# ACAN_STM32 Arduino library for Nucleo Boards Version 1.0.0

Pierre Molinaro

March 16, 2022

## Contents

# 1 Versions

| Version | Date | Comment |
|---|---|---|
| 1.0.0 | March ??, 2022 | Initial release. |

# 2 Features

The ACAN_STM32 library is a CAN (*Controller Area Network*) Controller driver for Nucleo boards.

This library is compatible with the ACAN2515[1], the ACAN2515Tiny[2] and the ACAN2517[3] libraries.

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

- default configuration sends and receives any frame – no default filter to provide;
- efficient built-in CAN bit settings computation from bit rates;
- user can fully define its own CAN bit setting values;
- up to 14 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;
- *internal loop back*, *external loop back*, *silent* controller modes are selectable.

# 3 Handled boards

Currently, two boards are handled (table 2).

| Nucleo 32 Board | Variable | TxCAN | RxCAN | CAN_CLOCK_FREQUENCY |
|---|---|---|---|---|
| NUCLEO-F303K8 | can | PA12 == D2 | PA11 == D3 | 32 MHz |
| NUCLEO-L432KC | can | PA12 == D2 | PA11 == D3 | 80 MHz |

**Table 2** – Nucleo 32 supported boards

[1] https://github.com/pierremolinaro/acan2515
[2] https://github.com/pierremolinaro/acan2515Tiny
[3] https://github.com/pierremolinaro/acan2517

# 4 Data flow

The figure 1 illustrates default message flow of sending and receiving CAN messages.

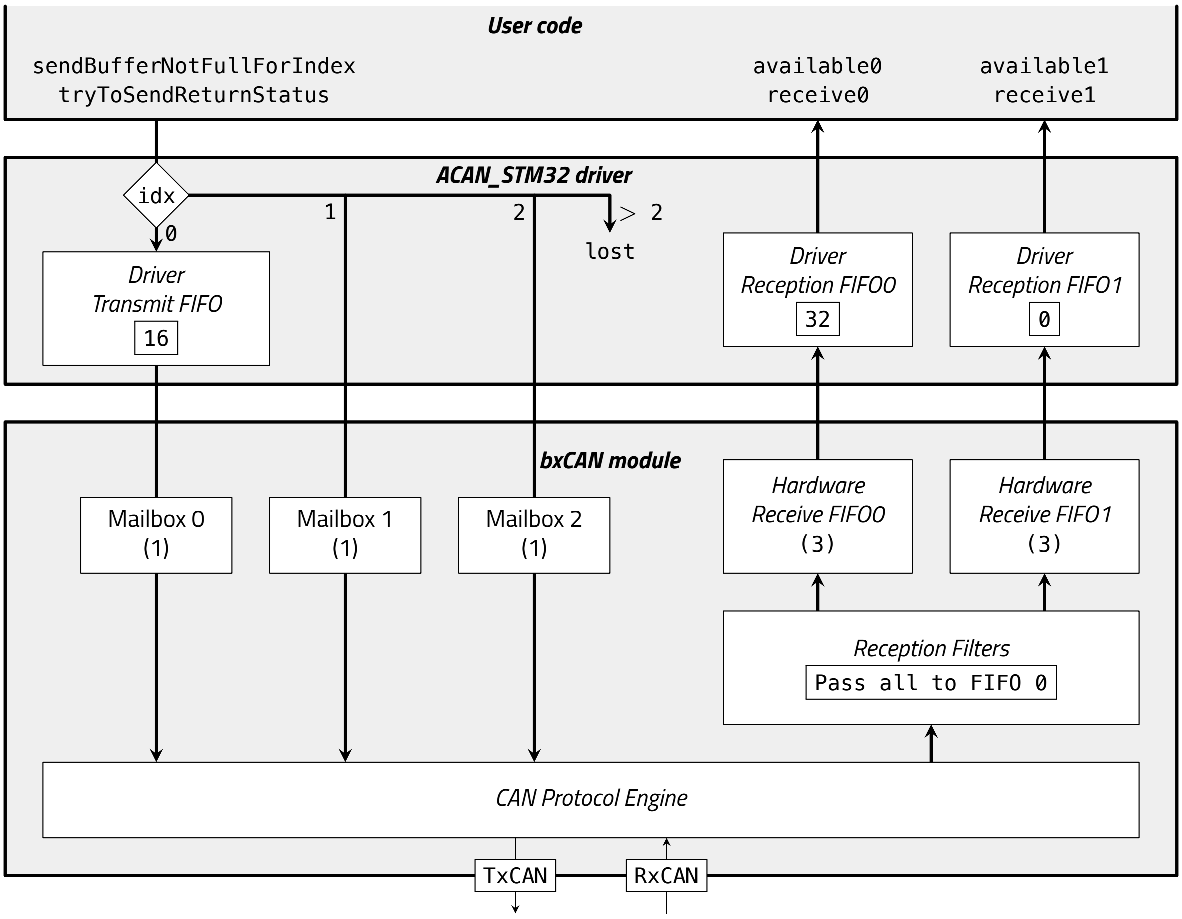


**Figure 1** – Message flow in ACAN_STM32 driver and bxCAN module, default configuration

**Sending messages.** The ACAN_STM32 driver defines a *driver transmit FIFO* (default size: 16 messages), and 3 individual `Mailboxes` whose capacity is one message.

