of hardware FIFO 0 elements. Received frame data bytes beyond this limit are not stored in the hardware FIFO 0. The retrived frame does not contain this data bytes, but `0xCC` bytes instead. For example, if it is set to `ACANFD_FeatherM4CAN_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

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

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

A message RAM contains<sup>11</sup>:

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

---

<sup>11</sup>See DS60001507G, section 39.9.1 page 1177.

---

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

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

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

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

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

- the `mHardwareRxFIFO0Size` property sets the hardware receive FIFO 0 element count (0-64);
- the `mHardwareRxFIFO0Payload` property sets the size of the hardware receive FIFO 0 element ([table 8](#));
- 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 8](#));
- the `mHardwareTransmitTxFIFOSize` property sets the hardware transmit FIFO element count (0-32);
- the `mHardwareDedicatedTxBufferCount` property set the number of dedicated TxBuffers (0-32);
- the `mHardwareTransmitBufferPayload` property sets the size of the TxBuffers and hardware transmit FIFO element ([table 8](#)).

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

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

## 15 Sending frames: the `tryToSendReturnStatusFD` method

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

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

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

## 15.1 Testing a send buffer: the `sendBufferNotFullForIndex` method

- `kInvalidMessage` (value: 1) if the message is not valid (see [section 8.8 page 17](#));
- `kTransmitBufferIndexTooLarge` (value: 2) if the `idx` property value does not specify a valid transmit buffer (see below);
- `kTransmitBufferOverflow` (value: 3) if the transmit buffer specified by the `idx` property value is full;
- 0 (no error) if the message has been successfully added to the transmit buffer specified by the `idx` property value.

The `idx` property of the message specifies the transmit buffer:

- 0 for the transmit FIFO ([section 9 page 18](#));
- 1 ... `settings.mHardwareDedicacedTxBufferCount` for a dedicaced TxBuffer ([section 10 page 19](#)).

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

- `CAN_REMOTE`, the frame is sent in the CAN 2.0B remote frame format;
- `CAN_DATA`, the frame is sent in the CAN 2.0B data frame format;
- `CANFD_NO_BIT_RATE_SWITCH`, the frame is sent in CANFD format at arbitration bit rate, regardless of the `ACANFD_FeatherM4CAN_Settings::DATA_BITRATE_xn` setting;
- `CANFD_WITH_BIT_RATE_SWITCH`, with the `ACANFD_FeatherM4CAN_Settings::DATA_BITRATE_x1` setting, the frame is sent in CANFD format at arbitration bit rate, and otherwise in CANFD format with bit rate switch.

## 15.1 Testing a send buffer: the `sendBufferNotFullForIndex` method

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

This method returns `true` if the corresponding transmit buffer is not full, and `false` otherwise ([table 9](#)).

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

**Table 9** – Value returned by the `sendBufferNotFullForIndex` method

## 15.2 Usage example

A way is to use a global variable to note if the message has been successfully transmitted to driver transmit buffer. For example, for sending a message every 2 seconds:

```

static uint32_t gSendDate = 0 ;

void loop () {
    if (gSendDate < millis ()) {
        CANFDMessage message ;
        // Initialize message properties
        const 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 ;
        }
    }
    ...
}

```

## 16 Retrieving received messages using the receiveFD*i* method

```

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

```

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

This is a basic example:

```

void loop () {
    CANFDMessage message ;

```



---

```

if (can0.receiveFD0 (message)) {
    // Handle received message
}
...
}

```

The receive method:

- returns false if the driver receive buffer is empty, message argument is not modified;
- returns true if a message has been removed from the driver receive buffer, and the message argument is assigned.

The type property contains the received frame format:

- CAN\_REMOTE, the received frame is a CAN 2.0B remote frame;
- CAN\_DATA, the received frame is a CAN 2.0B data frame;
- CANFD\_NO\_BIT\_RATE\_SWITCH, the frame received frame is a CANFD frame, received at at arbitration bit rate;
- CANFD\_WITH\_BIT\_RATE\_SWITCH, the frame received frame is a CANFD frame, received with bit rate switch.

You need to manually dispatch the received messages. If you did not provide any receive filter, you should check the type property (remote or data frame?), the ext bit (base or extended frame), and the id (identifier value). The following snippet dispatches three messages:

```

void loop () {
    CANFDMessage message ;
    if (can0.receiveFD0 (message)) {
        if (!message.rtr && message.ext && (message.id == 0x123456)) {
            handle_myMessage_0 (message) ; // Extended data frame, id is 0x123456
        } else if (!message.rtr && !message.ext && (message.id == 0x234)) {
            handle_myMessage_1 (message) ; // Base data frame, id is 0x234
        } else if (message.rtr && !message.ext && (message.id == 0x542)) {
            handle_myMessage_2 (message) ; // Base remote frame, id is 0x542
        }
    }
    ...
}

```

The handle\_myMessage\_0 function has the following header:

```

void handle_myMessage_0 (const CANFDMessage & inMessage) {
    ...
}

```

So are the header of the handle\_myMessage\_1 and the handle\_myMessage\_2 functions.

