ACAN_T4 CAN and CANFD library for Teensy 4.0 / 4.1
Version 1.1.5

Pierre Molinaro

October 1, 2021

Contents

1 Versions

Version	Date	Comment
1.1.5	October 1, 2021	Added data_s64, data_s32, data_s16 and data_s8 to CANMessage class union members, see section 5 page 10 (thanks to tomtom0707).
1.1.4	July 31, 2021	Added *root CAN Clock* API, see section 37 page 74.
1.1.3	July 19, 2021	Fixed FPROPSEG setting (thanks to Liz).
1.1.2	April 21, 2021	Added x9 and x10 data bitrate factors (thanks to Pedro Dionisio Pereira Junior).
1.1.1	April 27, 2020	Added dataFloat to CANMessage (thanks to Koryphon) Added several forgotten volatile
1.1.0	December 31, 2019	For compatibility with ACAN2517FD library, the DataBitRateFactor enumeration is declared outside of the ACAN_T4FD_Settings class.
1.0.0	October 18, 2019	Initial release

2 Features

The ACAN_T4 library is a CAN ("Controller Area Network") driver for Teensy 4.0 / 4.1. 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 and CANFD bit settings computation from user bitrate;
- user can fully define its own CAN and CANFD bit setting values;
- reception filters are easily defined;
- 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 (output impedance, open collector, alternate pin);
- `Rx` pin can be configured (intput pullup/pulldown, alternate pin).

Part I

CAN 2.0B

The three FLEXCAN modules of the Teensy 4.0 / 4.1 microcontroller handle CAN 2.0B.

3 Data flow

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

FLEXCAN module is hardware, integrated into the micro-controller. It implements 64 MBs (*Message Buffers*), used for the *data frame transmit buffer*, *remote frame transmit buffer(s)*, *reception FIFO* and *reception filters*. Theses 64 MBs are used as follows:

- MB 0-37 implement a 6-messages deep RxFIFO, up to 32 primary filters (see section 13 page 19) and up to 96 secondary filters (see section 14 page 22);
- MB 38-62 are used for sending remote frames;
- MB 63 is used for sending data frames.

Note. Teensy 3.x FLEXCAN modules implement 16 MBs. So the `ACANSetting` class has a `mConfiguration` property that defines the MB assignment. As Teensy 4.0 / 4.1 has 64 MBs, I had removed this property and defined a non configurable assignment.

Sending messages. The FLEXCAN hardware makes sending data frames different from sending remote frames. For both, user code calls the `tryToSend` method – see section 9 page 14 for sending data frames, and section 10 page 16 for sending remote frames. The data frames are stored in the *Driver Transmit Buffer*, before to 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 9.2 page 15 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 13 page 19 and section 14 page 22 for configuring them. Messages that pass the filters are stored in the *Reception FIFO*. Its depth is not configurable – it is always

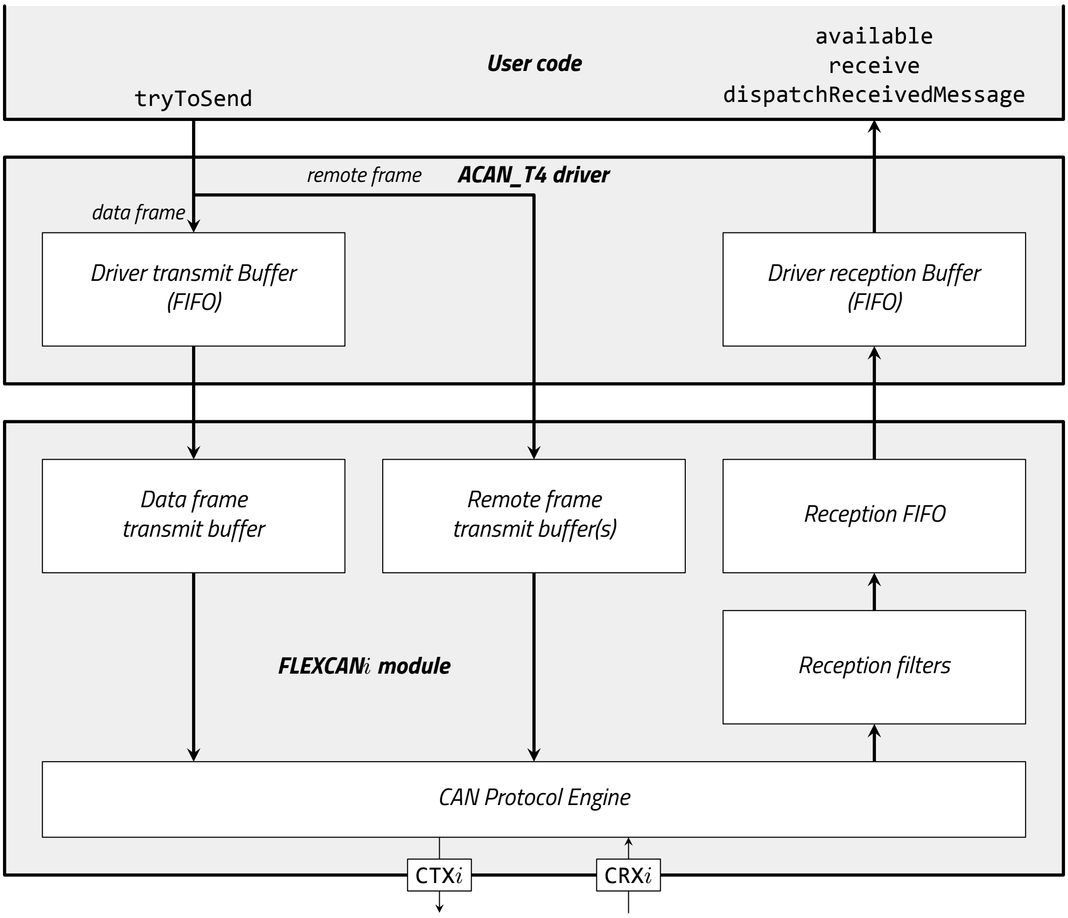

Figure 1 – Message flow in the ACAN_T4::can$i$ driver and FLEXCANi module, $1 \leqslant i \leqslant 3$

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 12.1 page 18 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 12 page 17, section 13.5 page 22 and section 14.5 page 26;
- the `dispatchReceivedMessage` method if you have defined primary and / or secondary filters that name a call-back function – see section 15 page 26.

Sequentiality. The ACAN_T4 driver and the configuration of the FLEXCAN module 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: LoopBackDemoCAN1

The LoopBackDemoCAN1 sketch is a sample code for introducing the ACAN_T4 library[1]. 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.

```
#ifndef __IMXRT1062__
  #error "This sketch should be compiled for Teensy 4.0 / 4.1"
#endif

#include <ACAN_T4.h>

void setup () {
  pinMode (LED_BUILTIN, OUTPUT) ;
  Serial.begin (9600) ;
  while (!Serial) {
    delay (50) ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
  }
  Serial.println ("CAN1 loopback test") ;
  ACAN_T4_Settings settings (125 * 1000) ; // 125 kbit/s
  settings.mLoopBackMode = true ;
  settings.mSelfReceptionMode = true ;
  const uint32_t errorCode = ACAN_T4::can1.begin (settings) ;
  if (0 == errorCode) {
    Serial.println ("can1 ok") ;
  }else{
    Serial.print ("Error can1: 0x") ;
    Serial.println (errorCode, HEX) ;
  }
}

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

void loop () {
  if (gBlinkDate <= millis ()) {
    gBlinkDate += 500 ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
  }
  CANMessage message ;
  if (gSendDate <= millis ()) {
    message.id = 0x542 ;
    const bool ok = ACAN_T4::can1.tryToSend (message) ;
    if (ok) {
      gSendDate += 2000 ;
      gSentCount += 1 ;
      Serial.print ("Sent: ") ;
      Serial.println (gSentCount) ;
    }
```

[1] See also the demoCAN1CAN2CAN3 sketch, section 19 page 39.

```
47     }
48     if (ACAN_T4::can1.receive (message)) {
49       gReceivedCount += 1 ;
50       Serial.print ("Received: ") ;
51       Serial.println (gReceivedCount) ;
52     }
53   }
```

Line 1 to 3. This ensures the Teensy 4.0 / 4.1 board is selected.

