ACAN library
for Teensy 3.1 / 3.2, 3.5, 3.6
Version 2.0.3

Pierre Molinaro

October 1, 2021

Contents

1 Versions

Version	Date	Comment
2.0.3	October 1, 2021	Added data_s64, data_s32, data_s16 and data_s8 to CANMessage class union members, see section 5 page 7 (thanks to tomtom0707).
2.0.2	April 27, 2020	Added dataFloat to CANMessage (thanks to Koryphon) Added several forgotten volatile
2.0.1	March 6, 2020	Fixed broken sequentiality (thanks to wangnick).
2.0.0	February 21, 2019	Updated documentation on error codes Renamed error codes
1.0.6	October 23, 2018	Compatibility with ACAN2515 version 1.0.1
1.0.5	October 12, 2018	Corrected interrupt masking, some messages to send were lost in previous releases
1.0.4	October 12, 2018	Adding include guard in CANmessage.h header file, for compatibility with ACAN2515 library
1.0.1	December 11, 2017	Added mTxPinIsOpenCollector (see section 16.7.5 page 35) and mRxPinHasInternalPullUp settings (section 16.7.6 page 35)
1.0.0	October 9, 2017	Initial release

2 Features

The ACAN library is a CAN ("Controller Area Network") driver for Teensy 3.1 / 3.2, 3.5, 3.6. 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 user bit rate;
- user can fully define its own CAN bit setting values;
- reception filters are easily defined – up to 14 primary filters and 18 secondary filters;
- reception filters accept call back functions;
- driver transmit buffer size is customisable;
- driver receive buffer size is customisable;

- overflow of the driver receive buffer is detectable;
- *loop back*, *self reception*, *listing only* FlexCAN controller modes are selectable;
- `Tx` pin can be configured as open collector;
- internal pullup can be enabled for `Rx` pin.

3 Data flow

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

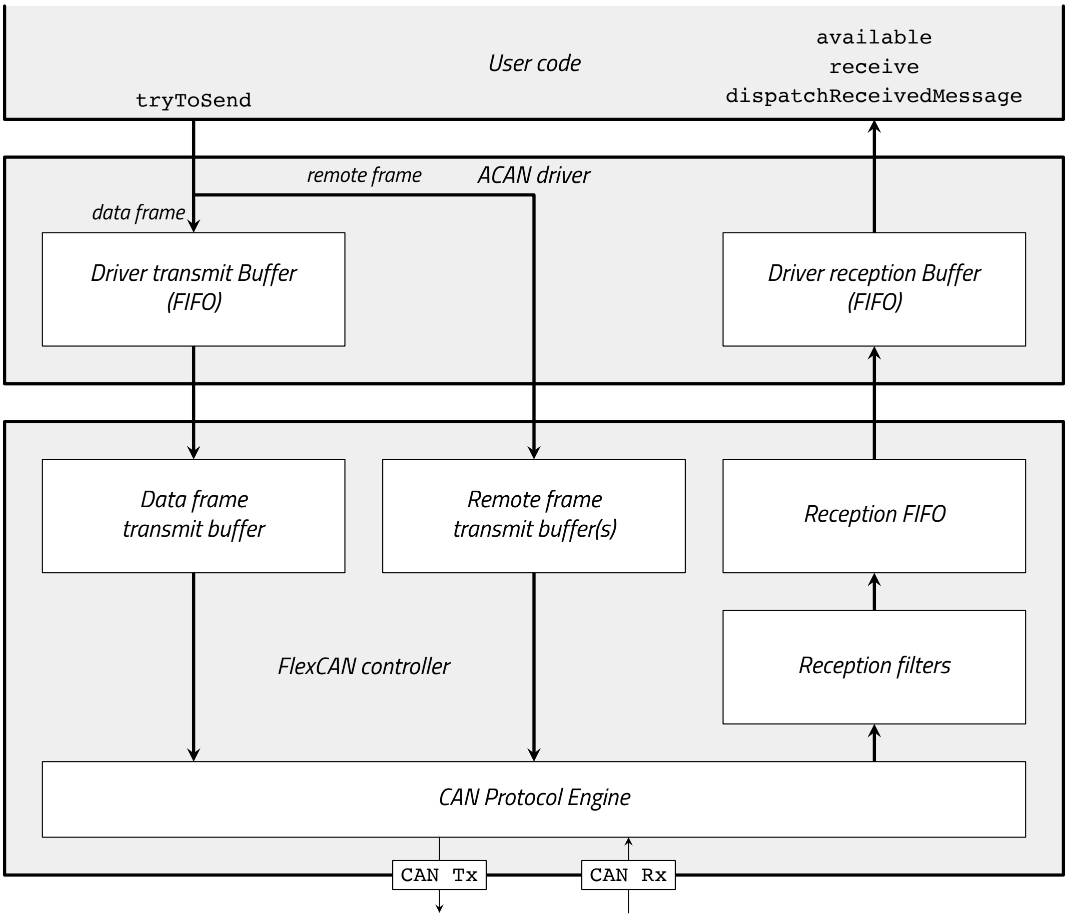

Figure 1 – Message flow in `ACAN` driver and FlexCAN controller

FlexCAN controller is hardware, a module of the micro-controller. It implements 16 MBs (*Mailboxes* or *Message Buffers*), used for the *data frame transmit buffer*, *remote frame transmit buffer(s)*, *reception FIFO* and *reception filters*. The actual partition depends from the selected configuration – see table 1 and section 11 page 14.

Sending messages. The FlexCAN hardware makes sending data frames different from sending remote frames. For both, user code calls the `tryToSend` method – see section 8 page 9 for sending data frames, and section 9 page 11 for sending remote frames. The data frames are stored in the *Driver Transmit Buffer*, before to

settings.mConfiguration	Reception	Sending remote frames	Sending data frames
`k8_0_Filters`	8 (MB0 … MB7)	7 (MB8 … MB14)	1 (MB15)
`k10_6_Filters`	10 (MB0 … MB9)	5 (MB10 … MB14)	1 (MB15)
`k12_12_Filters`	12 (MB0 … MB11)	3 (MB12 … MB14)	1 (MB15)
`k14_18_Filters`	14 (MB0 … MB13)	1 (MB14)	1 (MB15)

Table 1 – FlexCAN MBs assignments, following `settings.mConfiguration` value

be moved by the message interrupt service routine into the *data frame transmit buffer*. The size of the *Driver Transmit Buffer* is 16 by default – see section 8.2 page 10 for changing the default value.

Receiving messages. The FlexCAN *CAN Protocol Engine* transmits all correct frames to the *reception filters*. By default, they are configured as pass-all, see section 12 page 14 and section 13 page 18 for configuring them. Messages that pass the filters are stored in the *Reception FIFO*. Its depth is not configurable – it is always 6-message. The message interrupt service routine transfers the messages from *Reception FIFO* to the *Driver Receive Buffer*. The size of the *Driver Receive Buffer* is 32 by default – see section 10.1 page 13 for changing the default value. Three user methods are available:

- the `available` method returns `false` if the *Driver Receive Buffer* is empty, and `true` otherwise;
- the `receive` method retrieves messages from the *Driver Receive Buffer* – see section 10 page 12, section 12.5 page 18 and section 13.5 page 22;
- the `dispatchReceivedMessage` method if you have defined primary and / or secondary filters that name a call-back function – see section 14 page 23.