## 16.1 Driver receive FIFO *i* size

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

```
ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,  
                                       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`.

## 16.2 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 () ;
```

## 16.3 The `driverReceiveFIFOiCount` method

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

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

## 16.4 The `driverReceiveFIFOiPeakCount` method

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

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

If an overflow occurs, further calls of `can0.driverReceiveFIFOiPeakCount ()` return `can0.driverReceiveFIFOiSize ()+1`.

## 16.5 The `resetDriverReceiveFIFOiPeakCount` method

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

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

---

## 17 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.

### 17.1 Acceptance filters for standard frames

for an example sketch, see `LoopBackDemoCANFD_CAN1_StandardFilters`.

You have three ways to act on standard frame filtering:

- setting the `mDiscardReceivedStandardRemoteFrames` property of the `ACANFD_FeatherM4CAN_Settings` class discards every received remote frame (it is `false` by default);
- the `mNonMatchingStandardFrameReception` property value of the `ACANFD_FeatherM4CAN_Settings` class is applied to every standard frame that do not match any filter; its value can be `FIFO0` (default), `FIFO1` or `REJECT`;
- define standard filters (as described from [section 17.1.1 page 27](#)), up to 128, none by default.

The standard frame filtering is illustrated by [figure 6](#).

#### 17.1.1 Defining standard frame filters

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

The `ACANFD_FeatherM4CAN::StandardFilters` class handles a standard frame filter list. Default constructor constructs an empty list. For appending filters, use the `addSingle` ([section 17.1.2 page 27](#)), `addDual` ([section 17.1.3 page 28](#)), `addRange` ([section 17.1.4 page 29](#)) or `addClassic` ([section 17.1.5 page 29](#)) 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 6](#) for detail).