A message is defined by an instance of the `CANMessage` class. For sending a message, user code calls the `tryToSendReturnStatus` method – see section 10 page 9 for details, and the `idx` property of the sent message should be:

- 0 (default value), for sending via *driver transmit FIFO* and *Mailbox 0*;
- 1, for sending via *Mailbox 1*;
- 2, for sending via *Mailbox 2*.

If the `idx` property is greater than 2, the message is lost.

You can call the `sendBufferNotFullForIndex` method (section 10.1 page 10) 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 12 page 13 for configuring them. Messages that pass the filters are stored in the *Hardware Reception FIFO0* or in the *Hardware Reception FIFO1*. A hardware reception FIFO has a capacity of 3 messages. 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 11.1 page 12 for changing the default value. Seven user methods are available:

- the `available0` method returns `false` if the *Driver Receive FIFO0* is empty, and `true` otherwise;
- the `receive0` method retrieves messages from the *Driver Receive FIFO0* – see section 11 page 11;
- the `available1` method returns `false` if the *Driver Receive FIFO1* is empty, and `true` otherwise;
- the `receive1` method retrieves messages from the *Driver Receive FIFO1* – see section 11 page 11;
- the `dispatchReceivedMessage` method, see section 13 page 18;
- the `dispatchReceivedMessage0` method, see section 14 page 19;
- the `dispatchReceivedMessage1` method, see section 15 page 19.

# 5 A sample sketch: LoopBackDemo

The LoopBackDemo sketch is a sample code for introducing the ACAN_STM32 library. It demonstrates how to configure the library, to send a CAN message, and to receive a CAN message.

Note: this code runs without any CAN connection, the CAN module is configured in EXTERNAL_LOOP_BACK mode (see section 17.10.1 page 27); the CAN1 module receives every CANFD frame it sends, and emitted frames can be observed on TxCAN pin.

**ACAN_STM32 inclusion.**

```
#include <ACAN_STM32.h>
```

**The `setup` function.**

```
void setup () {
//--- Switch on builtin led
  pinMode (LED_BUILTIN, OUTPUT) ;
//--- 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.

```
  ACAN_STM32_Settings settings (125 * 1000) ;
```

Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACAN_STM32_Settings` class. The constructor has one parameter: the desired CAN bit rate (here, 125 kbit/s). It returns a `settings` object fully initialized with CAN bit settings for the desired bit rate, and default values for other configuration properties.

```
settings.mModuleMode = ACAN_STM32_Settings::EXTERNAL_LOOP_BACK ;
```

This is the second step. You can override the values of the properties of `settings` object. Here, the `mModuleMode` property is set to `EXTERNAL_LOOP_BACK` – its value is `NORMAL` by default. Setting this property enables *external loop back*, that is you can run this demo sketch even it you have no connection to a physical CAN network. The section 17.10 page 27 lists all properties you can override.

```
  const uint32_t errorCode = can.begin () ;
```

This is the third step, configuration of the `CAN` driver with `settings` values. The driver is configured for being able to send any (standard / extended, data / remote) frame, and to receive all (standard / extended, data / remote) frames. If you want to define reception filters, see section 12 page 13.

```
  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 16.2 page 21.

**The global variables.**

```
static uint32_t gSendDate = 0 ;
static uint32_t gSentCount = 0 ;
static uint32_t gReceivedCount = 0 ;
```

The `gBlinkDate` global variable is used for sending a CAN message every second. The `gSentCount` global variable counts the number of sent messages. The `gReceivedCount` global variable counts the number of sucessfully received messages.

**The `loop` function.**

```
void loop () {
  CANMessage message ;
  if (gSendDate < millis ()) {
    message.id = 0x542 ;
    message.len = 8 ;
    message.data [0] = 0 ;
    message.data [1] = 1 ;
    message.data [2] = 2 ;
    message.data [3] = 3 ;
    message.data [4] = 4 ;
    message.data [5] = 5 ;
    message.data [6] = 6 ;
    message.data [7] = 7 ;
```

```
    const bool ok = can.tryToSendReturnStatus (message) ;
    if (ok) {
      digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
      gSendDate += 1000 ;
      gSentCount += 1 ;
      Serial.print ("Sent: ") ;
      Serial.println (gSentCount) ;
    }
  }
  if (can.receive0 (message)) {
    gReceivedCount += 1 ;
    Serial.print ("Received: ") ;
    Serial.println (gReceivedCount) ;
  }
}
```

# 6 The CANMessage class

**Note.** The CANMessage class is declared in the CANMessage.h header file. The class declaration is protected by an include guard that causes the macro GENERIC_CAN_MESSAGE_DEFINED to be defined. The ACAN2515 driver[4], the ACAN2517 driver[5] and the ACAN2517FD driver[6] contain an identical CANMessage.h header file, enabling using the ACAN_STM32 driver, the ACAN2515 driver, ACAN2517 driver and ACAN2517FD driver in a same sketch.