Sequentiality. The ACAN driver and the configuration of the FlexCAN controller ensures sequentiality of data messages. This means that if an user program calls `tryToSend` first for a message M_1 and then for a message M_2, the message M_1 will be always retrieved by `receive` or `dispatchReceivedMessage` before the message M_2.

4 A simple example: `LoopBackDemo`

The following code is a sample code for introducing the ACAN library. It runs on Teensy 3.1 / 3.2, Teensy 3.5 and Teensy 3.6. It demonstrates how to configure the driver, to send a CAN message, and to receive a CAN message.

Note it runs without any external hardware, it uses the *loop back* mode and the *self reception* mode.

```
#include <ACAN.h>

void setup () {
  Serial.begin (9600) ;
  Serial.println ("Hello") ;
  ACANSettings settings (125 * 1000) ; // 125 kbit/s
  settings.mLoopBackMode = true ;
```

```
8     settings.mSelfReceptionMode = true ;
9     const uint32_t errorCode = ACAN::can0.begin (settings) ;
10    if (0 == errorCode) {
11      Serial.println ("ACAN::can0 ok") ;
12    }else{
13      Serial.print ("Error ACAN::can0: 0x") ;
14      Serial.println (errorCode, HEX) ;
15    }
16  }
17
18  static uint32_t gSendDate = 0 ;
19  static uint32_t gSentCount = 0 ;
20  static uint32_t gReceivedCount = 0 ;
21
22  void loop () {
23    CANMessage message ;
24    if (gSendDate < millis ()) {
25      message.id = 0x542 ;
26      const bool ok = ACAN::can0.tryToSend (message) ;
27      if (ok) {
28        gSendDate += 2000 ;
29        gSentCount += 1 ;
30        Serial.print ("Sent: ") ;
31        Serial.println (gSentCount) ;
32      }
33    }
34    if (ACAN::can0.receive (message)) {
35      gReceivedCount += 1 ;
36      Serial.print ("Received: ") ;
37      Serial.println (gReceivedCount) ;
38    }
39  }
```

Line 1. This line includes the `ACAN` library.

Line 6. Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACANSettings` class. The constructor has one parameter: the wished CAN bit rate. It returns a `settings` object fully initialized with CAN bit settings for the wished bit rate, and default values for other configuration properties.

Lines 7 and 8. This is the second step. You can override the values of the properties of `settings` object. Here, the `mLoopBackMode` and `mSelfReceptionMode` properties are set to `true` – they are `false` by default. Theses two properties fully enables *loop back*, that is you can run this demo sketch even it you have no connection to a physical CAN network. The section 16.7 page 34 lists all properties you can override.

Line 9. This is the third step, configuration of the `ACAN::can0` driver with `settings` values. You cannot change the `ACAN::can0` name – see section 6 page 8. 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 14 and section 13 page 18.

Lines 10 to 15. Last step: the configuration of the `ACAN::can0` driver returns an error code, stored in the `errorCode` constant. It has the value 0 if all is ok – see section 15.2 page 25.

Line 18. The `gSendDate` global variable is used for sending a CAN message every 2 s.

Line 19. The `gSentCount` global variable counts the number of sent messages.

Line 20. The `gReceivedCount` global variable counts the number of received messages.

Line 23. The `message` object is fully initialized by the default constructor, it represents a standard data frame, with an identifier equal to 0, and without any data – see section 5 page 7.

Line 24. It tests if it is time to send a message.

Line 25. Set the message identifier. In a real code, we set here message data, and for an extended frame the `ext` boolean property.

Line 26. We try to send the data message. Actually, we try to transfer it into the *Driver transmit buffer*. The transfer succeeds if the buffer is not full. The `tryToSend` method returns `false` if the buffer is full, and `true` otherwise. Note the returned value only tells if the transfer into the *Driver transmit buffer* is successful or not: we have no way to know if the frame is actually sent on the the CAN network.

Lines 27 to 32. We act the successfull transfer by setting `gSendDate` to the next send date and incrementing the `gSentCount` variable. Note if the transfer did fail, the send date is not changed, so the `tryToSend` method will be called on the execution of the `loop` function.

Line 34. As the FlexCAN controller is configured in *loop back* mode (see lines 7 and 8), all sent messages are received. The `receive` method returns `false` if no message is available from the *driver reception buffer*. It returns `true` if a message has been successfully removed from the *driver reception buffer*. This message is assigned to the `message` object.

Lines 35 to 37. It a message has been received, the `gReceivedCount` is incremented and displayed.

5 The `CANMessage` class

Note. The `CANMessage` class is declared in the `CANMessage.h` header file. The class declaration is protected by an include guard that causes the macro `GENERIC_CAN_MESSAGE_DEFINED` to be defined. The ACAN2515 driver contains an identical `CANMessage.h` file header, enabling using both ACAN driver and ACAN2515 driver in a sketch.

A *CAN message* is an object that contains all CAN frame user informations. All properties are initialized by default, and represent a standard 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] ;
    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 Teensy 3.x 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 12.5 page 18 and section 13.5 page 22);
- it is not used on sending messages.

6 Driver instances

Driver instances are global variables. You cannot choose their names, they are defined by the library.

Teensy	Driver name
Teensy 3.1 / 3.2	`ACAN::can0`
Teensy 3.5	`ACAN::can0`
Teensy 3.6	`ACAN::can0`, `ACAN::can1`

Table 2 – Driver global variables

Code snippets in this document uses `ACAN::can0`. They also apply to `ACAN::can1` of Teensy 3.6.

Note. Drivers variables are `ACAN` class static properties. This choice may seem strange. However, a common error is to declare its own driver variable:

```
ACAN myCAN ; // Don't do that, it is an error !!!
```

Declaring drivers variables as `ACAN` class static properties[1] enables the compiler to raise an error if you try to declare your own driver variable.

[1] The `ACAN` constructor is declared `private`.

7 Alternate pins

For using alternate pins, just set `mUseAlternateTxPin` and / or `mUseAlternateRxPin` properties of `settings` object:

```
ACANSettings settings (125 * 1000) ;
settings.mUseAlternateRxPin = true ;
settings.mUseAlternateTxPin = true ;
const uint32_t errorCode = ACAN::can0.begin (settings) ;
```

By default, theses properties are set to `false`. The table 3 lists default and alternate pins. Note that `ACAN::can1` does not support alternate pins. Trying to set alternate pin for `ACAN::can1` raises error bits in the value returned by `begin` (see section 15 page 24).

Teensy	Driver name	Default Tx pin	Alternate Tx pin	Default Rx pin	Alternate Rx pin
Teensy 3.1 / 3.2	`ACAN::can0`	3	32	4	25
Teensy 3.5	`ACAN::can0`	3	29	4	30
Teensy 3.6	`ACAN::can0`	3	29	4	30
Teensy 3.6	`ACAN::can1`	33	No alternate Tx pin	34	No alternate Rx pin

Table 3 – Alternate CAN Tx and Rx pins

8 Sending data frames

Note. This section applies only to **data** frames. For sending remote frames, see section 9 page 11.