#### 17.1.2 Add single filter

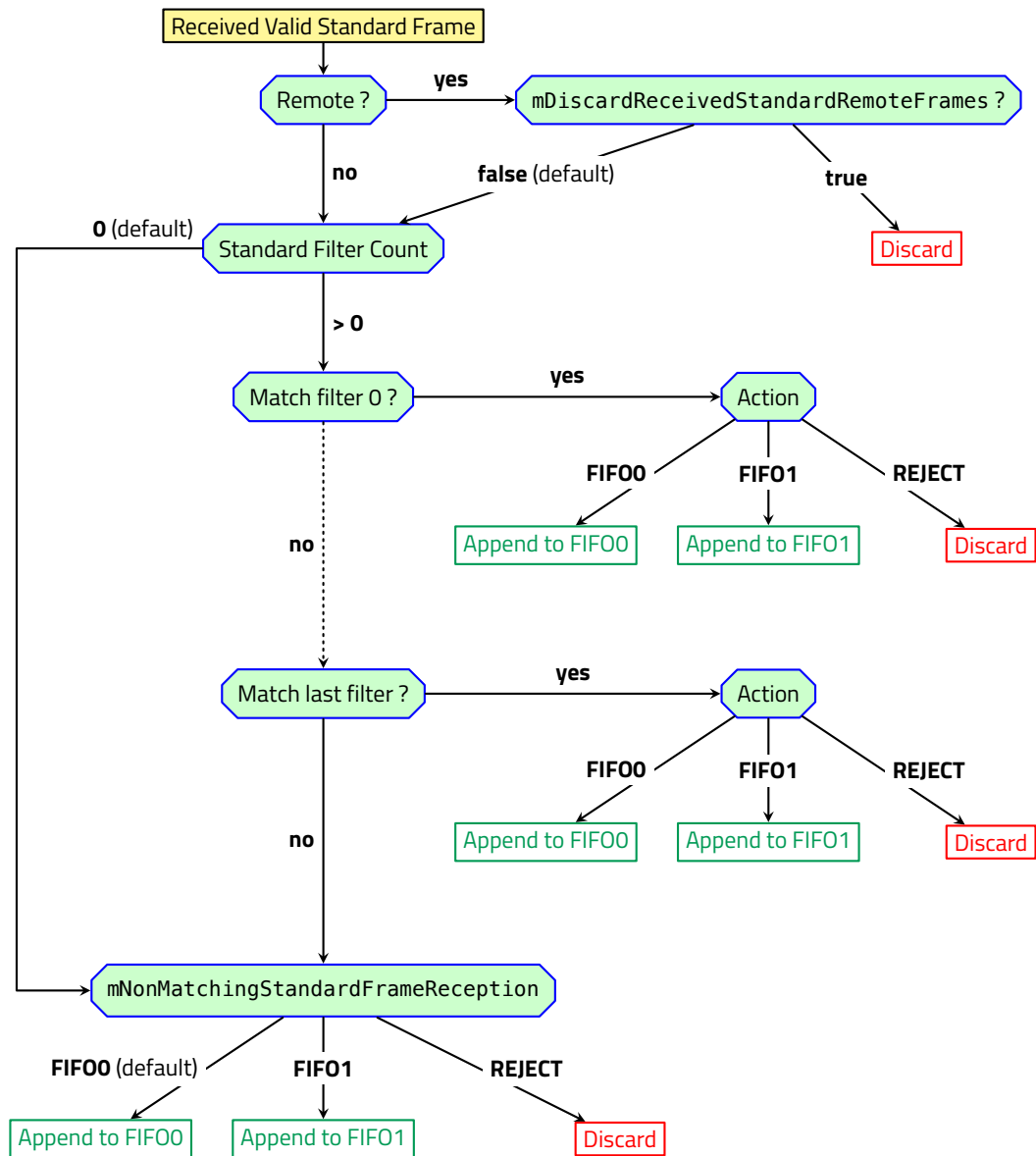


Figure 6 – Standard frame filtering

```

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

```

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

The last argument is optional and associates a callback routine to the filter. See [section 18 page 33](#).

### 17.1.3 Add dual filter

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

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

The last argument is optional and associates a callback routine to the filter. See [section 18 page 33](#).

### 17.1.4 Add range filter

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

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

The last argument is optional and associates a callback routine to the filter. See [section 18 page 33](#).

### 17.1.5 Add classic filter

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

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

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

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

- if a mask bit is a 1, the received standard frame identifier corresponding bit should match the `inIdentifier` corresponding bit;

## 17.2 Acceptance filters for extended frames

- 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 18 page 33](#).

For example:

```
standardFilters.addClassic (0x405, 0x7D5, ACANFD_FeatherM4CAN_FilterAction::FIF00) ;
```

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.

## 17.2 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 `FIF00` (default), `FIF01` or `REJECT`;
- define extended filters (as described from [section 17.2.1 page 30](#)), up to 128, none by default.

The extended frame filtering is illustrated by [figure 7](#).

### 17.2.1 Defining extended frame filters

```
ACANFD_FeatherM4CAN_Settings settings (... , ...) ;
...
ACANFD_FeatherM4CAN::ExtendedFilters extendedFilters ;
extendedFilters.addSingle (0x55, ACANFD_FeatherM4CAN_FilterAction::FIF00) ;
...
//--- Reject extended frames that do not match any filter
settings.mNonMatchingExtendedFrameReception = ACANFD_FeatherM4CAN_FilterAction::REJECT;
...
const uint32_t errorCode = can1.beginFD (settings, extendedFilters) ;
...
```