Line 5. This line includes the `ACAN_T4` library.

Line 9 to 13. Start serial (the `9600` argument value is ignored by Teensy), and blink quickly until the *Arduino Serial Monitor* is opened.

Line 15. Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACAN_T4_Settings` class. The constructor has one parameter: the wished CAN bitrate. It returns a `settings` object fully initialized with CAN bit settings for the wished bitrate, and default values for other configuration properties.

Lines 16 and 17. 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 enable *loop back*, that is you can run this demo sketch even it you have no connection to a physical CAN network. The section 17.8 page 37 lists all properties you can override.

Line 18. This is the third step, configuration of the `ACAN_T4::can1` driver with `settings` values. You cannot change the `ACAN_T4::can1` name – see section 6 page 11. 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 13 page 19 and section 14 page 22.

Lines 19 to 24. Last step: the configuration of the `ACAN_T4::can1` driver returns an error code, stored in the `errorCode` constant. It has the value 0 if all is ok – see section 16.2 page 29.

Line 27. The `gBlinkDate` global variable is used for blinking Teensy LED every 0.5 s.

Line 28. The `gSendDate` global variable is used for sending a CAN message every 2 s.

Line 29. The `gSentCount` global variable counts the number of sent messages.

Line 30. The `gReceivedCount` global variable counts the number of received messages.

Line 33 to 36. Blink Teensy LED.

Line 37. 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 10.

Line 38. It tests if it is time to send a message.

Line 39. Set the message identifier. In a real code, we set here message data, and for an extended frame the `ext` boolean property.

Line 40. 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 41 to 46. 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 48. As the FLEXCAN module is configured in *loop back* mode (see lines 16 and 17), 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 49 to 51. 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 M7 processor of Teensy 4.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 13.5 page 22 and section 14.5 page 26);

- 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.

Module	Driver name
FLEXCAN1	ACAN_T4::can1
FLEXCAN2	ACAN_T4::can2
FLEXCAN3	ACAN_T4::can3

Table 1 – Driver global variables

Note. Drivers variables are ACAN_T4 class static properties. This choice may seem strange. However, a common error is to declare its own driver variable:

```
ACAN_T4 myCAN ; // Don't do that, it is an error !!!
```

Declaring drivers variables as ACAN_T4 class static properties[2] enables the compiler to raise an error if you try to declare your own driver variable.

7 CRXi pin configuration

You can change CRXi pin following settings:

- its input impedance (section 7.1 page 11, 47kΩ pullup by default);
- choosing an alternate pin (section 7.2 page 12).

7.1 Input impedance

An input pin of the Teensy 4.0 / 4.1 micro-controller has different pullup / pulldown configurations. Five settings are available:

```
class ACAN_T4_Settings {
  ...
  public: typedef enum : uint8_t {
    NO_PULLUP_NO_PULLDOWN = 0, // PUS = 0, PUE = 0, PKE = 0
    PULLDOWN_100k = 0b0011, // PUS = 0, PUE = 1, PKE = 1
    PULLUP_47k    = 0b0111, // PUS = 1, PUE = 1, PKE = 1
    PULLUP_100k   = 0b1011, // PUS = 2, PUE = 1, PKE = 1
    PULLUP_22k    = 0b1111  // PUS = 3, PUE = 1, PKE = 1
  } RxPinConfiguration ;
  ...
} ;
```

[2]The ACAN_T4 constructor is declared `private`.

By default, PULLUP_47k is selected. For setting an other value, write for example:

```
settings.mRxPinConfiguration = ACAN_T4_Settings::PULLUP_100k ;
```

7.2 Alternate CRXi pin

FLEXCAN1 accepts one alternate input pin, FLEXCAN2 and FLEXCAN3 have no alternate input pin on Teensy 4.0 / 4.1 (table 2).

Module	Default Rx pin	Alternate Rx pin
FLEXCAN1	#23	#13
FLEXCAN2	#1	*no alternate pin*
FLEXCAN3	#30	*no alternate pin*

Table 2 – Teensy 4.0 / 4.1 CAN Rx pins

The `mRxPin` property of the `ACAN_T4_Settings` class specifies the pin number. By default, it is set to `255`, meaning using default pin.

For example, for using `FLEXCAN1` alternate pin, write:

```
settings.mRxPin = 13 ;
```

If you select an invalid pin number, the error `kInvalidRxPin` is raised (table 7).

8 CTXi pin configuration

You can change CTXi pin following settings:

- its output impedance (section 8.1 page 12, 78Ω by default);
- push/pull or open collector (section 8.2 page 13);
- choosing an alternate pin (section 8.3 page 13).

8.1 Output impedance

An output pin of the Teensy 4.0 / 4.1 micro-controller has a programmable output impedance. Seven settings are available[3]:

Theses settings are defined by an enumerated type:

```
class ACAN_T4_Settings {
  ...
  public: typedef enum {
    IMPEDANCE_R0 = 1,
    IMPEDANCE_R0_DIVIDED_BY_2 = 2,
```

[3] *i.MX RT1060 Crossover Processors for Consumer Products*, IMXRT1060CEC, Rev. 0.1, 04/2019, Table 27 page 38.

Symbol	Typical value at 3.3V
ACAN_T4_Settings::IMPEDANCE_R0	157 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_2	78 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_3	53 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_4	39 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_5	32 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_6	26 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_7	23 Ω

Table 3 – GPIO output buffer average impedance, 3.3 V

```
    IMPEDANCE_R0_DIVIDED_BY_3 = 3,
    IMPEDANCE_R0_DIVIDED_BY_4 = 4,
    IMPEDANCE_R0_DIVIDED_BY_5 = 5,
    IMPEDANCE_R0_DIVIDED_BY_6 = 6,
    IMPEDANCE_R0_DIVIDED_BY_7 = 7
  } TxPinOutputBufferImpedance ;
  ...
} ;
```

By default, IMPEDANCE_R0_DIVIDED_BY_2 is selected. For setting an other value, write:

```
  settings.mTxPinOutputBufferImpedance = ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_7;
```

8.2 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 4 – Tx pin output, following the mTxPinIsOpenCollector property setting

8.3 Alternate CTXi pin

FLEXCAN1 accepts one alternate output pin, FLEXCAN2 and FLEXCAN3 have no alternate output pin on Teensy 4.0 / 4.1 (table 5).

The mTxPin property of the ACAN_T4_Settings class specifies the pin number. By default, it is set to 255, meaning using default pin.

For example, for using FLEXCAN1 alternate pin, write:

```
  settings.mTxPin = 11 ;
```

Module	Default Tx pin	Alternate Tx pin
FLEXCAN1	#22	#11
FLEXCAN2	#0	*no alternate pin*
FLEXCAN3	#31	*no alternate pin*

Table 5 – Teensy 4.0 / 4.1 CAN Tx pins

If you select an invalid pin number, the error `kInvalidTxPin` is raised (table 7).

9 Sending data frames

Note. This section applies only to **data** frames. For sending remote frames, see section 10 page 16.

9.1 `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_T4::can1.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_T4::can1.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_T4::can1.tryToSend (message) ;
    if (ok) {
      gSendMessage = false ;
    }
  }
  ...
}
```

9.2 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:

```
ACAN_T4_Settings settings (125 * 1000) ;
settings.mTransmitBufferSize = 30 ;
const uint32_t errorCode = ACAN_T4::can1.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`.