8.1 `tryToSend` for sending data frames

Call the method `tryToSend` for sending data frames; it returns:

- `true` if the message has been successfully transmitted to driver transmit buffer; note that does not mean that the CAN frame has been actually sent;
- `false` if the message has not been successfully transmitted to driver transmit buffer, it was full.

So it is wise to systematically test the returned value. One way to achieve this is to loop while there is no room in driver transmit buffer:

```
while (!ACAN::can0.tryToSend (message)) {
  yield () ;
}
```

A better way is to use a global variable to note if 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 () {
  CANMessage message ;
  if (gSendDate < millis ()) {
    // Initialize message properties
    const bool ok = ACAN::can0.tryToSend (message) ;
    if (ok) {
      gSendDate += 2000 ;
    }
  }
}
```

An other hint to use a global boolean variable as a flag that remains `true` while the frame 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 bool ok = ACAN::can0.tryToSend (message) ;
    if (ok) {
      gSendMessage = false ;
    }
  }
  ...
}
```

8.2 Driver transmit buffer size

By default, driver transmit buffer size is 16. You can change this default value by setting the `mTransmitBufferSize` property of `settings` variable:

```
ACANSettings settings (125 * 1000) ;
settings.mTransmitBufferSize = 30 ;
const uint32_t errorCode = ACAN::can0.begin (settings) ;
...
```

As the size of `CANMessage` class is 16 bytes, the actual size of the driver transmit buffer is the value of `settings.mTransmitBufferSize * 16`.

8.3 The `transmitBufferSize` method

It returns the size of the driver transmit buffer, that is the value of `settings.mTransmitBufferSize`.

```
const uint32_t s = ACAN::can0.transmitBufferSize () ;
```

8.4 The `transmitBufferCount` method

The `transmitBufferCount` method returns the current number of messages in the transmit buffer.

```
const uint32_t n = ACAN::can0.transmitBufferCount () ;
```

8.5 The `transmitBufferPeakCount` method

The `transmitBufferPeakCount` method returns the peak value of message count in the transmit buffer.

```
const uint32_t max = ACAN::can0.transmitBufferPeakCount () ;
```

Il the transmit buffer is full when `tryToSend` is called, the return value is `false`. In such case, the following calls of `transmitBufferPeakCount` will return `transmitBufferSize ()+1`.

So, when `transmitBufferPeakCount` returns a value lower or equal to `transmitBufferSize ()`, it means that calls to `tryToSend` have always returned `true`.

9 Sending remote frames

Note. This section applies only to **remote** frames. For sending data frames, see section 8 page 9.

The hardware design of the FlexCAN module makes sending remote frames different from data frames.

However, for sending remote frames, you also invoke the `tryToSend` method. This method understands if a remote frame should be sent, the `rtr` property of its argument is set (it is cleared by default, denoting a data frame).

```
CanMessage message ;
message.rtr = true ; // Remote frame
...
const bool sent = ACAN::can0.tryToSend (message) ;
...
```

The FlexCAN module embedded in Teensy 3.x microcontrollers implements 16 *mailboxes*, for sending and receiving CAN frames. Following the `settings.mConfiguration`, it allocates 7, 5, 3 or 1 MBs for sending remote frames, as indicating by the table 4 page 14. By default, `settings.mConfiguration` is set to `k12_12_Filters`, as remote frames are rarely needed.

10 Retrieving received messages using the `receive` method

There are two ways for retrieving received messages :

- using the `receive` method, as explained in this section;
- using the `dispatchReceivedMessage` method (see section 14 page 23).

This is a basic example:

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  ...
  const uint32_t errorCode = ACAN::can0.begin (settings) ; // No receive filter
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN::can0.receive (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 `rtr` bit (remote or data frame?), the `ext` bit (standard or extended frame), and the `id` (identifier value). The following snippet dispatches three messages:

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  ...
  const uint32_t errorCode = ACAN::can0.begin (settings) ; // No receive filter
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN::can0.receive (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) ;  // Standard data frame, id is 0x234
```

```
    }else if (message.rtr && !message.ext && (message.id == 0x542)) {
      handle_myMessage_2 (message) ;  // Standard 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.

10.1 Driver receive buffer size

By default, the driver receive buffer size is 32. You can change this default value by setting the `mReceiveBufferSize` property of `settings` variable:

```
ACANSettings settings (125 * 1000) ;
settings.mReceiveBufferSize = 100 ;
const uint32_t errorCode = ACAN::can0.begin (settings) ;
...
```

As the size of `CANMessage` class is 16 bytes, the actual size of the driver receive buffer is the value of `settings.mReceiveBuffe` `* 16`.

10.2 The `receiveBufferSize` method

The `receiveBufferSize` method returns the size of the driver receive buffer, that is the value of `settings.mReceiveBufferS`

```
const uint32_t s = ACAN::can0.receiveBufferSize () ;
```

10.3 The `receiveBufferCount` method

The `receiveBufferCount` method returns the current number of messages in the driver receive buffer.

```
const uint32_t n = ACAN::can0.receiveBufferCount () ;
```

10.4 The `receiveBufferPeakCount` method

The `receiveBufferPeakCount` method returns the peak value of message count in the driver receive buffer.

```
const uint32_t max = ACAN::can0.receiveBufferPeakCount () ;
```

Note the driver receive buffer may overflow, if messages are not retrieved (by calls of the `receive` method or the `dispatchReceivedMessage` method). If an overflow occurs, further calls of the `ACAN::can0.receiveBufferPeakCount ()` method return `ACAN::can0.receiveBufferSize ()+1`.

11 Configuration

The `mConfiguration` property of the `settings` variable defines the FlexCAN module configuration – see table 4. By default, its value is `ACANSettings::k12_12_Filters`.

You can easily override the default configuration:

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  settings.mConfiguration = ACANSettings::k14_18_Filters ;
  const uint32_t errorCode = ACAN::can0.begin (settings) ; // No receive filter
  ...
}
```

settings.mConfiguration	**Primary filters** section 12 page 14	**Secondary filters** section 13 page 18	**MB for sending remote frames** section 9 page 11
`k8_0_Filters`	8	0	7
`k10_6_Filters`	10	6	5
`k12_12_Filters`	12	12	3
`k14_18_Filters`	14	18	1

Table 4 – FlexCAN configuration, following `settings.mConfiguration` value

12 Primary filters

A first step is to define *receive filters*[2]. The *receive filters* are set to the FlexCAN module, so filtering is performed by hardware, without any CPU charge. The messages that pass the filters are transfered into the FlexCAN RxFIFO by the FlexCAN module, and transfered info the driver receive buffer by the driver. So the `receive` method only gets messages that have passed the filters.

The driver lets you to define two kinds of filters: *primary filters* and *secondary filters*[3]. Making the difference is required by FlexCAN hardware design: *primary filters* are more powerfull than *secondary filters*.

12.1 Primary filter example

For defining *primary filters*[4], you write:

[2] The second step is to use the `dispatchReceivedMessage` method instead of the `receive` method, see section 14 page 23.

[3] The *primary filters* and *secondary filters* terms are used in this document for simplicity. FlexCAN documentation names them respectively *Rx FIFO filter Table Elements Affected by Rx Individual Masks* and *Rx FIFO filter Table Elements Affected by Rx FIFO Global Mask*.

[4] For *secondary filters*, see section 13 page 18.

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456), // Filter #0
    ACANPrimaryFilter (kData, kStandard, 0x234),    // Filter #1
    ACANPrimaryFilter (kRemote, kStandard, 0x542)   // Filter #2
  } ;
  const uint32_t errorCode = ACAN::can0.begin (settings,
                                               primaryFilters, // The filter array
                                               3) ; // Filter array size
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN::can0.receive (message)) { // Only frames that pass a filter are retrieved
    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) ;  // Standard data frame, id is 0x234
    }else if (message.rtr && !message.ext && (message.id == 0x542)) {
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
    }
  }
  ...
}
```

Each element of the `primaryFilters` constant array defines an acceptance filter. Should be specified[5]:

- the required kind: data frames (`kData`) or remote frames (`kRemote`);
- the required format: standard frames (`kStandard`) or extended frames (`kExtended`);
- the required identifier value.

Maximum number of *primary filters*. The number of *primary filters* is limited: 12 by default, as the default value of `settings.mConfiguration` is `ACANSettings::k12_12_Filters`. See section 11 page 14 for getting the number of *primary filters* for each configuration, and for setting your own configuration.

Test order. The FlexCAN hardware examines the filters in the increasing order of their indexes in the `primaryFilters` constant array. As soon as a match occurs, the message is transfered to Rx FIFO buffer and the examination process is completed. If no match occurs, the message is lost.

A consequence is if a filter appears twice, the second occurrence will never match. In the next example, the Filter #3 will never match, as it is identical to filter #1.

```
void setup () {
```

[5] There is a fourth optional argument, that is `NULL` by default – see section 14 page 23.

```
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456), // Filter #0
    ACANPrimaryFilter (kData, kStandard, 0x234),    // Filter #1
    ACANPrimaryFilter (kRemote, kStandard, 0x542),  // Filter #2
    ACANPrimaryFilter (kData, kStandard, 0x234)     // Filter #3
  } ;
  ...
}
```

12.2 Primary filter as pass-all filter

You can specify a primary filter that matches any frame:

```
    ACANPrimaryFilter ()
```

You can use it for accepting all frames that did not match previous filters:

```
void setup () {
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456), // Filter #0
    ACANPrimaryFilter (kData, kStandard, 0x234),    // Filter #1
    ACANPrimaryFilter (kRemote, kStandard, 0x542),  // Filter #2
    ACANPrimaryFilter ()                            // Filter #3
  } ; // Filter #3 catches any message that did not match filters #0, #1 and #2
  ...
}
```

Be aware if the pass-all filter is not the last one, following ones will never match.

```
void setup () {
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456), // Filter #0
    ACANPrimaryFilter (kData, kStandard, 0x234),    // Filter #1
    ACANPrimaryFilter (),                           // Filter #2
    ACANPrimaryFilter (kRemote, kStandard, 0x542)   // Filter #3
  } ; // Filter #3 will never match
  ...
}
```

12.3 Primary filter for matching several identifiers

A primary filter can be configured for matching several identifiers[6]. You provide two values: a `filter_mask` and a `filter_acceptance`. A message with an `identifier` is accepted if:

[6] A *secondary filter* cannot be configured for matching several identifiers.

$$\texttt{filter\_mask} \,\&\, \texttt{identifier} = \texttt{filter\_acceptance}$$

The & operator is the bit-wise *and* operator.

Let's take an example: the filter should match standard data frames with identifiers equal to `0x540`, `0x541`, `0x542` and `0x543`. The four identifiers differs by the two lower bits. As a standard identifiers are 11-bits wide, the `filter_mask` is `0x7FC`. The filter acceptance is `0x540`. The filter is declared by:

```
  ...
  ACANPrimaryFilter (kData,      // Accept only data frames
                     kStandard, // Accept only standard frames
                     0x7FC,      // Filter mask
                     0x540)      // Filter acceptance
  ...
}
```

For a standard frame (11-bit identifier), both `filter_mask` and a `filter_acceptance` should be lower or equal to `0x7FF`.

For a extended frame (29-bit identifier), both `filter_mask` and a `filter_acceptance` should be lower or equal to `0x1FFF_FFFF`.

Be aware that the `filter_mask` and a `filter_acceptance` must also conform to the following constraint: if a bit is clear in the `filter_mask`, the corresponding bit of the `filter_acceptance` should also be clear. In other words, `filter_mask` and a `filter_acceptance` should check:

$$\texttt{filter\_mask} \,\&\, \texttt{filter\_acceptance} = \texttt{filter\_acceptance}$$

For example, the filter mask `0x7FC` and the filter acceptance `0x541` do not conform because the bit 0 of `filter_mask` is clear and the bit 0 of the filter acceptance is set.

A non conform filter may never match.

12.4 Primary filter conformance

The pass-all primary filter (section 12.2 page 16) always conforms.

For a primary filter for matching several identifiers, see section 12.3 page 16.

For a primary filter for one single identifier:

- for a standard frame (11-bit identifier), the given identifier value should be lower or equal to `0x7FF`;
- for a extended frame (29-bit identifier), the given identifier value should be lower or equal to `0x1FFF_FFFF`.

If one or more primary filters do not conform, the execution of the `begin` method returns an error – see table 5 page 26.

12.5 The `receive` method revisited

The `receive` method retrieves a received message. When you define primary filters, the value of the `idx` property of the `message` is the matching filter index. For example:

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456), // Filter #0
    ACANPrimaryFilter (kData, kStandard, 0x234),    // Filter #1
    ACANPrimaryFilter (kRemote, kStandard, 0x542)   // Filter #2
  } ;
  const uint32_t errorCode = ACAN::can0.begin (settings, primaryFilters, 3) ;
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN::can0.receive (message)) { // Only frames that pass a filter are retrieved
    switch (message.idx) {
    case 0:
      handle_myMessage_0 (message) ; // Extended data frame, id is 0x123456
      break ;
    case 1:
      handle_myMessage_1 (message) ;  // Standard data frame, id is 0x234
      break ;
    case 2:
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
      break ;
    default:
      break ;
    }
  }
  ...
}
```

An improvement is to use the `dispatchReceivedMessage` method – see section 14 page 23.

13 Secondary filters

Depending from the configuration, you can define several *secondary filters* – see table 4 page 14.

13.1 Secondary filters, without primary filter

This is an example without primary filter, and with secondary filters:

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  ...
  const ACANSecondaryFilter secondaryFilters [] = {
    ACANSecondaryFilter (kData, kExtended, 0x123456), // Filter #0
    ACANSecondaryFilter (kData, kStandard, 0x234),    // Filter #1
    ACANSecondaryFilter (kRemote, kStandard, 0x542)   // Filter #2
  } ;
  const uint32_t errorCode = ACAN::can0.begin (settings,
                                             NULL, 0, // No primary filter
                                             secondaryFilters, // The filter array
                                             3) ; // Filter array size
  ...
void loop () {
  CANMessage message ;
  if (ACAN::can0.receive (message)) { // Only frames that pass a filter are retrieved
    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) ;  // Standard data frame, id is 0x234
    }else if (message.rtr && !message.ext && (message.id == 0x542)) {
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
    }
  }
  ...
}
}
```

Each element of the `secondaryFilters` constant array defines an acceptance filter. Should be specified[7]:

- the required kind: data frames (`kData`) or remote frames (`kRemote`);
- the required format: standard frames (`kStandard`) or extended frames (`kExtended`);
- the required identifier value.