A *CAN message* is an object that contains all CAN 2.0B frame user informations. All properties are initialized by default, and represent a base data frame, with an identifier equal to $0$, and without any data.

```
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] ;
```

[4]The ACAN2515 driver is a CAN driver for the MCP2515 CAN controller, https://github.com/pierremolinaro/acan2515.
[5]The ACAN2517 driver is a CAN driver for the MCP2517FD CAN controller in CAN 2.0B mode, https://github.com/pierremolinaro/acan2517.
[6]The ACAN2517FD driver is a CANFD driver for the MCP2517FD CAN controller in CANFD mode, https://github.com/pierremolinaro/acan2517FD.

```
    uint8_t  data      [8] = {0, 0, 0, 0, 0, 0, 0, 0} ;
  } ;
} ;
```

Note the message datas are defined by an **union**. So message datas can be seen as height bytes, four 16-bit unsigned integers, two 32-bit, one 64-bit or two 32-bit floats. Be aware that multi-byte integers and floats are subject to endianness (Cortex M4 processors of the STM32 are little-endian).

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

- for a received message, it contains the acceptance filter index (see section 13 page 18);
- on sending messages, it is used for selecting the transmit buffer (see section 10 page 9).

# 7 Transmit FIFO

The STM32 `bxCAN` module provides 3 *mailboxes* for sending CAN frames. Each mailbox has a capacity of 1 message. `MailBox 0` gets the message from the *driver transmit FIFO*, for `MailBox 1` and `MailBox 2` see section 8 page 9.

The transmit FIFO (see figure 1 page 4) size is 20 by default; you can change the default size by setting the `mDriverTransmitFIFOSize` property of your `settings` object.

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

- if the *controller transmit FIFO* is not full, the message is appended to it, and `tryToSendReturnStatus` returns `0`;
- otherwise, if the *driver transmit FIFO* is not full, the message is appended to it, and `tryToSendReturnStatus` 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 `tryToSendReturnStatus` returns the `kTransmitBufferOverflow` error.

The transmit FIFO ensures sequentiality of emission.

## 7.1 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 = can.driverTransmitFIFOSize () ;
```

### 7.2 The `driverTransmitFIFOCount` method

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

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

### 7.3 The `driverTransmitFIFOPeakCount` method

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

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

If the transmit FIFO is full when `tryToSendReturnStatus` 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 `tryToSendReturnStatus` do not provide any overflow of the driver transmit FIFO.

## 8 Transmit mailboxes (`MailBox1` and `MailBox2`)

A *Mailbox* has a capacity of 1 message. So it is either empty, either full. You can call the `sendBufferNotFullForIndex` method (section 10.1 page 10) for testing if a mailbox is empty or full.

## 9 Receive FIFOs

A CAN module contains two receive FIFOs, `FIFO0` and `FIFO1`. **By default, only `FIFO0` is enabled.**

the receive FIFO$i$ ($0 \leqslant i \leqslant 1$, see figure 1 page 4) is composed by:

- the *hardware receive FIFO$i$*, whose size is always 3;
- the *driver receive FIFO$i$* (in library software), whose size is positive (default 32 for `FIFO0`, 0 for `FIFO1`); you can change the default size by setting the `mDriverReceiveFIFO`$i$`Size` property of your `settings` object.

The receive FIFO mechanism ensures sequentiality of reception.

## 10 Sending frames: the `tryToSendReturnStatus` method

The `ACAN_STM32::tryToSendReturnStatus` method sends CAN 2.0B frames:

```
uint32_t ACAN_STM32::tryToSendReturnStatus (const CANMessage & inMessage);
```

You call the `tryToSendReturnStatus` 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:

- `kTransmitBufferIndexTooLarge` (value: 1) if the `idx` property value does not specify a valid transmit mailbox (see below);
- `kTransmitBufferOverflow` (value: 2) 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 7 page 8);
- 1 for the *mailbox 1* (section 8 page 9);
- 2 for the *mailbox 2* (section 8 page 9).

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

```
bool ACAN_STM32::sendBufferNotFullForIndex (const uint32_t inTxBufferIndex);
```

This method returns `true` if the corresponding transmit buffer is not full, and `false` otherwise (table 3).

| **inTxBufferIndex** | **Operation** |
|---:|---|
| `0` | `true` if the transmit FIFO is not full, and `false` otherwise |
| 1 | `true` if the Mailbox1 is empty, and `false` if it is full |
| 2 | `true` if the Mailbox2 is empty, and `false` if it is full |
| > 2 | Always `false` |

**Table 3** – Value returned by the `sendBufferNotFullForIndex` method

## 10.2 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 ()) {
    CANMessage message ;
    // Initialize message properties
```

```
    const uint32_t sendStatus = can.tryToSendReturnStatus (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 = can.tryToSendReturnStatus (message) ;
    if (sendStatus == 0) {
      gSendMessage = false ;
    }
  }
  ...
}
```