9.3 The `transmitBufferSize` method

The `transmitBufferSize` method returns the size of the driver transmit buffer, that is the value of `settings.mTransmitBuf`

```
const uint32_t s = ACAN_T4::can1.transmitBufferSize () ;
```

9.4 The `transmitBufferCount` method

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

```
const uint32_t n = ACAN_T4::can1.transmitBufferCount () ;
```

9.5 The `transmitBufferPeakCount` method

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

```
const uint32_t max = ACAN_T4::can1.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`.

10 Sending remote frames

Note. This section applies only to **remote** frames. For sending data frames, see section 9 page 14.

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_T4::can1.tryToSend (message) ;
...
```

11 Sending frames using the `tryToSendReturnStatus` method

```
uint32_t ACAN_T4::tryToSendReturnStatus (const CANMessage & inMessage) ;
```

This method is functionally identical to the `tryToSend` method, the only difference is the detailled return status:

- 0 if message has been successfully submitted (the call to the `tryToSend` method would have returned `true`);
- non zero if message has not been successfully submitted (the call to the `tryToSend` method would have returned `false`).

A non-zero return value is a bit field that details the error, as listed in table 6.

Bit Index	Constant	Comment
0	`kTransmitBufferOverflow`	Trying to send a data frame, but the transmit buffer is full (retry later).
1	`kNoAvailableMBForSendingRemoteFrame`	Trying to send a remote frame, but currently there is no available Message Buffer (retry later).
5	`kFlexCANinCANFDBMode`	CAN3 is in CANFD mode, not CAN 2.0B mode.

Table 6 – `tryToSendReturnStatus` method returned status bits

12 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 15 page 26).

This is a basic example:

```
void setup () {
  ACAN_T4_Settings settings (125 * 1000) ;
  ...
  const uint32_t errorCode = ACAN_T4::can1.begin (settings) ; // No receive filter
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN_T4::can1.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 () {
  ACAN_T4_Settings settings (125 * 1000) ;
  ...
  const uint32_t errorCode = ACAN_T4::can1.begin (settings) ; // No receive filter
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN_T4::can1.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.

12.1 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:

```
ACAN_T4_Settings settings (125 * 1000) ;
settings.mReceiveBufferSize = 100 ;
const uint32_t errorCode = ACAN_T4::can1.begin (settings) ;
...
```

As the size of CANMessage class is 16 bytes, the actual size of the driver receive buffer is:

$$\texttt{settings.mReceiveBufferSize} * 16$$

12.2 The receiveBufferSize method

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

```
const uint32_t s = ACAN_T4::can1.receiveBufferSize () ;
```

12.3 The receiveBufferCount method

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

```
const uint32_t n = ACAN_T4::can1.receiveBufferCount () ;
```

12.4 The receiveBufferPeakCount method

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

```
const uint32_t max = ACAN_T4::can1.receiveBufferPeakCount () ;
```

Note the driver receive buffer may overflow, if messages are not retrieved (by calls of the receive or the dispatchReceivedMessage methods). If an overflow occurs, further calls of ACAN_T4::can1.receiveBufferPeakCount () return ACAN_T4::can1.receiveBufferSize ()+1.

13 Primary filters

A first step is to define *receive filters*[4]. 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*[5]. Making the difference is required by FLEXCAN hardware design: *primary filters* are more powerfull than *secondary filters*.

13.1 Primary filter example

For defining *primary filters*[6], you write:

```
void setup () {
  ACAN_T4_Settings 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_T4::can1.begin (settings,
                                                  primaryFilters, // The filter array
                                                  3) ; // Filter array size
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN_T4::can1.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[7]:

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

[4] The second step is to use the `dispatchReceivedMessage` method instead of the `receive` method, see section 15 page 26.

[5] 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*.

[6] For *secondary filters*, see section 14 page 22.

[7] There is a fourth optional argument, that is `NULL` by default – see section 15 page 26.

- the required identifier value.

Maximum number of *primary filters*. The number of *primary filters* is limited by hardware to 32.

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

13.2 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
  ...
}
```

13.3 Primary filter for matching several identifiers

A primary filter can be configured for matching several identifiers[8]. You provide two values: a `filter_mask` and a `filter_acceptance`. A message with an `identifier` is accepted if:

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

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.

13.4 Primary filter conformance

The pass-all primary filter (section 13.2 page 20) always conforms.

For a primary filter for matching several identifiers, see section 13.3 page 21.

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`.

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

If one or more primary filters do not conform, the execution of the begin method returns an error – see table 7 page 29.

13.5 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 () {
  ACAN_T4_Settings 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_T4::can1.begin (settings, primaryFilters, 3) ;
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN_T4::can1.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 15 page 26.

14 Secondary filters

Depending from the configuration, you can define up to 96 *secondary filters*.

14.1 Secondary filters, without primary filter

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

```
void setup () {
  ACAN_T4_Settings 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_T4::can1.begin (settings,
                                                  NULL, 0, // No primary filter
                                                  secondaryFilters, // The filter array
                                                  3) ; // Filter array size
  ...
void loop () {
  CANMessage message ;
  if (ACAN_T4::can1.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[9]:

- 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 by hardware to 96.

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.

14.2 Primary and secondary filters

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

```
void setup () {
  ACAN_T4_Settings settings (125 * 1000) ;
  ...
```

[9] There is a fourth optional argument, that is `NULL` by default – see section 15 page 26.

```
  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_T4::can1.begin (settings,
                                                  primaryFilters,
                                                  1, // Primary filter array size
                                                  secondaryFilters,
                                                  2) ; // Secondary filter array size
  ...
void loop () {
  CANMessage message ;
  if (ACAN_T4::can1.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.

14.3 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
  } ;
  ...
```

14.4 Secondary filter conformance

The pass-all secondary filter (section 14.3 page 24) 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`.

14.5 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 () {
  ACAN_T4_Settings 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_T4::can1.begin (settings,
                                                  primaryFilters, 1,
                                                  secondaryFilters, 2) ;
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN_T4::can1.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 15 page 26.

15 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 () {
  ACAN_T4_Settings 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_T4::can1.begin (settings,
                                                  primaryFilters, 1,
                                                  secondaryFilters, 2) ;
  ...
}

void loop () {
  ACAN_T4::can1.dispatchReceivedMessage () ; // Do not use ACAN_T4::can1.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_T4::can1.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_T4::can1.dispatchReceivedMessage (filterMatchFunction) ;
  ...
}
```

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

16 The ACAN_T4::begin method reference

16.1 The ACAN_T4::begin method prototype

The begin method prototype is:

```
uint32_t ACAN_T4::begin (const ACAN_T4_Settings & 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_T4::can1.begin (settings,
                                                primaryFilters, primaryFilterCount,
                                                secondaryFilters) ;
```

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

```
const uint32_t errorCode = ACAN_T4::can1.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_T4::can1.begin (settings, primaryFilters) ;
```

```
const uint32_t errorCode = ACAN_T4::can1.begin (settings) ;
```

16.2 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 7. An error code could report several errors. Bits from 0 to 11 are actually defined by the `ACAN_T4_Settings` class and are also returned by the `CANBitSettingConsistency` method (see section 17.3 page 34). Bits from 12 are defined by the `ACAN_T4` class.

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	`mPhaseSegment1` == 1 and *triple sampling*	
25	Inconsistent CAN Bit configuration	section 16.2.2 page 30
26	Invalid Rx pin selection	section 8.3 page 13
27	Invalid Tx pin selection	section 7.2 page 12
28	Secondary filter conformance error	section 16.3.2 page 31
30	Primary filter conformance error	section 16.3 page 30
29	Too much secondary filters	section 16.3.1 page 31
31	Too much primary filters	section 16.2.3 page 30

Table 7 – The `ACAN_T4::begin` method error codes

The `ACAN_T4_Settings` 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_T4` class defines static constant properties that can be used as mask error:

```
public: static const uint32_t kTooMuchPrimaryFilters     = 1 << 31 ;
public: static const uint32_t kNotConformPrimaryFilter   = 1 << 30 ;
public: static const uint32_t kTooMuchSecondaryFilters   = 1 << 29 ;
public: static const uint32_t kNotConformSecondaryFilter = 1 << 28 ;
```

```
public: static const uint32_t kInvalidTxPin              = 1 << 27 ;
public: static const uint32_t kInvalidRxPin              = 1 << 26 ;
public: static const uint32_t kCANBitConfiguration       = 1 << 25 ;
```

For example, you can write:

```
const uint32_t errorCode = ACAN_T4::can1.begin (settings,
                                                primaryFilters, primaryFilterCount,
                                                secondaryFilters, secondaryFilterCount) ;
if (errorCode != 0) {
  // Error(s)
  if (errorCode & ACAN_T4::kTooMuchPrimaryFilters) {
    // Error: too much primary filters
  }
  ...
}
```

16.2.1 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 `ACAN_T4_Settings` constructor cannot compute a CAN bit configuration close enough to the wished bitrate. When the `begin` is called with `settings.mBitConfigurationClosedToWishedRate` false, this error is reported. For example:

```
void setup () {
  ACAN_T4_Settings settings (1) ; // 1 bit/s !!!
  // Here, settings.mBitConfigurationClosedToWishedRate is false
  const uint32_t errorCode = ACAN_T4::can1.begin (settings) ;
  // Here, errorCode == ACAN_T4::kCANBitConfigurationTooFarFromWishedBitRateErrorMask
}
```

This error is a fatal error, the driver and the FLEXCAN module are not configured. See section 17.1 page 31 for a discussion about CAN bit setting computation.

16.2.2 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 17.3 page 34.

16.2.3 Too much primary filters error

The number of *primary filters* is limited by hardware to 32.

16.3 Primary filters conformance error

One or several primary filters do not conform: see section 13.4 page 21. Comment out primary filter definitions until finding the faulty definition.

16.3.1 Too much secondary filters error

The number of *secondary filters* is limited by hardware to 96.

16.3.2 Secondary filter conformance error

One or several secondary filters do not conform: see section 14.4 page 25. Comment out secondary filter definitions until finding the faultly definition.

17 ACAN_T4_Settings class reference

Note. The `ACAN_T4_Settings` class is not Arduino specific. You can compile it on your desktop computer with your favorite C++ compiler.

17.1 The ACAN_T4_Settings constructor: computation of the CAN bit settings

The constructor of the `ACAN_T4_Settings` has one mandatory argument: the wished bitrate. It tries to compute the CAN bit settings for this bitrate. If it succeeds, the constructed object has its `mBitConfigurationClosedToWishedRa` property set to `true`, otherwise it is set to `false`. For example:

```
void setup () {
  ACAN_T4_Settings settings (1 * 1000 * 1000) ; // 1 Mbit/s
  // Here, settings.mBitConfigurationClosedToWishedRate is true
  ...
}
```

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4_Settings settings (842 * 1000) ; // 842 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual bitrate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 124 ppm
  ...
}
```

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

By default, a wished bitrate is accepted if the distance from the computed actual bitrate is lower or equal to $1,000$ ppm $= 0.1$ %. You can change this default value by adding your own value as second argument of `ACAN_T4_Settings` constructor:

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4_Settings 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 bitrate: ") ;
  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 bitrate:

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4_Settings 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 bitrate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  500,000 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 0 ppm
  ...
}
```

With default CAN root clock settings (see section 37 page 74), the fastest exact bitrate is 3,2 Mbit/s. It works when the FLEXCAN module is configured in both *loop back* mode (section 17.8.3 page 38) and *self reception* mode (section 17.8.2 page 37). Note bitrates above 1 Mbit/s do not conform to the `ISO-11898`; CAN transceivers as `MCP2551` require the bitrate lower or equal to 1 Mbit/s.

With default CAN root clock settings (see section 37 page 74), the slowest exact bitrate is 9 375 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 bitrate to be greater or equal to 16 kbit/s. If you want to work with slower bitrates and you need a transceiver, use one without this detection, as the `PCA82C250`.

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4_Settings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual bitrate: ") ;
  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) ;
  ACAN_T4_Settings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual bitrate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 10,100 ppm
  Serial.print ("Bitrate 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 bitrate, 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) ;
  ACAN_T4_Settings 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 bitrate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  500000: ok, bitrate did not change
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

Be aware to always respect CAN bit timing consistency!

17.2 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 bitrate is given by:

$$\text{Actual bitrate} = \frac{\texttt{CANRootClockFrequency} \,/\, \texttt{CANRootClockDivisor}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mPropagationSegment} + \texttt{mPhaseSegment1} + \texttt{mPhaseSegment2})}$$

Where (see section 37 page 74):

- `CANRootClockFrequency` is either 60 MHz (default) or 24 MHz;
- `CANRootClockDivisor` is an integer in [1, 64], default is value is 1.

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

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

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

17.3 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) ;
  ACAN_T4_Settings 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 8.

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 8 – The ACAN_T4_Settings::CANBitSettingConsistency method error codes

The ACAN_T4_Settings 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 ;
```