Maximum number of *secondary filters*. The number of *secondary filters* is limited: 12 by default, as the default value of `settings.mConfiguration` is `ACANSettings::k12_12_Filters`. See section 11 page 14 for getting the number of *secondary filters* for each configuration, and for changing default value.

Test order. The FlexCAN hardware examines the filters in the increasing order of their indexes in the `secondaryFilters` constant array. As soon as a match occurs, the message is transfered to Rx FIFO buffer and the examination process is completed. If no match occurs, the message is lost.

A consequence is if a filter appears twice, the second occurrence will never match.

[7] There is a fourth optional argument, that is `NULL` by default – see section 14 page 23.

13.2 Primary and secondary filters

This is an example with one primary filter, and two secondary filters:

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456), // Filter #0
  } ;
  const ACANSecondaryFilter secondaryFilters [] = {
    ACANSecondaryFilter (kData, kStandard, 0x234),    // Filter #1
    ACANSecondaryFilter (kRemote, kStandard, 0x542)   // Filter #2
  } ;
  const uint32_t errorCode = ACAN::can0.begin (settings,
                                               primaryFilters,
                                               1, // Primary filter array size
                                               secondaryFilters,
                                               2) ; // Secondary filter array size
  ...
void loop () {
  CANMessage message ;
  if (ACAN::can0.receive (message)) { // Only frames that pass a filter are retrieved
    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) ;  // Standard data frame, id is 0x234
    }else if (message.rtr && !message.ext && (message.id == 0x542)) {
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
    }
  }
  ...
}
```

Test order. The FlexCAN hardware performs sequentially:

- testing the primary filters in the increasing order of their indexes in the `primaryFilters` constant array;
- as soon as a match with a primary filter occurs, the message is transfered to Rx FIFO buffer and the examination process is completed;
- if no match occurs, testing the secondary filters in the increasing order of their indexes in the `secondaryFilters` constant array;
- as soon as a match with a secondary filter occurs, the message is transfered to Rx FIFO buffer and the examination process is completed;
- if no match occurs, the message is lost.

A consequence is if a filter appears twice, the second occurrence will never match. If a secondary filter matches the same message that a primary filter, the secondary filter will never match.

13.3 Secondary filter as pass-all filter

You can specify a secondary filter that matches any frame:

```
    ACANSecondaryFilter ()
```

You can use it for accepting all frames that did not match previous filters:

```
void setup () {
  ...
  const ACANSecondaryFilter secondaryFilters [] = {
    ACANSecondaryFilter (kData, kExtended, 0x123456), // Filter #0
    ACANSecondaryFilter (kData, kStandard, 0x234),     // Filter #1
    ACANSecondaryFilter (kRemote, kStandard, 0x542),  // Filter #2
    ACANSecondaryFilter ()                             // Filter #3
  } ; // Filter #3 catches any message that did not match filters #0, #1 and #2
  ...
}
```

Be aware if the pass-all filter is not the last one, following ones will never match.

```
void setup () {
  ...
  const ACANSecondaryFilter primaryFilters [] = {
    ACANSecondaryFilter (kData, kExtended, 0x123456), // Filter #0
    ACANSecondaryFilter (kData, kStandard, 0x234),     // Filter #1
    ACANSecondaryFilter (),                            // Filter #2
    ACANSecondaryFilter (kRemote, kStandard, 0x542)   // Filter #3
  } ; // Filter #3 will never match
  ...
}
```

If you use a primary pass-all filter, secondary filters will never match:

```
void setup () {
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456) // Filter #0
    ACANPrimaryFilter (),                          // Filter #1 - pass-all
  } ;
  const ACANSecondaryFilter secondaryFilters [] = {
    ACANSecondaryFilter (kData, kStandard, 0x234),   // Filter never matches
    ACANSecondaryFilter (kRemote, kStandard, 0x542) // Filter never matches
  } ;
  ...
```

13.4 Secondary filter conformance

The pass-all secondary filter (section 13.3 page 21) always conforms.

For a standard frame (11-bit identifier), a secondary filter definition is conform if the given identifier value is lower or equal to `0x7FF`.

For a extended frame (29-bit identifier), a secondary filter definition is conform if the given identifier value is lower or equal to `0x1FFF_FFFF`.

13.5 The `receive` method revisited

The `receive` method retrieves a received message. When you define primary and secondary filters, the value of the `idx` property of the `message` is the matching filter index. Filters are numbering from 0, starting by the first element of the first primary filter array until the last one, and continuing from the first element of the secondary filter array, until its last element. So the the `idx` property of the `message` can be used for dispatching the received message:

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456), // Filter #0
  } ;
  const ACANSecondaryFilter secondaryFilters [] = {
    ACANSecondaryFilter (kData, kStandard, 0x234),    // Filter #1
    ACANSecondaryFilter (kRemote, kStandard, 0x542)   // Filter #2
  } ;
  const uint32_t errorCode = ACAN::can0.begin (settings,
                                               primaryFilters, 1,
                                               secondaryFilters, 2) ;
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN::can0.receive (message)) { // Only frames that pass a filter are retrieved
    switch (message.idx) {
    case 0:
      handle_myMessage_0 (message) ; // Extended data frame, id is 0x123456
      break ;
    case 1:
      handle_myMessage_1 (message) ;  // Standard data frame, id is 0x234
      break ;
    case 2:
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
      break ;
```

```
    default:
      break ;
    }
  }
  ...
}
```

An improvement is to use the `dispatchReceivedMessage` method – see section 14 page 23.

14 The `dispatchReceivedMessage` method

The last improvement is to call the `dispatchReceivedMessage` method – do not call the `receive` method any more. You can use it if you have defined primary and / or secondary filters that name a call-back function.

The primary and secondary filter constructors have as a last argument a call back function pointer. It defaults to `NULL`, so until now the code snippets do not use it.