## 11 Retrieving received messages using the `receive`$i$ method

```
bool ACAN_STM32::receive0 (CANMessage & outMessage) ;
bool ACAN_STM32::receive1 (CANMessage & 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 () {
  CANMessage message ;
  if (can.receive0 (message)) {
    // Handle received message
  }
  ...
}
```

The `receive` method:

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

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 () {
  CANMessage message ;
  if (can.receive0 (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 CANMessage & inMessage) {
  ...
}
```

So are the header of the `handle_myMessage_1` and the `handle_myMessage_2` functions.

## 11.1 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:

```
ACAN_STM32_Settings settings (125 * 1000) ;
settings.mDriverReceiveFIFO0Size = 100 ;
const uint32_t errorCode = can.begin (settings) ;
...
```

As the size of `CANMessage` class is 16 bytes, the actual size of the driver receive FIFO 0 is the value of `settings.mDriverReceiveFIFO0Size * 16`, and the actual size of the driver receive FIFO 1 is the value of `settings.mDriverReceiveFIFO1Size * 16`.

### 11.2 The driverReceiveFIFO$i$Size method

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

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

### 11.3 The driverReceiveFIFO$i$Count method

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

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

### 11.4 The driverReceiveFIFO$i$PeakCount method

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

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

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

### 11.5 The resetDriverReceiveFIFO$i$PeakCount method

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

```
can.resetDriverReceiveFIFO0PeakCount () ;
```

## 12 Acceptance filters

**for an example sketch, see LoopBackDemoFilters.**

Due to bxCAN features, you can define up to 14 filter banks. A filter bank can be either:

- a standard quad filter bank (section 12.1 page 14);
- a standard mask dual filter bank (section 12.2 page 15);
- an extended dual filter bank (section 12.3 page 16);
- an extended mask single filter bank (section 12.4 page 17).

The bxCAN module bases the filtering of the received frames on the nature of their identifier value, format (standard / extended, data / remote).

## 12.1 Quad standard filter bank

```
bool Filters::addStandardQuad (const uint16_t inIdentifier1,
                               const bool inRTR1,
                               const uint16_t inIdentifier2,
                               const bool inRTR2,
                               const uint16_t inIdentifier3,
                               const bool inRTR3,
                               const uint16_t inIdentifier4,
                               const bool inRTR4,
                               const ACAN_STM32::Action inAction) ;

bool Filters::addStandardQuad (const uint16_t inIdentifier1,
                               const bool inRTR1,
                               const ACANCallBackRoutine inCallBack1,
                               const uint16_t inIdentifier2,
                               const bool inRTR2,
                               const ACANCallBackRoutine inCallBack2,
                               const uint16_t inIdentifier3,
                               const bool inRTR3,
                               const ACANCallBackRoutine inCallBack3,
                               const uint16_t inIdentifier4,
                               const bool inRTR4,
                               const ACANCallBackRoutine inCallBack4,
                               const ACAN_STM32::Action inAction) ;
```

This bank defines 4 individual independant filters; one of theses filters matches if all following conditions are met:

- the received frame is a standard frame;
- the received frame identifier is equal to `inIdentifier`$i$ value;
- if `inRTR`$i$ is `false`, the received frame is a data frame;
- if `inRTR`$i$ is `true`, the received frame is a remote frame.

If the received frame matches one of theses filters, it is appended to Hardware receive FIFO0 or Hardware receive FIFO1, depending from `inAction` value.

Therefore this bank is valid if all `inIdentifier`$i$ are lower or equal to `0x7FF`. The method returns `true` if the bank is valid, and `false` otherwise. If the bank is valid, this method appends it to the receiver list. If the bank is not valid, the bank is not appended.

A `inCallBack`$i$ callback function can be associated to each individual filter. The first prototype does not name any callback function, it is equivalent to the second one when all `inCallBack`$i$ are `nullptr`.

See section 13 page 18 for using callback routines.

## 12.2 Dual standard mask filter bank