17.4 The actualBitRate method

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4_Settings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual bitrate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  ...
}
```

Note. If CAN bit settings are not consistent (see section 17.3 page 34), the returned value is irrelevant.

17.5 The exactBitRate method

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4_Settings settings (842 * 1000) ; // 842 kbit/s
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual bitrate: ") ;
  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 17.3 page 34), the returned value is irrelevant.

With the default CAN root clock settings (60 MHz CAN root clock, CAN root clock divisor equal to 1, see section 37 page 74), there are 52 exact bit rates : 9 375 bit/s, 9 600 bit/s, 10 000 bit/s, 12 000 bit/s, 12 500 bit/s, 15 000 bit/s, 15 625 bit/s, 16 000 bit/s, 18 750 bit/s, 19 200 bit/s, 20 000 bit/s, 24 000 bit/s, 25 000 bit/s, 30 000 bit/s, 31 250 bit/s, 32 000 bit/s, 37 500 bit/s, 40 000 bit/s, 46 875 bit/s, 48 000 bit/s, 50 000 bit/s, 60 000 bit/s, 62 500 bit/s, 75 000 bit/s, 78 125 bit/s, 80 000 bit/s, 93 750 bit/s, 96 000 bit/s, 100 000 bit/s, 120 000 bit/s, 125 000 bit/s, 150 000 bit/s, 156 250 bit/s, 160 000 bit/s, 187 500 bit/s, 200 000 bit/s, 234 375 bit/s, 240 000 bit/s, 250 000 bit/s, 300 000 bit/s, 312 500 bit/s, 375 000 bit/s, 400 000 bit/s, 468 750 bit/s, 480 000 bit/s, 500 000 bit/s, 600 000 bit/s, 625 000 bit/s, 750 000 bit/s, 800 000 bit/s, 937 500 bit/s, 1 000 000 bit/s.

With the 24 MHz CAN root clock and the CAN root clock divisor equal to 1 (see section 37 page 74), there are 62 exact bit rates : 3 750 bit/s, 3 840 bit/s, 4 000 bit/s, 4 800 bit/s, 5 000 bit/s, 6 000 bit/s, 6 250 bit/s, 6 400 bit/s, 7 500 bit/s, 7 680 bit/s, 8 000 bit/s 9 375 bit/s, 9 600 bit/s, 10 000 bit/s, 12 000 bit/s, 12 500 bit/s, 12 800 bit/s, 15 000 bit/s, 15 625 bit/s, 16 000 bit/s, 18 750 bit/s, 19 200 bit/s, 20 000 bit/s, 24 000 bit/s, 25 000 bit/s, 30 000 bit/s, 31 250 bit/s, 32 000 bit/s, 37 500 bit/s, 38 400 bit/s, 40 000 bit/s, 46 875 bit/s, 48 000 bit/s, 50 000 bit/s, 60 000 bit/s, 62 500 bit/s, 64 000 bit/s, 75 000 bit/s, 80 000 bit/s, 93 750 bit/s, 96 000 bit/s, 100 000 bit/s, 120 000 bit/s, 125 000 bit/s, 150 000 bit/s, 160 000 bit/s, 187 500 bit/s, 192 000 bit/s, 200 000 bit/s, 240 000 bit/s, 250 000 bit/s, 300 000 bit/s, 320 000 bit/s, 375 000 bit/s, 400 000 bit/s, 480 000 bit/s, 500 000 bit/s, 600 000 bit/s, 750 000 bit/s, 800 000 bit/s, 960 000 bit/s, 1 000 000 bit/s.

17.6 The ppmFromWishedBitRate method

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4_Settings settings (842 * 1000) ; // 842 kbit/s
```

```
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual bitrate: ") ;
  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 17.3 page 34), the returned value is irrelevant.

17.7 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 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) ;
  ACAN_T4_Settings 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 17.3 page 34), the returned value is irrelevant.

17.8 Properties of the `ACAN_T4_Settings` class

All properties of the `ACAN_T4_Settings` class are declared `public` and are initialized (table 9). The default values of properties from `mWhishedBitRate` until `mTripleSampling` corresponds to a CAN bitrate of 250,000 bit/s.

17.8.1 The `mListenOnlyMode` property

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

17.8.2 The `mSelfReceptionMode` property

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

Property	Type	Initial value	Comment
mWhishedBitRate	uint32_t	250,000	See section 17.1 page 31
mBitRatePrescaler	uint16_t	10	See section 17.1 page 31
mPropagationSegment	uint8_t	8	See section 17.1 page 31
mPhaseSegment1	uint8_t	8	See section 17.1 page 31
mPhaseSegment2	uint8_t	7	See section 17.1 page 31
mRJW	uint8_t	4	See section 17.1 page 31
mTripleSampling	bool	false	See section 17.1 page 31
mBitConfigurationClosedToWishedRate	bool	true	See section 17.1 page 31
mListenOnlyMode	bool	false	See section 17.8.1 page 37
mSelfReceptionMode	bool	false	See section 17.8.2 page 37
mLoopBackMode	bool	false	See section 17.8.3 page 38
mTxPin	uint8_t	255	See section 8.3 page 13
mRxin	uint8_t	255	See section 7.2 page 12
mReceiveBufferSize	uint16_t	32	See section 12.1 page 18
mTransmitBufferSize	uint16_t	16	See section 9.2 page 15
mTxPinIsOpenCollector	bool	false	See section 8.2 page 13

Table 9 – Properties of the `ACAN_T4_Settings` class

17.8.3 The `mLoopBackMode` property

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

18 CAN controller state

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

18.1 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 ;
```

18.2 The `receiveErrorCounter` method

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

18.3 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.

18.4 The globalStatus method

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

This method returns a value bit field value. All bits are 0 when there is no error. The bits are described in the table 10.

Constant	Value	Comment
kGlobalStatusInitError	1 << 0	The begin method did return a not null value.
kGlobalStatusRxFIFOWarning	1 << 1	The hardware RxFIFO has at one time contained 5 or more messages. No message loss.
kGlobalStatusRxFIFOOverflow	1 << 2	The hardware RxFIFO did overflow. Message loss.
kGlobalStatusReceiveBufferOverflow	1 << 3	The driver receive buffer did overflow. Message loss.

Table 10 – The globalStatus bits

18.5 The resetGlobalStatus method

```
public : void resetGlobalStatus (const uint32_t inReset) ;
```

The inReset value is bit field. For every global status bit :

- if a bit of inReset value is 0, no effect;
- if a bit of inReset value is 1, the correspondant bit of the global status is reseted.

Note: the kGlobalStatusInitError bit (bit 0) cannot be reseted.

19 The demoCAN1CAN2CAN3 sketch

I use this sketch for testing the ACAN_T4 library. An elementary CAN network is built, that consists of the three FLEXCAN modules. Every ACAN_T4::can$i$ sends messages as quickly as possible that are received by the other two.

Hardware. Simply connect the six CTX1, CRX1, CTX2, CRX2, CTX3, CRX3 signals together, nothing more (figure 2). As there is no CAN transceiver, do not use wires that are too long, 20 cm is a maximum.

This is consistent because:

- all CTXi pins are configured in *open collector* mode;

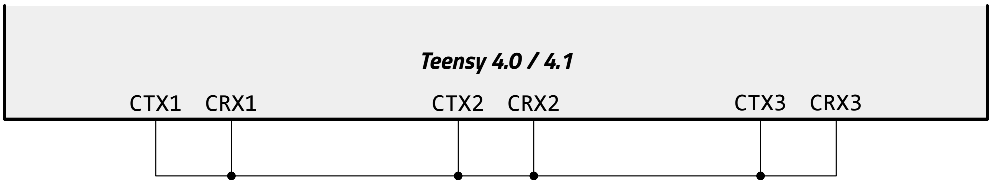

Figure 2 – Connections for the demoCAN1CAN2CAN3 sketch

- all CRXi pins are configured with the smallest pullup value, 22kΩ.

Running the sketch. Every `ACAN_T4::can`i sends 50,000 standard messages as quickly as possible. For avoiding identifier collisions, the identifiers are randomly computed as follows:

- `ACAN_T4::can1` sends standard frame with identifier equal to `((micros () % 682) * 3)`;
- `ACAN_T4::can2` sends standard frame with identifier equal to `((micros () % 682) * 3 + 1)`;
- `ACAN_T4::can3` sends standard frame with identifier equal to `((micros () % 682) * 3 + 2)`.

Note :

- $0 \leqslant ((\texttt{micros ()} \,\%\, 682) \leqslant 681$
- $0 \leqslant ((\texttt{micros ()} \,\%\, 682) * 3 \leqslant 2043$

The largest generated value is 2045, less than the maximum standard identifier value `0x7FF` = 2047.

After initialization messages, the serial monitor outputs for every CANi:

- the sent message count;
- the received message count;
- the global status (`0` if all is ok, function `globalStatus`, see section 18.4 page 39);
- the received buffer peak count (function `receiveBufferPeakCount`, see section 12.4 page 18).

```
CAN1-CAN2-CAN3 test
Bitrate: 1000000 bit/s
can1 ok
can2 ok
can3 ok
CAN1: 0 / 0 / 0 / 0, CAN2: 0 / 0 / 0 / 0, CAN3: 0 / 0 / 0 / 0
CAN1: 5877 / 7386 / 0x0 / 1, CAN2: 927 / 12336 / 0x0 / 1, CAN3: 6493 / 6770 / 0x0 / 1
CAN1: 26326 / 27834 / 0x0 / 1, CAN2: 927 / 53233 / 0x0 / 1, CAN3: 26941 / 27219 / 0x0 / 1
CAN1: 46776 / 48285 / 0x0 / 1, CAN2: 927 / 94134 / 0x0 / 1, CAN3: 47392 / 47669 / 0x0 / 1
CAN1: 50000 / 85246 / 0x0 / 1, CAN2: 35263 / 100000 / 0x0 / 1, CAN3: 50000 / 85246 / 0x0 / 1
CAN1: 50000 / 100000 / 0x0 / 1, CAN2: 50000 / 100000 / 0x0 / 1, CAN3: 50000 / 100000 / 0x0 / 1
CAN1: 50000 / 100000 / 0x0 / 1, CAN2: 50000 / 100000 / 0x0 / 1, CAN3: 50000 / 100000 / 0x0 / 1
...
```

Part II

CANFD

Only the FLEXCAN 3 module of the Teensy 4.0 / 4.1 microcontroller handles CANFD.