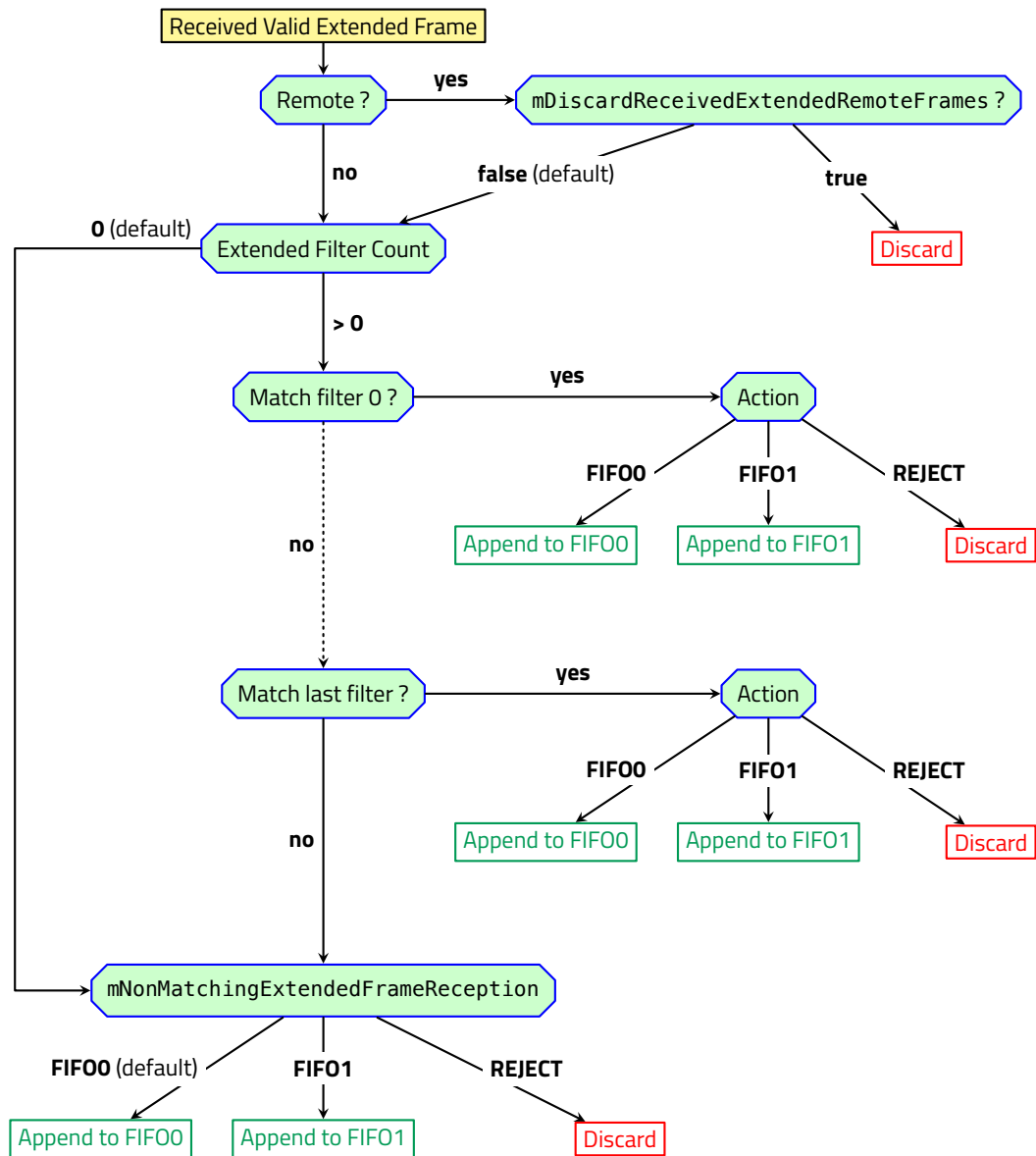


Figure 7 – Extended frame filtering

The `ACANFD_FeatherM4CAN::ExtendedFilters` class handles an extended frame filter list. Default constructor constructs an empty list. For appending filters, use the `addSingle` (section 17.2.2 page 31), `addDual` (section 17.2.3 page 32), `addRange` (section 17.2.4 page 32) or `addClassic` (section 17.2.5 page 32) methods. Then, add the `ExtendedFilters` as second argument of `beginFD` call.

**Note.** Do not forget to set `settings.mNonMatchingExtendedFrameReception` to `REJECT`, otherwise all frames rejected by the filters are appended to FIFO 0 (see figure 7 for detail).

### 17.2.2 Add single filter

```
bool ExtendedFilters::addSingle (const uint32_t inIdentifier,
                                const ACANFD_FeatherM4CAN_FilterAction inAction,
```

```
const ACANFDCallbackRoutine inCallback = nullptr) ;
```

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

The last argument is optional and associates a callback routine to the filter. See [section 18 page 33](#).

### 17.2.3 Add dual filter

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

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

The last argument is optional and associates a callback routine to the filter. See [section 18 page 33](#).

### 17.2.4 Add range filter

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

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

The last argument is optional and associates a callback routine to the filter. See [section 18 page 33](#).

### 17.2.5 Add classic filter

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

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

- `inIdentifier` is lower or equal to `0x1FFF_FFFF`;



- inMask is lower or equal to 0x1FFF\_FFFF;
- (inIdentifier & inMask) is equal to inIdentifier.

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

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

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

The last argument is optional and associates a callback routine to the filter. See [section 18 page 33](#).

For example:

```
extendedFilters.addClassic (0x6789, 0x1FFF67BD, ACANFD_FeatherM4CAN_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
inIdentifier: 0x6789	0	0	1	1	0	0	1	1	1	1	1	0	0	0	1	0	0	1
inMask: 0x1FFF67BD	1	0	1	1	0	0	1	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	1	0	<i>x</i>	1

Therefore there are 32 matching identifiers.

## 18 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 () {
  can1.dispatchReceivedMessage () ; // Do not call can1.receiveFD0, can1.receiveFD1 any more
  ...
}
```

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 ()) {  
    }  
    ...  
}
```

### 18.1 Dispatching non matching standard frames

Following the [figure 6 page 28](#), 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_FeatherM4CAN_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.

### 18.2 Dispatching non matching extended frames

Following the [figure 7 page 31](#), 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_FeatherM4CAN_Settings` class. This provides a callback function to non matching extended frames, so they are dispatched by the `dispatchReceivedMessage` method. By default, `mNonMatchingExtendedMessageCallback` value is `nullptr`.

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

---

## 19 The `dispatchReceivedMessageFIFO0` method

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

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

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

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

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

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

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

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

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

## 20 The `dispatchReceivedMessageFIFO1` method

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