```
bool Filters::addStandardMasks (const uint16_t inBase1,
                                const uint16_t inMask1,
                                const Format inFormat1,
                                const uint16_t inIdentifier2,
                                const uint16_t inMask2,
                                const Format inFormat2,
                                const ACAN_STM32::Action inAction) ;

bool Filters::addStandardMasks (const uint16_t inBase1,
                                const uint16_t inMask1,
                                const Format inFormat1,
                                const ACANCallBackRoutine inCallBack1,
                                const uint16_t inIdentifier2,
                                const uint16_t inMask2,
                                const Format inFormat2,
                                const ACANCallBackRoutine inCallBack2,
                                const ACAN_STM32::Action inAction) ;
```

This bank defines 2 individual independant filters; one of theses filters matches if all following conditions are met:

- the received frame is a standard frame;
- the received frame identifier verifies (*received_frame_identifier* & `inMask`$i$) is equal to `inBase`$i$;
- if `inFormat`$i$ is equal to `ACAN_STM32::DATA`, the received frame is a data frame;
- if `inFormat`$i$ is equal to `ACAN_STM32::REMOTE`, the received frame is a remote frame;
- if `inFormat`$i$ is equal to `ACAN_STM32::DATA_OR_REMOTE`, the received frame can be a data frame or a remote frame.

If the received frame matches one of theses filters, it is appended to Hardware receive FIFO0 or Hardware receive FIFO1, depending from `inAction` value.

This bank is valid if for all $i$:

- (`inBase`$i$ is lower or equal to `0x7FF`;
- (`inMask`$i$ is lower or equal to `0x7FF`;
- (`inBase`$i$ & `inMask`$i$) is equal to `inBase`$i$.

The method returns `true` if the bank is valid, and `false` otherwise. If the bank is valid, this method appends it to the receiver list. If the bank is not valid, the bank is not appended.

A `inCallBack`$i$ callback function can be associated to each individual filter. The first prototype does not name any callback function, it is equivalent to the second one when all `inCallBack`$i$ are `nullptr`.

See section 13 page 18 for using callback routines.

For example:

```
filters.addStandardMasks (0x405, 0x7D5, ACAN_STM32::DATA, // 8 Standard data frames
                          0x605, 0x7D5, ACAN_STM32::REMOTE, // 8 Standard remote frames
                          ACAN_STM32::FIFO1) ;
```

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

| | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| inBase: 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`.

## 12.3 Dual extended filter bank

```
bool Filters::addExtendedDual (const uint32_t inIdentifier1,
                               const bool inRTR1,
                               const uint32_t inIdentifier2,
                               const bool inRTR2,
                               const ACAN_STM32::Action inAction) ;

bool Filters::addExtendedDual (const uint32_t inIdentifier1,
                               const bool inRTR1,
                               const ACANCallBackRoutine inCallBack1,
                               const uint32_t inIdentifier2,
                               const bool inRTR2,
                               const ACANCallBackRoutine inCallBack2,
                               const ACAN_STM32::Action inAction) ;
```

This bank defines 2 individual independant filters; one of theses filters matches if all following conditions are met:

- the received frame is a extended frame;
- the received frame identifier is equal to `inIdentifier`$i$ value;
- if `inRTR`$i$ is `false`, the received frame is a data frame;
- if `inRTR`$i$ is `true`, the received frame is a remote frame.

If the received frame matches one of theses filters, it is appended to Hardware receive FIFO0 or Hardware receive FIFO1, depending from `inAction` value.

Therefore this bank is valid if all `inIdentifier`$i$ are lower or equal to `0x1FFF_FFFF`. The method returns `true` if the bank is valid, and `false` otherwise. If the bank is valid, this method appends it to the receiver list. If the bank is not valid, the bank is not appended.

A `inCallBack`$i$ callback function can be associated to each individual filter. The first prototype does not name any callback function, it is equivalent to the second one when all `inCallBack`$i$ are `nullptr`.

See section 13 page 18 for using callback routines.

## 12.4 Single extended mask filter bank

```
bool Filters::addExtendedMask (const uint32_t inBase,
                               const uint32_t inMask,
                               const Format inFormat,
                               const ACAN_STM32::Action inAction) ;

bool Filters::addExtendedMask (const uint32_t inBase,
                               const uint32_t inMask,
                               const Format inFormat,
                               const ACANCallBackRoutine inCallBack,
                               const ACAN_STM32::Action inAction) ;
```

This bank defines one individual filter that matches if all following conditions are met:

- the received frame is an extended frame;
- the received frame identifier verifies (*received_frame_identifier* & `inMask`) is equal to `inBase`;
- if `inFormat` is equal to `ACAN_STM32::DATA`, the received frame is a data frame;
- if `inFormat` is equal to `ACAN_STM32::REMOTE`, the received frame is a remote frame;
- if `inFormat` is equal to `ACAN_STM32::DATA_OR_REMOTE`, the received frame can be a data frame or a remote frame.

If the received frame matches this filter, it is appended to Hardware receive FIFO0 or Hardware receive FIFO1, depending from `inAction` value.

This bank is valid if :

- (`inBase` is lower or equal to `0x1FFF_FFFF`;
- (`inMask` is lower or equal to `0x1FFF_FFFF`;
- (`inBase` & `inMask`$i$) is equal to `inBase`.

The method returns `true` if the bank is valid, and `false` otherwise. If the bank is valid, this method appends it to the receiver list. If the bank is not valid, the bank is not appended.

A `inCallBack` callback function can be associated to the filter. The first prototype does not name any callback function, it is equivalent to the second one when `inCallBack` is `nullptr`.

See section 13 page 18 for using callback routines.

For example:

```
filters.addExtendedMask (0x6789, 0x1FFF67BD, ACAN_STM32::REMOTE, ACAN_STM32::FIFO0) ;
```

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

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

Therefore there are 32 matching extended remote frames.

# 13 The `dispatchReceivedMessage` method

**Sample sketch:** the `LoopBackDemoDispatch` sketch shows how using the `dispatchReceivedMessage` method.

Instead of calling the `receive0` and the `receive1` methods, call the `dispatchReceivedMessage` method in your `loop` function. For every message extracted from `FIFO0` and `FIFO1`, it calls the callback function associated with the corresponding filter.

If you have not defined any filter, do not use this function, call the `receive0` and / or the `receive1` methods.

```
void loop () {
  can.dispatchReceivedMessage () ; // Do not call can.receive0, can.receive1 any more
  ...
}
```

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

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

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

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

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

## 14 The dispatchReceivedMessage0 method

The `dispatchReceivedMessage0` method dispatches the messages stored in the `FIFO0`. The messages stored is `FIFO1` are retrieved using the `receive1` method.

```
void loop () {
  can.dispatchReceivedMessage0 () ; // Do not call can.receive0 any more
  CANMessage ;
  if (can.receive1 (message)) {
    ... handle FIFO1 message ...
  }
  ...
}
```

Instead of calling the `receive0` method, call the `dispatchReceivedMessage0` 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 `receive0` method.

The `dispatchReceivedMessage0` method handles one `FIFO0` message on each call. Specifically:

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

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

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

```
void loop () {
  while (can.dispatchReceivedMessage0 ()) {
  }
  CANMessage ;
  if (can.receive1 (message)) {
    ... handle FIFO1 message ...
  }
  ...
}
```

## 15 The dispatchReceivedMessage1 method

The `dispatchReceivedMessage1` method dispatches the messages stored in the `FIFO1`. The messages stored is `FIFO0` are retrieved using the `receive0` method.

```
void loop () {
```

```
  can.dispatchReceivedMessage1 () ; // Do not call can.receive1 any more
  CANMessage ;
  if (can.receive0 (message)) {
    ... handle FIFO0 message ...
  }
  ...
}
```

Instead of calling the receive1 method, call the dispatchReceivedMessage1 method in your loop function. For every message extracted from FIFO1, it calls the callback function associated with the corresponding filter.

If you have not defined any filter that targets the FIFO1, do not use this function (messages will be not dispatched and therefore lost), call the receive1 method.

The dispatchReceivedMessage1 method handles one FIFO1 message on each call. Specifically:

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

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

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

```
void loop () {
  while (can.dispatchReceivedMessage1 ()) {
  }
  CANMessage ;
  if (can.receive0 (message)) {
    ... handle FIFO0 message ...
  }
  ...
}
```

# 16 The ACAN_STM32::begin method reference

## 16.1 The prototype

```
uint32_t ACAN_STM32::begin (const ACAN_STM32_Settings & inSettings,
                            const ACAN_STM32::Filters & inFilters = ACAN_STM32::Filters ()) ;
```

The first argument is a ACAN_STM32_Settings instance that defines the settings.

The second one is optional, and specifies the filter bank list (see section 12 page 13). By default, the filter bank list is empty.

### 16.2 The error codes

The ACAN_STM32::begin method returns an error code. The value 0 denotes no error. Otherwise, you consider every bit as an error flag, as described in table 4. An error code could report several errors. The ACAN_STM32 class defines static constants for naming errors. Bits 0 to 16 denote a bit configuration error, see table 5 page 25.

| Bit | Code | Static constant Name | Comment |
|---|---|---|---|
| 0 | 0x1 | kBitRatePrescalerIsZero | See table 5 page 25 |
| … | … | …. | See table 5 page 25 |
| 9 | 0x200 | kDataSJWIsGreaterThanPhaseSegment2 | See table 5 page 25 |

**Table 4** – The ACAN_STM32::begin method error code bits

## 17 ACAN_STM32_Settings class reference

**Note.** The ACAN_STM32_Settings class is not Arduino specific. You can compile it on your desktop computer with your favorite C++ compiler.

### 17.1 The ACAN_STM32_Settings constructor: computation of the CAN bit settings

```
void setup () {
ACAN_STM32_Settings::ACAN_STM32_Settings (const uint32_t inDesiredBitRate,
                                          const uint32_t inTolerancePPM = 1000) ;
```

The constructor of the ACAN_STM32_Settings has two mandatory arguments: the desired bit rate. It tries to compute the CAN bit settings for theses bit rates. If it succeeds, the constructed object has its mBitRateClosedToDesiredRate property set to true, otherwise it is set to false. For example, for an 1 Mbit/s bit rate:

```
void setup () {
 // Bit rate: 1 Mbit/s
  ACAN_STM32_Settings settings (1000 * 1000) ;
  // Here, settings.mBitRateClosedToDesiredRate is true
  ...
}
```

But this does not mean there is no possibility to get such data bit rates factors. For example, we can have a bit rate of 4/7 Mbit/s = 571 428 kbit/s with the STM32L432KC clock (80 MHz):

```
void setup () {
  ...
  ACAN_STM32_Settings settings (571428) ;
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (true)
  Serial.print ("Actual Bit Rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  571428 bit/s
  Serial.print ("distance: ") ;
```

```
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 1 ppm= 0,0001 %
  ...
}
```

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

By default, a desired bit rate is accepted if the distance from the computed actual bit rate is lower or equal to $1,000$ ppm $= 0.1$ %. You can change this default value by adding your own value as third argument of ACAN_STM32_Settings constructor. For example, with a bit rate equal to 727 kbit/s (with the STM32L432KC clock, 80 MHz):

```
void setup () {
  ...
  ACAN_STM32_Settings settings (727 * 1000, 100) ; // 100 ppm
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; // 727272 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 375 ppm
  ...
}
```

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

```
void setup () {
  ...
  ACAN_STM32_Settings settings (500 * 1000, 0) ; // Max distance is 0 ppm
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (true)
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 0 ppm
  ...
}
```

In any way, the bit rate computation always gives a consistent result, resulting an actual bit rate closest from the desired bit rate. For example, we query a 330 kbit/s bit rate (with the STM32L432KC clock, 80 MHz):

```
void setup () {
  ...
  ACAN_STM32_Settings settings (330 * 1000) ;
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (false)
  Serial.print ("Actual Bit Rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  330 578 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 1 753 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 () {
  ...
  ACAN_STM32_Settings settings (330 * 1000) ;
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (false)
  Serial.print ("Actual Bit Rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  330 578 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 1 753 ppm
  Serial.print ("Bit rate prescaler: ") ;
  Serial.println (settings.mBitRatePrescaler) ; // BRP = 11
  Serial.print ("Phase segment 1: ") ;
  Serial.println (settings.mPhaseSegment1) ; // PS1 = 14
  Serial.print ("Phase segment 2: ") ;
  Serial.println (settings.mPhaseSegment2) ; // PS2 = 7
  Serial.print ("Resynchronization Jump Width: ") ;
  Serial.println (settings.mRJW) ; // RJW = 4
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 68, meaning 68%
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

The `samplePointFromBitStart` method returns sample point, expressed in per-cent of the bit duration from the beginning of the bit.

Note the computation may calculate a bit decomposition too far from the desired bit rate, but it is always consistent. You can check this by calling the `CANBitSettingConsistency` method.

You can change the property values for adapting to the particularities of your CAN network propagation time. By example, you can increment the `mPhaseSegment1` property value, and decrement the `mPhaseSegment2` property value in order to sample the RxCAN pin later.

```
void setup () {
  ...
  ACAN_STM32_Settings settings (500 * 1000) ;
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (true)
  settings.mPhaseSegment1 += 1 ; // 13 -> 14: safe, 1 <= PS1 <= 16
  settings.mPhaseSegment2 -= 1 ; // 6 -> 5: safe, 1 <= PS2 <= 8, and PS2 >= RJW
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 75, meaning 75%
```

```
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; // 500 000: ok, no change
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

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

$$
\begin{aligned}
1 &\leqslant \texttt{mBitRatePrescaler} \leqslant 1024 \\
1 &\leqslant \texttt{mPhaseSegment1} \leqslant 16 \\
2 &\leqslant \texttt{mPhaseSegment2} \leqslant 18 \\
1 &\leqslant \texttt{mRJW} \leqslant 4 \\
&\texttt{mRJW} \leqslant \texttt{mPhaseSegment2}
\end{aligned}
$$

Resulting actual bit rates are given by (CAN_CLOCK_FREQUENCY is defined in table 2 page 3):

$$\text{Actual Bit Rate} = \frac{\texttt{CAN\_CLOCK\_FREQUENCY}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2})}$$

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

$$\text{Sampling Point} = 100 \cdot \frac{1 + \texttt{mPhaseSegment1}}{1 + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2}}$$

## 17.2 The CANBitSettingConsistency method

This method checks the CAN bit decomposition (specified by mBitRatePrescaler, mPhaseSegment1, mPhase-Segment2, mRJW and mTripleSampling property values) is consistent.