In short: for using FLEXCAN 3 module in CANFD mode, use the methods with the `FD` suffix:

- `beginFD` instead of `begin`;
- `tryToSendFD` instead of `tryToSend`;
- `availableFD` instead of `available`;
- `receiveFD` instead of `receive`;
- `dispatchReceivedMessageFD` instead of `dispatchReceivedMessage`.

Note the CANFD receive filter mecanism is different from CAN 2.0B.

20 Data flow

The figure 3 illustrates message flow for sending and receiving CANFD messages.

FLEXCAN3 module is hardware, integrated into the micro-controller. It implements several MBs (*Message Buffers*), used for the *data frame transmit buffer*, *remote frame transmit buffer(s)*, *reception buffers*. By default, the number of MBs is 14.

Sending CANFD messages. The FLEXCAN3 hardware makes sending data frames different from sending remote frames. For both, user code calls the `tryToSendFD` method – see section 26 page 50 for sending data frames, and section 27 page 52 for sending remote frames. The data frames are stored in the *Driver Transmit Buffer*, before to 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 26.2 page 51 for changing the default value.

Receiving CANFD messages. The FLEXCAN *CAN Protocol Engine* transmits all correct frames to the *reception filters*. By default, they are configured as pass-all, see section 13 page 19 and section 14 page 22 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 29.1 page 54 for changing the default value. Three user methods are available:

- the `availableFD` method returns `false` if the *Driver Receive Buffer* is empty, and `true` otherwise;
- the `receiveFD` method retrieves messages from the *Driver Receive Buffer* – see section 29 page 53, section 31.5 page 60;
- the `dispatchReceivedMessageFD` method if you have defined CANFD filters that name a call-back function – see section 32 page 61.

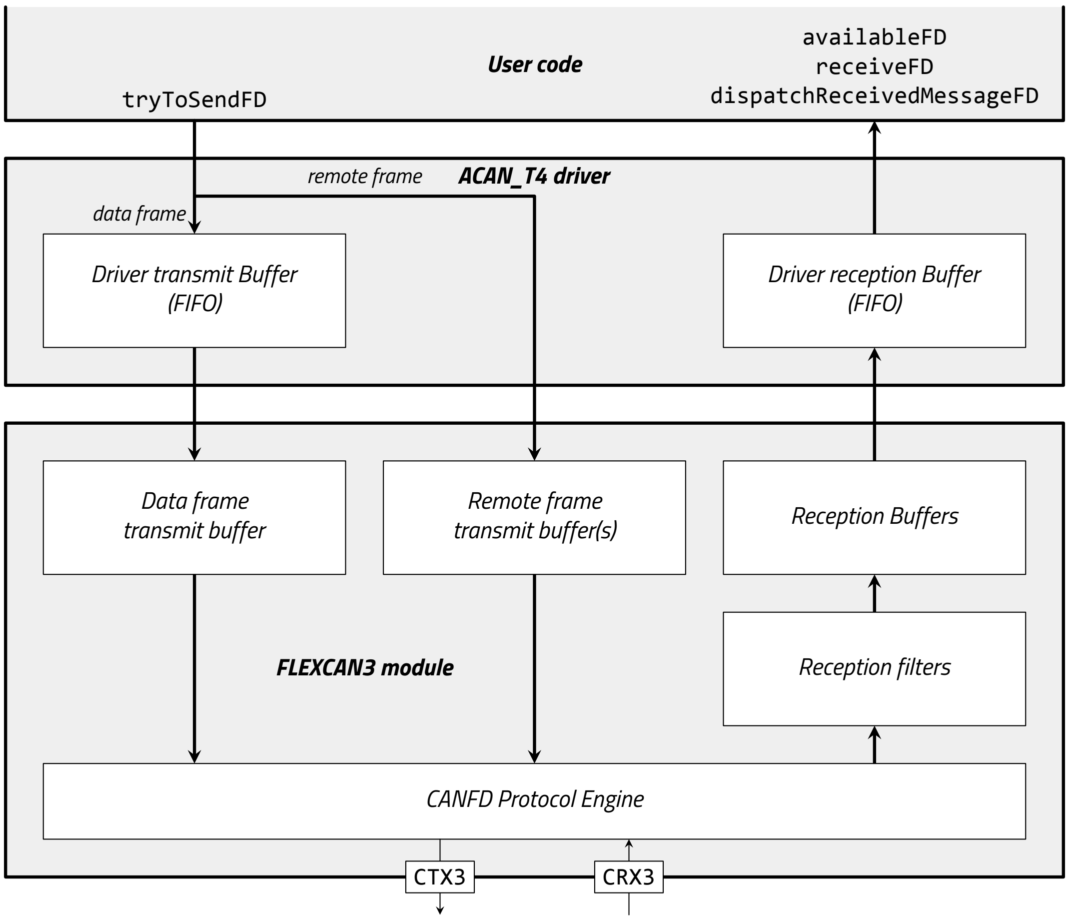

Figure 3 – Message flow in the ACAN_T4::can3 driver and FLEXCAN3 module, in CANFD mode

Sequentiality. The ACAN_T4 driver and the configuration of the FLEXCAN module ensures sequentiality of sent data messages. This means that if an user program calls tryToSendFD first for a message M_1 and then for a message M_2, the message M_1 is sent in the CANFD network before the message M_2.

21 A simple example: LoopBackDemoCAN3FD

The LoopBackDemoCAN3FD sketch is a sample code for introducing the ACAN_T4 library in CANFD mode[10]. It demonstrates how to configure the driver, to send a CANFD message, and to receive a CANFD message

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

```
#ifndef __IMXRT1062__
  #error "This sketch should be compiled for Teensy 4.0 / 4.1"
#endif

#include <ACAN_T4.h>

void setup () {
```

[10] See also the LoopBackDemoCAN3FDWithCheck sketch.

```
8     pinMode (LED_BUILTIN, OUTPUT) ;
9     Serial.begin (9600) ;
10    while (!Serial) {
11      delay (50) ;
12      digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
13    }
14    Serial.println ("CAN3FD loopback test") ;
15    ACAN_T4FD_Settings settings (125 * 1000, DataBitRateFactor::x4) ;
16    settings.mLoopBackMode = true ;
17    settings.mSelfReceptionMode = true ;
18    const uint32_t errorCode = ACAN_T4::can3.beginFD (settings) ;
19    if (0 == errorCode) {
20      Serial.println ("can3 ok") ;
21    }else{
22      Serial.print ("Error can3: 0x") ;
23      Serial.println (errorCode, HEX) ;
24    }
25  }
26
27  static uint32_t gBlinkDate = 0 ;
28  static uint32_t gSendDate = 0 ;
29  static uint32_t gSentCount = 0 ;
30  static uint32_t gReceivedCount = 0 ;
31
32  void loop () {
33    if (gBlinkDate <= millis ()) {
34      gBlinkDate += 500 ;
35      digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
36    }
37    CANFDMessage message ; // By default: standard data CANFD frame, zero length
38    if (gSendDate <= millis ()) {
39      message.id = 0x123 ;
40      const bool ok = ACAN_T4::can3.tryToSendFD (message) ;
41      if (ok) {
42        gSendDate += 2000 ;
43        gSentCount += 1 ;
44        Serial.print ("Sent: ") ;
45        Serial.println (gSentCount) ;
46      }
47    }
48    if (ACAN_T4::can3.receiveFD (messageFD)) {
49      gReceivedCount += 1 ;
50      Serial.print ("Received: ") ;
51      Serial.println (gReceivedCount) ;
52    }
53  }
```

Line 1 to 3. This ensures the Teensy 4.0 / 4.1 board is selected.

Line 5. This line includes the ACAN_T4 library.

Line 9 to 13. Start serial (the `9600` argument value is ignored by Teensy), and blink quickly until the *Arduino Serial Monitor* is opened.