For enabling the use of the `dispatchReceivedMessage` method, you add to each filter definition as last argument the function that will handle the message. In the `loop` function, call the `dispatchReceivedMessage` method: it dispatches the messages to the call back functions.

```
void setup () {
  ACANSettings settings (125 * 1000) ;
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456, handle_myMessage_0)
  } ;
  const ACANSecondaryFilter secondaryFilters [] = {
    ACANSecondaryFilter (kData, kStandard, 0x234, handle_myMessage_1),
    ACANSecondaryFilter (kRemote, kStandard, 0x542, handle_myMessage_2)
  } ;
  const uint32_t errorCode = ACAN::can0.begin (settings,
                                               primaryFilters, 1,
                                               secondaryFilters, 2) ;
  ...
}

void loop () {
  ACAN::can0.dispatchReceivedMessage () ; // Do not use ACAN::can0.receive any more
  ...
}
```

The `dispatchReceivedMessage` method handles one message at a time. More precisely:

- if it returns `false`, the driver receive buffer was empty;
- if it returns `true`, the driver receive buffer was not empty, one message has been removed and dispatched.

So, the return value can used for emptying and dispatching all received messages:

```
void loop () {
  while (ACAN::can0.dispatchReceivedMessage ()) {
  }
  ...
}
```

If a filter definition does not name a call back function, the corresponding messages are lost. In the code below, filter #1 does not name a call back function, standard data frames with identifier `0x234` are lost.

```
void setup () {
  ...
  const ACANPrimaryFilter primaryFilters [] = {
    ACANPrimaryFilter (kData, kExtended, 0x123456, handle_myMessage_0)
  } ;
  const ACANSecondaryFilter secondaryFilters [] = {
    ACANSecondaryFilter (kData, kStandard, 0x234), // Filter #1
    ACANSecondaryFilter (kRemote, kStandard, 0x542, handle_myMessage_2)
  } ;
  ...
}
```

The `dispatchReceivedMessage` method has an optional argument – `NULL` by default: a function name. This function is called for every message that pass the receive filters, with an argument equal to the matching filter index:

```
void filterMatchFunction (const uint32_t inFilterIndex) {
  ...
}

void loop () {
  ACAN::can0.dispatchReceivedMessage (filterMatchFunction) ;
  ...
}
```

You can use this function for maintaining statistics about receiver filter matches.

15 The `ACAN::begin` method reference

15.1 The `ACAN::begin` method prototype

The `begin` method prototype is:

```
uint32_t ACAN::begin (const ACANSettings & inSettings,
                      const ACANPrimaryFilter inPrimaryFilters [] = NULL,
                      const uint32_t inPrimaryFilterCount = 0,
                      const ACANSecondaryFilter inSecondaryFilters [] = NULL,
```

```
                        const uint32_t inSecondaryFilterCount = 0) ;
```

The four last arguments have default values.

Omitting the last argument makes no secondary filter is defined:

```
const uint32_t errorCode = ACAN::can0.begin (settings,
                                             primaryFilters, primaryFilterCount,
                                             secondaryFilters) ;
```

Omitting the last two arguments makes no secondary filter is defined:

```
const uint32_t errorCode = ACAN::can0.begin (settings, primaryFilters, primaryFilterCount) ;
```

Omitting the last three or the last four arguments makes no primary and no secondary filter is defined – so any (data / remote, standard / extended) frame is received:

```
const uint32_t errorCode = ACAN::can0.begin (settings, primaryFilters) ;
```

```
const uint32_t errorCode = ACAN::can0.begin (settings) ;
```

15.2 The error code

The `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 5. An error code could report several errors. Bits from 0 to 11 are actually defined by the `ACANSettings` class and are also returned by the `CANBitSettingConsistency` method (see section 16.2 page 31). Bits from 12 are defined by the `ACAN` class.

The `ACANSettings` class defines static constant properties that can be used as mask error:

```
  public: static const uint32_t kBitRatePrescalerIsZero           = 1 <<  0 ;
  public: static const uint32_t kBitRatePrescalerIsGreaterThan256 = 1 <<  1 ;
  public: static const uint32_t kPropagationSegmentIsZero         = 1 <<  2 ;
  public: static const uint32_t kPropagationSegmentIsGreaterThan8 = 1 <<  3 ;
  public: static const uint32_t kPhaseSegment1IsZero              = 1 <<  4 ;
  public: static const uint32_t kPhaseSegment1IsGreaterThan8      = 1 <<  5 ;
  public: static const uint32_t kPhaseSegment2IsZero              = 1 <<  6 ;
  public: static const uint32_t kPhaseSegment2IsGreaterThan8      = 1 <<  7 ;
  public: static const uint32_t kRJWIsZero                        = 1 <<  8 ;
  public: static const uint32_t kRJWIsGreaterThan4                = 1 <<  9 ;
  public: static const uint32_t kRJWIsGreaterThanPhaseSegment2    = 1 << 10 ;
  public: static const uint32_t kPhaseSegment1Is1AndTripleSampling = 1 << 11 ;
```

The `ACAN` class defines static constant properties that can be used as mask error:

```
  public: static const uint32_t kTooMuchPrimaryFilters     = 1 << 12 ;
  public: static const uint32_t kNotConformPrimaryFilter   = 1 << 13 ;
  public: static const uint32_t kTooMuchSecondaryFilters   = 1 << 14 ;
  public: static const uint32_t kNotConformSecondaryFilter = 1 << 15 ;
  public: static const uint32_t kNoAlternateTxPinForCan1   = 1 << 16 ;
  public: static const uint32_t kNoAlternateRxPinForCan1   = 1 << 17 ;
  public: static const uint32_t kCANBitConfiguration       = 1 << 18 ;
```

Bit number	Comment	Link
0	`mBitRatePrescaler` == 0	
1	`mBitRatePrescaler` > 256	
2	`mPropagationSegment` == 0	
3	`mPropagationSegment` > 8	
4	`mPhaseSegment1` == 0	
5	`mPhaseSegment1` > 8	
6	`mPhaseSegment2` == 0	
7	`mPhaseSegment2` > 8	
8	`mRJW` == 0	
9	`mRJW` > 4	
10	`mRJW` > `mPhaseSegment2`	
11	`mPhaseSegment2` == 1 and *triple sampling*	
12	Too much primary filters	section 15.2.3 page 27
13	Primary filter conformance error	section 15.3 page 27
14	Too much secondary filters	section 15.3.1 page 27
15	Secondary filter conformance error	section 15.3.2 page 27
16	`ACAN::can1` has no Tx alternate pin	section 15.3.3 page 27
17	`ACAN::can1` has no Rx alternate pin	section 15.3.4 page 27
18	Inconsistent CAN Bit configuration	section 15.2.2 page 27

Table 5 – The `ACAN::begin` method error codes

For example, you can write:

```
const uint32_t errorCode = ACAN::can0.begin (settings,
                                             primaryFilters, primaryFilterCount,
                                             secondaryFilters, secondaryFilterCount) ;
if (errorCode != 0) {
  // Error(s)
  if (errorCode & ACAN::kTooMuchPrimaryFilters) {
    // Error: too much primary filters
  }
  ...
}
```

15.2.1 CAN Bit setting too far from wished rate

This error is raised when the `mBitConfigurationClosedToWishedRate` of the `settings` object is false. This means that the `ACANSettings` constructor cannot compute a CAN bit configuration close enough to the wished bit rate. When the `begin` is called with `settings.mBitConfigurationClosedToWishedRate` false, this error is reported. For example:

```
void setup () {
  ACANSettings settings (1) ; // 1 bit/s !!!
  // Here, settings.mBitConfigurationClosedToWishedRate is false
  const uint32_t errorCode = ACAN::can0.begin (settings) ;
```

```
  // Here, errorCode == ACAN::kCANBitConfigurationTooFarFromWishedBitRateErrorMask
}
```

This error is a fatal error, the driver and the FlexCAN module are not configured. See section 16.1 page 28 for a discussion about CAN bit setting computation.