```
void loop () {
```

---

```

can1.dispatchReceivedMessageFIF01 () ; // Do not call can1.receiveFD1 any more
CANFDMessage ;
if (can1.receiveFD0 (message)) {
    ... handle FIF00 message ...
}
...
}

```

Instead of calling the `receiveFD1` method, call the `dispatchReceivedMessageFIF01` method in your loop function. For every message extracted from FIF01, it calls the callback function associated with the corresponding filter.

If you have not defined any filter that targets the FIF01, do not use this function (messages will be not dispatched and therefore lost), call the `receiveFD1` method.

The `dispatchReceivedMessageFIF01` method handles one FIF01 message on each call. Specifically:

- if FIF01 is empty, it returns false;
- if FIF01 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.dispatchReceivedMessageFIF01 ()) {
    }
    CANFDMessage ;
    if (can1.receiveFD0 (message)) {
        ... handle FIF00 message ...
    }
    ...
}

```

## 21 The `ACANFD_FeatherM4CAN::beginFD` method reference

### 21.1 The prototypes

```

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

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

```

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

## 21.2 The error codes

The second one is optional, and specifies the standard filter list (see [section 17.1 page 27](#)). By default, the standard filter list is empty.

The third one is optional, and specifies the extended filter list (see [section 17.2 page 30](#)). By default, the extended filter list is empty.

## 21.2 The error codes

The `ACANFD_FeatherM4CAN::beginFD` method returns an error code. The value `0` denotes no error. Otherwise, you consider every bit as an error flag, as described in [table 10](#). An error code could report several errors. The `ACANFD_FeatherM4CAN` class defines static constants for naming errors. Bits 0 to 16 denote a bit configuration error, see [table 12 page 44](#).

Bit	Code	Static constant Name	Comment
0	0x1	kBitRatePrescalerIsZero	See <a href="#">table 12 page 44</a>
...	...	....	See <a href="#">table 12 page 44</a>
16	0x1_0000	kDataSJWIsGreaterThanOrEqualToPhaseSegment2	See <a href="#">table 12 page 44</a>
20	0x10_0000	kMessageRamTooSmall	See <a href="#">section 14 page 21</a>
21	0x20_0000	kMessageRamNotInFirst64kio	See <a href="#">section 14 page 21</a>
22	0x40_0000	kHardwareRxFIFO0SizeGreaterThan64	settings.mHardwareRxFIFO0Size > 64
23	0x80_0000	kHardwareTransmitFIFOSizeGreaterThan32	settings.mHardwareTransmitTxFIFOSize > 32
24	0x100_0000	kDedicacedTransmitTxBufferCountGreaterThan30	settings.mHardwareDedicacedTxBufferCount > 30
25	0x200_0000	kTxBufferCountGreaterThan32	See <a href="#">section 21.2.1 page 37</a>
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 <a href="#">section 17.1 page 27</a>
29	0x2000_0000	kExtendedFilterCountGreaterThan128	More than 128 extended filters, see <a href="#">section 17.2 page 30</a>

**Table 10** – The `ACANFD_FeatherM4CAN::beginFD` method error code bits

### 21.2.1 The `kTxBufferCountGreaterThan32` error code

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

## 22 ACANFD\_FeatherM4CAN\_Settings class reference

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

## 22.1 The ACANFD\_FeatherM4CAN\_Settings constructors: computation of the CAN bit settings

### 22.1.1 6 arguments constructor

```
ACANFD_FeatherM4CAN_Settings::  
ACANFD_FeatherM4CAN_Settings (const ClockSelection inClockSelection,  
                               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\_FeatherM4CAN\_Settings five mandatory arguments:

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

It tries to compute the CAN bit settings for these 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 8 page 42](#).

For example, for an 1 Mbit/s arbitration bit rate and an 8 Mbit/s data bit rate:

```
void setup () {  
    // Arbitration bit rate: 1 Mbit/s, data bit rate: 8 Mbit/s  
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz, 1000 * 1000, 75, DataBitRateFactor::x8,  
    // Here, settings.mArbitrationBitRateClosedToDesiredRate is true  
    ...  
}
```

Note the data bit rate is not defined by its frequency, but by its multiplicative factor from arbitration bit rate. If you want a single bit rate, use `DataBitRateFactor::x1` as data bit rate factor.

### 22.1.2 4-arguments constructor

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

## 22.1 The ACANFD\_FeatherM4CAN\_Settings constructors: computation of the CAN bit settings