Line 15. Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACAN_T4_Settings` class. The constructor has two parameters: the wished CAN arbitration bitrate, and the data bitrate factor. Here, it is `DataBitRateFactor::x4`, meaning the data bitrate is four times the arbitration bitrate. It returns a `settings` object fully initialized with CAN bit settings for the wished bitrate, and default values for other configuration properties.

Lines 16 and 17. 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 enable *loop back*, that is you can run this demo sketch even it you have no connection to a physical CAN network. The section 17.8 page 37 lists all properties you can override.

Line 18. This is the third step, configuration of the `ACAN_T4::can1` driver with `settings` values. You cannot change the `ACAN_T4::can3` name – see section 6 page 11. The driver is configured for being able to send any CAN 2.0B frame(standard / extended, data / remote frame), any CANFD frame (up to 64 data byte / frame, with or without data bitrate switch, and to receive all theses frames. If you want to define reception filters, see section 30 page 55.

Lines 19 to 24. Last step: the configuration of the `ACAN_T4::can1` driver returns an error code, stored in the `errorCode` constant. It has the value 0 if all is ok – see section 16.2 page 29.

Line 27. The `gBlinkDate` global variable is used for blinking Teensy LED every 0.5 s.

Line 28. The `gSendDate` global variable is used for sending a CAN message every 2 s.

Line 29. The `gSentCount` global variable counts the number of sent messages.

Line 30. The `gReceivedCount` global variable counts the number of received messages.

Line 33 to 36. Blink Teensy LED.

Line 37. 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, sent with bitrate switch – see section 22 page 45.

Line 38. It tests if it is time to send a message.

Line 39. Set the message identifier. In a real code, we set here message data, and for an extended frame the `ext` boolean property.

Line 40. 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 `tryToSendFD` 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 CANFD network.

Lines 41 to 46. 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 48. As the FLEXCAN3 module is configured in *loop back* mode (see lines 16 and 17), all sent messages are received. The `receiveFD` 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 49 to 51. It a message has been received, the `gReceivedCount` is incremented and displayed.

22 The CANFDMessage class

Note. The CANFDMessage class is declared in the CANFDMessage.h header file. The class declaration is protected by an include guard that causes the macro GENERIC_CANFD_MESSAGE_DEFINED to be defined. This allows an other library, as the ACAN2717FD library, to freely include this file without any declaration conflict.

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

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

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

22.1 Properties

```
class CANFDMessage {
  ...
  public : uint32_t id;  // Frame identifier
  public : bool ext ; // false -> base frame, true -> extended frame
  public : Type type ;
  public : uint8_t idx ;  // Used by the driver
  public : uint8_t len ;  // Length of data (0 ... 64)
  public : union {
    uint64_t data64    [ 8] ; // Caution: subject to endianness
    int64_t  data_s64  [ 8] ; // Caution: subject to endianness
    uint32_t data32    [16] ; // Caution: subject to endianness
    int32_t  data_s32  [16] ; // Caution: subject to endianness
    float    dataFloat [16] ; // Caution: subject to endianness
    uint16_t data16    [32] ; // Caution: subject to endianness
    int16_t  data_s16  [32] ; // Caution: subject to endianness
    int8_t   data_s8   [64] ;
    uint8_t  data      [64] ;
  } ;
  ...
} ;
```

Note the message datas are defined by an **union**. So message datas can be seen as 64 bytes, 32 x 16-bit unsigned integers, 16 x 32-bit, or 8 x 64-bit. Be aware that multi-byte integers are subject to endianness (Cortex M7 processors of Teensy 4.x are little-endian).

22.2 The default constructor

All properties are initialized by default, and represent a base data frame, with an identifier equal to 0, and without any data (table 11).

Property	Initial value	Comment
id	0	
ext	false	Base frame
type	CANFD_WITH_BIT_RATE_SWITCH	CANFD frame, with bitrate switch
idx	0	
len	0	No data
data	–	*unitialized*

Table 11 – CANFDMessage default constructor initialization

22.3 Constructor from CANMessage

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

All properties are initialized from the `inCANMessage` (table 12). Note that only `data64[0]` is initialized from `inCANMessage.data64`.

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

Table 12 – CANFDMessage constructor CANMessage

22.4 The type property

Its value is an instance of an enumerated type:

```
class CANFDMessage {
...
public: typedef enum : uint8_t {
  CAN_REMOTE,
  CAN_DATA,
  CANFD_NO_BIT_RATE_SWITCH,
  CANFD_WITH_BIT_RATE_SWITCH
} Type ;
...
} ;
```

The `type` property specifies the frame format, as indicated in the table 13.

type property	Meaning	Constraint on len
CAN_REMOTE	CAN 2.0B remote frame	0 ... 8
CAN_DATA	CAN 2.0B data frame	0 ... 8
CANFD_NO_BIT_RATE_SWITCH	CANFD frame, no bitrate switch	0 ... 8, 12, 16, 20, 24, 32, 48, 64
CANFD_WITH_BIT_RATE_SWITCH	CANFD frame, bitrate switch	0 ... 8, 12, 16, 20, 24, 32, 48, 64

Table 13 – CANFDMessage `type` property

22.5 The `len` property

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

22.6 The `idx` property

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

- for a received message, it contains the acceptance filter index (see section 32 page 61);
- it is not used for on sending messages.

22.7 The `pad` method

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

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

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

22.8 The `isValid` method

```
bool CANFDMessage::isValid (void) const ;
```

Not all settings of `CANFDMessage` instances represent a valid frame. For example, there is no CANFD remote frame, so a remote frame should have its length lower than or equal to 8. There is no constraint on extended / base identifier (`ext` property).

The `isValid` returns `true` if the contraints on the `len` property are checked, as indicated the table 13 page 47, and `false` otherwise.

23 Driver instance

For using CAN3 in CANFD mode, you use the `ACAN_T4::can3` variable, as for CAN2.0B.

24 CRX3 pin configuration

You can change CRX3 pin following setting:

- its input impedance (section 7.1 page 11, 47kΩ pullup by default);

FLEXCAN3 of Teensy 4.0 / 4.1 does not support alternate pins.

24.1 Input impedance

An input pin of the Teensy 4.0 / 4.1 micro-controller has different pullup / pulldown configurations. Five settings are available:

```
class ACAN_T4_Settings {
  ...
  public: typedef enum : uint8_t {
    NO_PULLUP_NO_PULLDOWN = 0, // PUS = 0, PUE = 0, PKE = 0
    PULLDOWN_100k = 0b0011, // PUS = 0, PUE = 1, PKE = 1
    PULLUP_47k     = 0b0111, // PUS = 1, PUE = 1, PKE = 1
    PULLUP_100k    = 0b1011, // PUS = 2, PUE = 1, PKE = 1
    PULLUP_22k     = 0b1111  // PUS = 3, PUE = 1, PKE = 1
  } RxPinConfiguration ;
  ...
} ;
```

By default, PULLUP_47k is selected. For setting an other value, write for example:

```
  settings.mRxPinConfiguration = ACAN_T4_Settings::PULLUP_100k ;
```

25 CTX3 pin configuration

You can change CTX3 pin following settings:

- its output impedance (section 8.1 page 12, 78Ω by default);
- push/pull or open collector (section 8.2 page 13);

FLEXCAN3 of Teensy 4.0 / 4.1 does not support alternate pins.