15.2.2 CAN Bit inconsistent configuration error

This error is raised when you have changed the CAN bit properties (`mBitRatePrescaler`, `mPropagationSegment`, `mPhaseSegment1`, `mPhaseSegment2`, `mRJW`), and one or more resulting values are inconsistent. See section 16.2 page 31.

15.2.3 Too much primary filters error

The number of *primary filters* is limited. See section 11 page 14 for getting the number of *primary filters* for each configuration, and for changing default value.

15.3 Primary filters conformance error

One or several primary filters do not conform: see section 12.4 page 17. Comment out primary filter definitions until finding the faulty definition.

15.3.1 Too much secondary filters error

The number of *secondary filters* is limited. See section 11 page 14 for getting the number of *secondary filters* for each configuration, and for changing default value.

15.3.2 Secondary filter conformance error

One or several secondary filters do not conform: see section 13.4 page 22. Comment out secondary filter definitions until finding the faulty definition.

15.3.3 No alternate Tx pin error

In the Teensy 3.6, `ACAN::can1` does not support alternate Tx pin.

15.3.4 No alternate Rx pin error

In the Teensy 3.6, `ACAN::can1` does not support alternate Rx pin.

16 ACANSettings class reference

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

16.1 The ACANSettings constructor: computation of the CAN bit settings

The constructor of the ACANSettings has one mandatory argument: the wished bit rate. It tries to compute the CAN bit settings for this bit rate. If it succeeds, the constructed object has its mBitConfigurationClosedToWishedRate property set to true, otherwise it is set to false. For example:

```
void setup () {
  ACANSettings settings (1 * 1000 * 1000) ; // 1 Mbit/s
  // Here, settings.mBitConfigurationClosedToWishedRate is true
  ...
}
```

Of course, CAN bit computation always succeeds for classical bit rates: 1 Mbit/s, 500 kbit/s, 250 kbit/s, 125 kbit/s. But CAN bit computation can also succeed for some unusual bit rates, as 842 kbit/s. You can check the result by computing actual bit rate, and the distance from the wished bit rate:

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (842 * 1000) ; // 842 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 124 ppm
  ...
}
```

The actual bit rate is 842,105 bit/s, and its distance from wished bit rate is 124 ppm. "ppm" stands for "part-per-million", and $1\ \text{ppm} = 10^{-6}$. In other words, $10,000\ \text{ppm} = 1\%$.

By default, a wished 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 second argument of ACANSettings constructor:

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (842 * 1000, 100) ; // 842 kbit/s, max distance is 100 ppm
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 124 ppm
```

```
  ...
}
```

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

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (500 * 1000, 0) ; // 500 kbit/s, max distance is 0 ppm
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  500,000 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 0 ppm
  ...
}
```

The fastest exact bit rate is 3,2 Mbit/s. It works when the FlexCAN module is configured in both *loop back* mode (section 16.7.3 page 35) and *self reception* mode (section 16.7.2 page 35). Note bit rates above 1 Mbit/s do not conform to the `ISO-11898`; CAN transceivers as `MCP2551` require the bit rate lower or equal to 1 Mbit/s.

The slowest exact bit rate is 2.5 kbit/s. Note many CAN transceivers as the `MCP2551` provide "*detection of ground fault (permanent Dominant) on TXD input*". For example, the `MCP2551` constraints the bit rate to be greater or equal to 16 kbit/s. If you want to work with slower bit rates and you need a transceiver, use one without this detection, as the `PCA82C250`.

In any way, the bit rate computation always gives a consistent result, resulting an actual bit rate closest from the wished bit rate. For example:

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 10,100 ppm
  ...
}
```

You can get the details of the CAN bit decomposition. For example:

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
```

```
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 10,100 ppm
  Serial.print ("Bit rate prescaler: ") ;
  Serial.println (settings.mBitRatePrescaler) ; // BRP = 2
  Serial.print ("Propagation segment: ") ;
  Serial.println (settings.mPropagationSegment) ; // PropSeg = 6
  Serial.print ("Phase segment 1: ") ;
  Serial.println (settings.mPhaseSegment1) ; // PS1 = 5
  Serial.print ("Phase segment 2: ") ;
  Serial.println (settings.mPhaseSegment2) ; // PS2 = 6
  Serial.print ("Resynchronization Jump Width: ") ;
  Serial.println (settings.mRJW) ; // RJW = 4
  Serial.print ("Triple Sampling: ") ;
  Serial.println (settings.mTripleSampling) ; // 0, meaning single sampling
  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 wished 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` value, and decrement the `mPhaseSegment2` value in order to sample the `CAN Rx` pin later.

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  settings.mPhaseSegment1 ++ ; // 5 -> 6: safe, 1 <= PS1 <= 8
  settings.mPhaseSegment2 -- ; // 5 -> 4: safe, 2 <= PS2 <= 8 and RJW <= PS2
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 75, meaning 75%
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  500000: ok, bit rate did not change
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

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

$$
\begin{aligned}
1 &\leqslant \texttt{mBitRatePrescaler} \leqslant 256 \\
1 &\leqslant \texttt{mRJW} \leqslant 4 \\
1 &\leqslant \texttt{mPropagationSegment} \leqslant 8 \\
\text{Single sampling: } 1 &\leqslant \texttt{mPhaseSegment1} \leqslant 8 \\
\text{Triple sampling: } 2 &\leqslant \texttt{mPhaseSegment1} \leqslant 8 \\
2 &\leqslant \texttt{mPhaseSegment2} \leqslant 8 \\
\texttt{mRJW} &\leqslant \texttt{mPhaseSegment2}
\end{aligned}
$$

Resulting actual bit rate is given by:

$$\text{Actual bit rate} = \frac{16\,\text{MHz}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2})}$$

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

$$\text{Sampling point \textit{(single sampling)}} = 100 \cdot \frac{1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1}}{1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2}}$$

$$\text{Sampling first point \textit{(triple sampling)}} = 100 \cdot \frac{\texttt{mPropagationSegment} + \texttt{mPhaseSegment1}}{1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2}}$$

16.2 The `CANBitSettingConsistency` method

This method checks the CAN bit decomposition (given by `mBitRatePrescaler`, `mPropagationSegment`, `mPhaseSegment1`, `mPhaseSegment2`, `mRJW` property values) is consistent.

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  settings.mPhaseSegment1 = 0 ; // Error, mPhaseSegment1 should be >= 1 (and <= 8)
  Serial.print ("Consistency: 0x") ;
  Serial.println (settings.CANBitSettingConsistency (), HEX) ; // 0x10, 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 6.