```
ACANFD_FeatherM4CAN_Settings::
ACANFD_FeatherM4CAN_Settings (const ClockSelection inClockSelection,
                             const uint32_t inDesiredArbitrationBitRate,
                             const DataBitRateFactor inDataBitRateFactor,
                             const uint32_t inTolerancePPM = 1000) ;
```

### 22.1.3 Exact bit rates

There are 313 exact arbitration / data bit rate combinations ([table 11 page 39](#)).

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

**Table 11** – The 313 exact bit rates

But this does not mean there is no possibility to get such data bit rates factors. For example, we can have a data bit rate of 4 Mbit/s, and an arbitration bit rate of 4/7 Mbit/s = 571 428 kbit/s:

```
void setup () {
    ...
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz, 571428, DataBitRateFactor
    Serial.print ("mArbitrationBitRateClosedToDesiredRate:");
    Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (true)
```

## 22.1 The ACANFD\_FeatherM4CAN\_Settings constructors: computation of the CAN bit settings

```
Serial.print ("Actual_Arbitration_Bit_Rate:_");  
Serial.println (settings.actualArbitrationBitRate ()) ; // 571428 bit/s  
Serial.print ("distance:_");  
Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 1 ppm= 0,0001 %  
Serial.print ("Actual_Data_Bit_Rate:_");  
Serial.println (settings.actualDataBitRate ()) ; // 4 Mbit/s  
...  
}
```

Due to integer computations, and the distance from desired arbitration bit rate is 1 ppm. "ppm" stands for "part-per-million", and  $1 \text{ ppm} = 10^{-6}$ . In other words,  $10,000 \text{ ppm} = 1\%$ .

By default, a desired bit rate is accepted if the distance from the computed actual bit rate is lower or equal to  $1,000 \text{ ppm} = 0.1\%$ . You can change this default value by adding your own value as third argument of ACANFD\_FeatherM4CAN\_Settings constructor. For example, with an arbitration bit rate equal to 727 kbit/s:

```
void setup () {  
    ...  
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,  
                                             727 * 1000,  
                                             DataBitRateFactor::x1,  
                                             100) ; // 100 ppm  
    Serial.print ("mArbitrationBitRateClosedToDesiredRate:_");  
    Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (false)  
    Serial.print ("actual_arbitration_bit_rate:_");  
    Serial.println (settings.actualArbitrationBitRate ()) ; // 727272 bit/s  
    Serial.print ("distance:_");  
    Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 375 ppm  
    ...  
}
```

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

```
void setup () {  
    ...  
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,  
                                             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  
    ...  
}
```



## 22.1 The ACANFD\_FeatherM4CAN\_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  $423 \text{ kbit/s} * 3 = 1\,269 \text{ kbit/s}$  data bit rate:

```
void setup () {
    ...
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,
                                           423 * 1000,
                                           DataBitRateFactor::x3) ;

    Serial.print ("mArbitrationBitRateClosedToDesiredRate: ") ;
    Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 0 (false)
    Serial.print ("Actual Arbitration Bit Rate: ") ;
    Serial.println (settings.actualArbitrationBitRate ()) ; // 421 052 bit/s
    Serial.print ("Actual Data Bit Rate: ") ;
    Serial.println (settings.actualDataBitRate ()) ; // 1 263 157 bit/s
    Serial.print ("distance: ") ;
    Serial.println (settings.ppmFromDesiredArbitrationBitRate ()) ; // 4 603 ppm
    ...
}
```

The resulting bit rates settings are far from the desired values, the CAN bit decomposition is consistent. You can get its details:

```
void setup () {
    ...
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,
                                           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: ") ;
}
```

## 22.1 The ACANFD\_FeatherM4CAN\_Settings constructors: computation of the CAN bit settings

```
Serial.println (settings.mDataPhaseSegment2) ; // PS2 = 10
Serial.print ("Data_Resynchronization_Jump_Width:");
Serial.println (settings.mDataSJW) ; // SJW = 10
Serial.print ("Data_Sample_Point:");
Serial.println (settings.dataSamplePointFromBitStart ()) ; // 69, meaning 59%
Serial.print ("Consistency:");
Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning 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 8](#), you can increment the `mArbitrationPhaseSegment1` property value, and decrement the `mArbitrationPhaseSegment2` property value in order to sample the CAN Rx pin later.

```
void setup () {
    ...
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,
                                           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 8** – Adapting property values

Be aware to always respect CAN bit timing consistency! The ATSAME51G19A constraints are:

## 22.2 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 (SYSCLK = 48 MHz):

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

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

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

## 22.2 The CANBitSettingConsistency method

This method checks the CAN bit decomposition (given by mBitRatePrescaler, mArbitrationPhaseSegment1, mArbitrationPhaseSegment2, mArbitrationSJW, mDataPhaseSegment1, mDataPhaseSegment2, mDataSJW property values) is consistent.