```
void setup () {
  ...
  ACAN_STM32_Settings settings (500 * 1000) ;
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (true)
  settings.mPhaseSegment1 = 0 ; // Error, mPhaseSegment1 should be >= 1 (and <= 16)
  Serial.print ("Consistency: 0x") ;
  Serial.println (settings.CANBitSettingConsistency (), HEX) ; // != 0, meaning error
  ...
}
```

The `CANBitSettingConsistency` method returns 0 if CAN bit decomposition is consistent. Otherwise, the returned value is a bit field that can report several errors – see table 5.

The `ACAN_STM32_Settings` class defines static constant properties that can be used as mask error. For example:

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

| Bit | Code | Error Name | Error |
|---|---|---|---|
| 0 | 0x1 | kBitRatePrescalerIsZero | mBitRatePrescaler == 0 |
| 1 | 0x2 | kBitRatePrescalerIsGreaterThan1024 | mBitRatePrescaler > 1024 |
| 2 | 0x4 | kPhaseSegment1IsZero | mPhaseSegment1 == 0 |
| 3 | 0x8 | kPhaseSegment1IsGreaterThan16 | mPhaseSegment1 > 16 |
| 4 | 0x10 | kPhaseSegment2IsZero | mPhaseSegment2 == 0 |
| 5 | 0x20 | kPhaseSegment2IsGreaterThan8 | mPhaseSegment2 > 8 |
| 6 | 0x40 | kRJWIsZero | mRJW == 0 |
| 7 | 0x80 | kRJWIsGreaterThan4 | mRJW > 4 |
| 8 | 0x100 | kRJWIsGreaterThanPhaseSegment2 | mRJW > mPhaseSegment2 |
| 9 | 0x200 | kPhaseSegment1Is1AndTripleSampling | (mPhaseSegment1 == 1) and triple sampling |

**Table 5** – The `ACAN_STM32_Settings::CANBitSettingConsistency` method error codes

## 17.3 The actualBitRate method

The `actualBitRate` method returns the actual bit computed from `mBitRatePrescaler`, `mPropagationSegment`, `mPhaseSegment1`, `mPhaseSegment2`, `mRJW` property values.

```
void setup () {
  ...
  ACAN_STM32_Settings settings (440 * 1000) ;
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  ...
}
```

**Note.** If CAN bit settings are not consistent (see section 17.2 page 24), the returned value is irrelevant.

## 17.4 The exactBitRate method

```
bool ACAN_STM32_Settings::exactBitRate (void) const ;
```

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

**Note.** If CAN bit settings are not consistent (see section 17.2 page 24), the returned value is irrelevant.

## 17.5 The exactDataBitRate method

```
bool ACAN_STM32_Settings::exactDataBitRate (void) const ;
```

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

**Note.** If CAN bit settings are not consistent (see section 17.2 page 24), the returned value is irrelevant.

## 17.6 The ppmFromDesiredBitRate method

```
uint32_t ACAN_STM32_Settings::ppmFromDesiredBitRate (void) const ;
```

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

**Note.** If CAN bit settings are not consistent (see section 17.2 page 24), the returned value is irrelevant.

## 17.7 The ppmFromDesiredDataBitRate method

```
uint32_t ACAN_STM32_Settings::ppmFromDesiredDataBitRate (void) const ;
```

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

**Note.** If CAN bit settings are not consistent (see section 17.2 page 24), the returned value is irrelevant.

## 17.8 The samplePointFromBitStart method

```
uint32_t ACAN_STM32_Settings::samplePointFromBitStart (void) const ;
```

The `samplePointFromBitStart` method returns the distance of sample point from the start of the 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 17.2 page 24), the returned value is irrelevant.

## 17.9 The dataSamplePointFromBitStart method

```
uint32_t ACAN_STM32_Settings::dataSamplePointFromBitStart (void) const ;
```

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

**Note.** If CAN bit settings are not consistent (see section 17.2 page 24), the returned value is irrelevant.

## 17.10 Properties of the ACAN_STM32_Settings class

All properties of the ACAN_STM32_Settings class are declared public and are initialized (table 6).

| Property | Type | Initial value | Comment |
|---|---|---|---|
| mDesiredBitRate | uint32_t | Constructor argument | |
| mBitRatePrescaler | uint8_t | 1024 | See section 17.1 page 21 |
| mPhaseSegment1 | uint16_t | 16 | See section 17.1 page 21 |
| mPhaseSegment2 | uint8_t | 8 | See section 17.1 page 21 |
| mRJW | uint8_t | 4 | See section 17.1 page 21 |
| mTripleSampling | bool | true | See section 17.1 page 21 |
| mBitSettingOk | bool | true | See section 17.1 page 21 |
| mModuleMode | ModuleMode | NORMAL | See section 17.10.1 page 27 |
| mOpenCollectorOutput | bool | false | See section 17.10.2 page 27 |
| mDriverReceiveFIFO0Size | uint16_t | 32 | See section 11.1 page 12 |
| mDriverReceiveFIFO1Size | uint16_t | 0 | See section 11.1 page 12 |
| mDriverTransmitFIFOSize | uint16_t | 8 | See section 7 page 8 |

**Table 6** – Properties of the ACAN_STM32_Settings class

### 17.10.1 The mModuleMode property

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

### 17.10.2 The mOpenCollectorOutput property

By default, mOpenCollectorOutput property is false, therefore TxCAN pin is 2-state push / pull pin. If mOpenCollectorOutput property is set to true, TxCAN pin is an open collector pin.