The `ACANSettings` class defines static constant properties that can be used as mask error:

```
  public: static const uint32_t kBitRatePrescalerIsZero           = 1 <<  0 ;
  public: static const uint32_t kBitRatePrescalerIsGreaterThan256  = 1 <<  1 ;
```

Bit number	Error
0	mBitRatePrescaler == 0
1	mBitRatePrescaler > 256
2	mPropagationSegment == 0
3	mPropagationSegment > 8
4	mPhaseSegment1 == 0
5	mPhaseSegment1 > 8
6	mPhaseSegment2 == 0
7	mPhaseSegment2 > 8
8	mRJW == 0
9	mRJW > 4
10	mRJW > mPhaseSegment2
11	mPhaseSegment2 == 1 and *triple sampling*

Table 6 – The ACANSettings::CANBitSettingConsistency method error codes

```
public: static const uint32_t kPropagationSegmentIsZero          = 1 <<  2 ;
public: static const uint32_t kPropagationSegmentIsGreaterThan8  = 1 <<  3 ;
public: static const uint32_t kPhaseSegment1IsZero               = 1 <<  4 ;
public: static const uint32_t kPhaseSegment1IsGreaterThan8       = 1 <<  5 ;
public: static const uint32_t kPhaseSegment2IsZero               = 1 <<  6 ;
public: static const uint32_t kPhaseSegment2IsGreaterThan8       = 1 <<  7 ;
public: static const uint32_t kRJWIsZero                         = 1 <<  8 ;
public: static const uint32_t kRJWIsGreaterThan4                 = 1 <<  9 ;
public: static const uint32_t kRJWIsGreaterThanPhaseSegment2     = 1 << 10 ;
public: static const uint32_t kPhaseSegment1Is1AndTripleSampling = 1 << 11 ;
```

16.3 The actualBitRate method

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

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  ...
}
```

Note. If CAN bit settings are not consistent (see section 16.2 page 31), the returned value is irrelevant.

16.4 The `exactBitRate` method

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

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (842 * 1000) ; // 842 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 124 ppm
  Serial.print ("Exact: ") ;
  Serial.println (settings.exactBitRate ()) ; // 0 (---> false)
  ...
}
```

Note. If CAN bit settings are not consistent (see section 16.2 page 31), the returned value is irrelevant.

16.5 The `ppmFromWishedBitRate` method

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

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (842 * 1000) ; // 842 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 124 ppm
  ...
}
```

Note. If CAN bit settings are not consistent (see section 16.2 page 31), the returned value is irrelevant.

16.6 The `samplePointFromBitStart` method

The `samplePointFromBitStart` method returns the distance of sample point from the start of the CAN bit, expressed in part-per-cent (ppc): $1\ \mathrm{ppc} = 1\% = 10^{-2}$. If triple sampling is selected, the returned value is the distance of the first sample point from the start of the CAN bit. It is a good practice to get sample point from 65% to 80%.

```
void setup () {
  Serial.begin (9600) ;
  ACANSettings settings (500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("Sample point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 68 --> 68%
  ...
}
```

Note. If CAN bit settings are not consistent (see section 16.2 page 31), the returned value is irrelevant.

16.7 Properties of the ACANSettings class

All properties of the `ACANSettings` class are declared `public` and are initialized (table 7). The default values of properties from `mWhishedBitRate` until `mTripleSampling` corresponds to a CAN bit rate of 250,000 bit/s.

Property	Type	Initial value	Comment
mWhishedBitRate	uint32_t	250,000	See section 16.1 page 28
mBitRatePrescaler	uint16_t	4	See section 16.1 page 28
mPropagationSegment	uint8_t	5	See section 16.1 page 28
mPhaseSegment1	uint8_t	5	See section 16.1 page 28
mPhaseSegment2	uint8_t	5	See section 16.1 page 28
mRJW	uint8_t	4	See section 16.1 page 28
mTripleSampling	bool	false	See section 16.1 page 28
mBitConfigurationClosedToWishedRate	bool	true	See section 16.1 page 28
mListenOnlyMode	bool	false	See section 16.7.1 page 34
mSelfReceptionMode	bool	false	See section 16.7.2 page 35
mLoopBackMode	bool	false	See section 16.7.3 page 35
mConfiguration	tConfiguration	k12_12_Filters	See section 11 page 14
mUseAlternateTxPin	bool	false	See section 7 page 9
mUseAlternateRxPin	bool	false	See section 7 page 9
mMessageIRQPriority	uint8_t	64	See section 16.7.4 page 35
mReceiveBufferSize	uint16_t	32	See section 10.1 page 13
mTransmitBufferSize	uint16_t	16	See section 8.2 page 10
mTxPinIsOpenCollector	bool	false	See section 16.7.5 page 35
mRxPinHasInternalPullUp	bool	false	See section 16.7.6 page 35

Table 7 – Properties of the `ACANSettings` class

16.7.1 The `mListenOnlyMode` property

This boolean property corresponds to the `LOM` bit of the FlexCAN `CTRL1` control register.

16.7.2 The `mSelfReceptionMode` property

This boolean property corresponds to the complement of the `SRXDIS` bit of the FlexCAN `MCR` control register.

16.7.3 The `mLoopBackMode` property

This boolean property corresponds to the `LBP` bit of the FlexCAN `CTRL1` control register.

16.7.4 The `mMessageIRQPriority` property

This property sets the priority of the *CAN message* interrupt. Highest priority is 0, lowest is 255.

16.7.5 The `mTxPinIsOpenCollector` property

When the `mTxPinIsOpenCollector` property is set to `true`, the `RECESSIVE` output state puts the `Tx` pin `Hi-Z`, instead of driving high. The `Tx` pin is always driving low in `DOMINANT` state.

Output state	Tx Pin Output
`DOMINANT`	`0`
`RECESSIVE`	`1`

(a) `mTxPinIsOpenCollector` is `false` (default)

Output state	Tx Pin Output
`DOMINANT`	`0`
`RECESSIVE`	`Hi-Z`

(b) `mTxPinIsOpenCollector` is `true`

Table 8 – `Tx` pin output, following the `mTxPinIsOpenCollector` property setting

16.7.6 The `mRxPinHasInternalPullUp` property

By setting this property, the `Rx` pin is configured with the internal pullup enabled. This ensures that `RECESSIVE` values are received if the pin is unconnected.

17 CAN controller state

Three methods return the CAN controller state, the receive error counter and the transmit error counter.

17.1 The `controllerState` method

```
public: tControllerState controllerState (void) const ;
```

This method returns the current state (*error active*, *error passive*, *bus off*) of the CAN controller. The `tControllerState` type is defined by an enumeration:

```
typedef enum {kActive, kPassive, kBusOff} tControllerState ;
```

17.2 The `receiveErrorCounter` method

```
public: uint32_t receiveErrorCounter (void) const ;
```

17.3 The `transmitErrorCounter` method

```
public: uint32_t transmitErrorCounter (void) const ;
```

As the `CANx_ESR` FlexCAN control register does not return a valid value when the CAN controller is in the *bus off* state, the value 256 is forced.