```
void setup () {
    ...
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,
                                           500 * 1000,
                                           DataBitRateFactor::x2) ;

    Serial.print ("mArbitrationBitRateClosedToDesiredRate:\u0000") ;
    Serial.println (settings.mArbitrationBitRateClosedToDesiredRate) ; // 1 (true)
    settings.mDataPhaseSegment1 = 0 ; // Error, mDataPhaseSegment1 should be >= 1 (and <= 32)
    Serial.print ("Consistency:\u0000x") ;
    Serial.println (settings.CANBitSettingConsistency (), HEX) ; // != 0, meaning error
    ...
}
```

## 22.3 The actualArbitrationBitRate method

The CANBitSettingConsistency method returns 0 if CAN bit decomposition is consistent. Otherwise, the returned value is a bit field that can report several errors – see [table 12](#).

The ACANFD\_FeatherM4CAN\_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>not used</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 12** – The ACANFD\_FeatherM4CAN\_Settings::CANBitSettingConsistency method error codes

## 22.3 The actualArbitrationBitRate method

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

```
void setup () {  
    ...  
    ACANFD_FeatherM4CAN_Settings settings (ACANFD_FeatherM4CAN_Settings::CLOCK_48MHz,  
                                             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 22.2 page 43](#)), the returned value is irrelevant.

### 22.4 The exactArbitrationBitRate method

```
bool ACANFD_FeatherM4CAN_Settings::exactArbitrationBitRate (void) const ;
```

The `exactArbitrationBitRate` method returns `true` if the actual arbitration bit rate is equal to the desired arbitration bit rate, and `false` otherwise.

**Note.** If CAN bit settings are not consistent (see [section 22.2 page 43](#)), the returned value is irrelevant.

### 22.5 The exactDataBitRate method

```
bool ACANFD_FeatherM4CAN_Settings::exactDataBitRate (void) const ;
```

The `exactDataBitRate` method returns `true` if the actual data bit rate is equal to the desired data bit rate, and `false` otherwise.

**Note.** If CAN bit settings are not consistent (see [section 22.2 page 43](#)), the returned value is irrelevant.

### 22.6 The ppmFromDesiredArbitrationBitRate method

```
uint32_t ACANFD_FeatherM4CAN_Settings::ppmFromDesiredArbitrationBitRate (void) const ;
```

The `ppmFromDesiredArbitrationBitRate` method returns the distance from the actual arbitration bit rate to the desired arbitration bit rate, expressed in part-per-million (ppm):  $1 \text{ ppm} = 10^{-6}$ . In other words,  $10,000 \text{ ppm} = 1\%$ .

**Note.** If CAN bit settings are not consistent (see [section 22.2 page 43](#)), the returned value is irrelevant.

### 22.7 The ppmFromDesiredDataBitRate method

```
uint32_t ACANFD_FeatherM4CAN_Settings::ppmFromDesiredDataBitRate (void) const ;
```

The `ppmFromDesiredDataBitRate` method returns the distance from the actual data bit rate to the desired data bit rate, expressed in part-per-million (ppm):  $1 \text{ ppm} = 10^{-6}$ . In other words,  $10,000 \text{ ppm} = 1\%$ .

**Note.** If CAN bit settings are not consistent (see [section 22.2 page 43](#)), the returned value is irrelevant.

### 22.8 The arbitrationSamplePointFromBitStart method

```
uint32_t ACANFD_FeatherM4CAN_Settings::arbitrationSamplePointFromBitStart (void) const ;
```

The `arbitrationSamplePointFromBitStart` method returns the distance of sample point from the start of the arbitration CAN bit, expressed in part-per-cent (ppc):  $1 \text{ ppc} = 1\% = 10^{-2}$ . It is a good practice to get sample point from 65% to 80%. The bit rate calculator tries to set the sample point at 80%.

**Note.** If CAN bit settings are not consistent (see [section 22.2 page 43](#)), the returned value is irrelevant.

## 22.9 The dataSamplePointFromBitStart method

```
uint32_t ACANFD_FeatherM4CAN_Settings::dataSamplePointFromBitStart (void) const ;
```

The dataSamplePointFromBitStart method returns the distance of sample point from the start of the data CAN bit, expressed in part-per-cent (ppc): 1 ppc = 1% =  $10^{-2}$ . It is a good practice to get sample point from 65% to 80%. The bit rate calculator tries to set the sample point at 80%.

**Note.** If CAN bit settings are not consistent (see [section 22.2 page 43](#)), the returned value is irrelevant.

## 22.10 Properties of the ACANFD\_FeatherM4CAN\_Settings class

All properties of the ACANFD\_FeatherM4CAN\_Settings class are declared public and are initialized ([table 13](#)).

Property	Type	Initial value	Comment
mDesiredArbitrationBitRate	uint32_t	Constructor argument	
mDataBitRateFactor	DataBitRateFactor	Constructor argument	
mBitRatePrescaler	uint8_t	32	See <a href="#">section 22.1 page 38</a>
mArbitrationPhaseSegment1	uint16_t	256	See <a href="#">section 22.1 page 38</a>
mArbitrationPhaseSegment2	uint8_t	128	See <a href="#">section 22.1 page 38</a>
mArbitrationSJW	uint8_t	128	See <a href="#">section 22.1 page 38</a>
mDataPhaseSegment1	uint8_t	32	See <a href="#">section 22.1 page 38</a>
mDataPhaseSegment2	uint8_t	16	See <a href="#">section 22.1 page 38</a>
mDataSJW	uint8_t	16	See <a href="#">section 22.1 page 38</a>
mBitSettingOk	bool	true	See <a href="#">section 22.1 page 38</a>
mModuleMode	ModuleMode	NORMAL_FD	See <a href="#">section 22.10.1 page 47</a>
mDriverReceiveFIFO0Size	uint16_t	10	See <a href="#">section 16.1 page 26</a>
mHardwareRxFIFO0Size	uint8_t	64	See <a href="#">section 14 page 21</a>
mHardwareRxFIFO0Payload	Payload	PAYLOAD_64_BYTES	See <a href="#">section 14 page 21</a>
mDriverReceiveFIFO1Size	uint16_t	0	See <a href="#">section 16.1 page 26</a>
mHardwareRxFIFO1Size	uint8_t	0	See <a href="#">section 14 page 21</a>
mHardwareRxFIFO1Payload	Payload	PAYLOAD_64_BYTES	See <a href="#">section 14 page 21</a>
mEnableRetransmission	bool	true	See <a href="#">section 22.10.2 page 47</a>
mDiscardReceivedStandardRemoteFrames	bool	false	See <a href="#">section 17 page 27</a>
mDiscardReceivedExtendedRemoteFrames	bool	false	See <a href="#">section 17 page 27</a>
mNonMatchingStandardFrameReception	FilterAction	FIFO0	See <a href="#">section 17 page 27</a>
mNonMatchingExtendedFrameReception	FilterAction	FIFO0	See <a href="#">section 17 page 27</a>
mTransceiverDelayCompensation	uint8_t	5	See <a href="#">section 22.10.3 page 47</a>
mDriverTransmitFIFOSize	uint8_t	20	See <a href="#">section 9 page 18</a>
mHardwareTransmitTxFIFOSize	uint8_t	24	See <a href="#">section 9 page 18</a>
mHardwareDedicatedTxBufferCount	uint8_t	8	See <a href="#">section 10 page 19</a>
mHardwareTransmitBufferPayload	Payload	PAYLOAD_64_BYTES	See <a href="#">section 13 page 19</a>
mNonMatchingStandardMessageCallback	ACANFD_CallbackRoutine	nullptr	See <a href="#">section 18.1 page 34</a>
mNonMatchingExtendedMessageCallback	ACANFD_CallbackRoutine	nullptr	See <a href="#">section 18.2 page 34</a>

**Table 13** – Properties of the ACANFD\_FeatherM4CAN\_Settings class

### 22.10.1 The mModuleMode property

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

**BUS\_MONITORING mode.** See DS60001507G datasheet, section 39.6.2.6 page 1096.

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

**INTERNAL\_LOOP\_BACK mode.** See DS60001507G datasheet, section 39.6.2.8 page 1098.

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

**EXTERNAL\_LOOP\_BACK mode.** See DS60001507G datasheet, section 39.6.2.8 page 1098.

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

### 22.10.2 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`.

### 22.10.3 The mTransceiverDelayCompensation property

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

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

---

## 23 Other ACANFD\_FeatherM4CAN methods

### 23.1 The getStatus method

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

#### 23.1.1 The txErrorCount method

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

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

#### 23.1.2 The rxErrorCount method

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

This method returns the *Receive Error Counter* value.

#### 23.1.3 The isBusOff method

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

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

#### 23.1.4 The transceiverDelayCompensationOffset method

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

This method returns *Transceiver Delay Compensation Offset* value.

#### 23.1.5 The hardwareTxBufferPayload method

```
ACANFD_FeatherM4CAN_Settings::Payload ACANFD_FeatherM4CAN::hardwareTxBufferPayload (void) const ;
```

This method returns the payload of transmit TxBuffers.

#### 23.1.6 The hardwareRxFIFO0Payload method

```
ACANFD_FeatherM4CAN_Settings::Payload ACANFD_FeatherM4CAN::hardwareRxFIFO0Payload (void) const ;
```

This method returns the payload of hardware receive FIFO 0.



## 23.1 The getStatus method

---

### 23.1.7 The hardwareRxFIFO1Payload method

```
ACANFD_FeatherM4CAN_Settings::Payload ACANFD_FeatherM4CAN::hardwareRxFIFO1Payload (void) const ;
```

This method returns the payload of hardware receive FIFO 1.