25.1 Output impedance

An output pin of the Teensy 4.0 / 4.1 micro-controller has a programmable output impedance. Seven settings are available[11]:

Symbol	Typical value at 3.3V
ACAN_T4_Settings::IMPEDANCE_R0	157 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_2	78 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_3	53 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_4	39 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_5	32 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_6	26 Ω
ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_7	23 Ω

Table 14 – GPIO output buffer average impedance, 3.3 V

Theses settings are defined by an enumerated type:

```
class ACAN_T4_Settings {
  ...
  public: typedef enum {
    IMPEDANCE_R0 = 1,
    IMPEDANCE_R0_DIVIDED_BY_2 = 2,
    IMPEDANCE_R0_DIVIDED_BY_3 = 3,
    IMPEDANCE_R0_DIVIDED_BY_4 = 4,
    IMPEDANCE_R0_DIVIDED_BY_5 = 5,
    IMPEDANCE_R0_DIVIDED_BY_6 = 6,
    IMPEDANCE_R0_DIVIDED_BY_7 = 7
  } TxPinOutputBufferImpedance ;
  ...
} ;
```

By default, `IMPEDANCE_R0_DIVIDED_BY_2` is selected. For setting an other value, write:

```
  settings.mTxPinOutputBufferImpedance = ACAN_T4_Settings::IMPEDANCE_R0_DIVIDED_BY_7;
```

25.2 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 15 – `Tx` pin output, following the `mTxPinIsOpenCollector` property setting

[11] *i.MX RT1060 Crossover Processors for Consumer Products*, IMXRT1060CEC, Rev. 0.1, 04/2019, Table 27 page 38.

26 Sending CAN2.0B and CANFD data frames

Note. This section applies only to **data** frames. For sending remote frames, see section 27 page 52. The `type` property should have one of the following values:

- `CANFDMessage::CAN_DATA` (sending a CAN 2.0B data frame);
- `CANFDMessage::CANFD_NO_BIT_RATE_SWITCH` (sending a CANFD frame, without bitrate switch);
- `CANFDMessage::CANFD_WITH_BIT_RATE_SWITCH` (sending a CANFD frame, with bitrate switch).

26.1 `tryToSendFD` for sending data frames

Call the method `tryToSendFD` 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_T4::can3.tryToSendFD (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 () {
  CANFDMessage message ;
  if (gSendDate < millis ()) {
    // Initialize message properties
    const bool ok = ACAN_T4::can3.tryToSendFD (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) {
    CANFDMessage message ;
    // Initialize message properties
    const bool ok = ACAN_T4::can3.tryToSendFD (message) ;
    if (ok) {
      gSendMessage = false ;
    }
  }
  ...
}
```

26.2 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:

```
ACAN_T4FD_Settings settings (125 * 1000, DataBitRateFactor::x2) ;
settings.mTransmitBufferSize = 30 ;
const uint32_t errorCode = ACAN_T4::can3.begin (settings) ;
...
```

As the size of `CANFDMessage` class is 80 bytes, the actual size of the driver transmit buffer is the value of `settings.mTransmitBufferSize * 80`.

26.3 The `transmitBufferSize` method

The `transmitBufferSize` method returns the size of the driver transmit buffer, that is the value of the `settings.mTransmitBufferSize` property.

```
const uint32_t s = ACAN_T4::can3.transmitBufferSize () ;
```

26.4 The `transmitBufferCount` method

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

```
const uint32_t n = ACAN_T4::can3.transmitBufferCount () ;
```

26.5 The `transmitBufferPeakCount` method

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

```
const uint32_t max = ACAN_T4::can3.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 `tryToSendFD` have always returned `true`.

27 Sending remote frames in CANFD mode

Note. This section applies only to **remote** frames. For sending data frames, see section 26 page 50.

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

However, for sending remote frames, you also invoke the `tryToSendFD` method. This method understands if a remote frame should be sent, the `type` property of its argument is equal to `CANFDMessage::CAN_REMOTE`.

You should set this value, the `type` property value is `CANFDMessage::CANFD_WITH_BIT_RATE_SWITCH` by default.

```
CANFDMessage message ;
message.type = CANFDMessage::CAN_REMOTE ; // Remote frame
...
const bool sent = ACAN_T4::can3.tryToSendFD (message) ;
...
```

28 Sending frames using the `tryToSendReturnStatusFD` method

```
uint32_t ACAN_T4::tryToSendReturnStatusFD (const CANFDMessage & inMessage) ;
```

This method is functionally identical to the `tryToSendFD` method, the only difference is the detailled return status:

- 0 if message has been successfully submitted (the call to the `tryToSendFD` method would have returned `true`);
- non zero if message has not been successfully submitted (the call to the `tryToSendFD` method would have returned `false`).

A non-zero return value is a bit field that details the error, as listed in table 16.

Bit Index	Constant	Comment
0	`kTransmitBufferOverflow`	Trying to send a data frame, but the transmit buffer is full (retry later).
1	`kNoAvailableMBForSendingRemoteFrame`	Trying to send a remote frame, but currently there is no available Message Buffer (retry later).
2	`kNoReservedMBForSendingRemoteFrame`	Trying to send a remote frame, but there is no dedicaced Message Buffer for sending remote frames, due to `mRxCANFDMBCount` value (permanent error).
3	`kMessageLengthExceedsPayload`	Trying to send a data frame, but frame length is greater than the length allowed by `mPayload`
4	`kFlexCANinCAN20BMode`	CAN3 is in CAN 2.0B mode, not CANFD mode.

Table 16 – `tryToSendReturnStatusFD` method returned status bits

29 Retrieving received messages using the `receiveFD` method

There are two ways for retrieving received messages :

- using the `receiveFD` method, as explained in this section;
- using the `dispatchReceivedMessageFD` method (see section 32 page 61).

This is a basic example:

```
void setup () {
  ACAN_T4FD_Settings settings (125 * 1000, DataBitRateFactor::x2) ;
  ...
  const uint32_t errorCode = ACAN_T4::can3.begin (settings) ; // No receive filter
  ...
}

void loop () {
  CANFDMessage message ;
  if (ACAN_T4::can1.receiveFD (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 () {
  ACAN_T4FD_Settings settings (125 * 1000, DataBitRateFactor::x2) ;
  ...
  const uint32_t errorCode = ACAN_T4::can3.begin (settings) ; // No receive filter
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN_T4::can3.receive (message)) {
    if ((message.type == CANFDMessage::CAN_REMOTE)
       && message.ext && (message.id == 0x123456)) {
      handle_myMessage_0 (message); // Extended remote CAN frame, id is 0x123456
    }else if ((message.type == CANFDMessage::CAN_DATA)
       && !message.ext && (message.id == 0x234)) {
      handle_myMessage_1 (message); // Standard data CAN frame, id is 0x234
    }else if ((message.type == CANFDMessage::CANFD_WITH_BIT_RATE_SWITCH)
       && !message.ext && (message.id == 0x542)) {
```

```
      handle_myMessage_2 (message); // Standard CANFD frame, id is 0x542
    }
  }
  ...
}
```

The `handle_myMessage_0` function has the following header:

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

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

29.1 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:

```
ACAN_T4_Settings settings (125 * 1000) ;
settings.mReceiveBufferSize = 100 ;
const uint32_t errorCode = ACAN_T4::can3.begin (settings) ;
...
```

29.2 The `receiveBufferSize` method

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

```
const uint32_t s = ACAN_T4::can3.receiveBufferSize () ;
```

29.3 The `receiveBufferCount` method

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

```
const uint32_t n = ACAN_T4::can3.receiveBufferCount () ;
```

29.4 The `receiveBufferPeakCount` method

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

```
const uint32_t max = ACAN_T4::can3.receiveBufferPeakCount () ;
```

Note the driver receive buffer may overflow, if messages are not retrieved (by calling the `receiveFD` method or the `dispatchReceivedMessageFD` method). If an overflow occurs, further calls of `ACAN_T4::can3.receiveBufferPeakCount ()` return `ACAN_T4::can3.receiveBufferSize ()`+1.

30 CANFD receive filters

A first step is to define *receive filters*[12]. Note the CANFD filters are very different from CAN *primary filters* (section 13 page 19) and CAN *secondary filters* (section 14 page 22). Let me explain why.

The *CANFD/FlexCAN3* chapter of the reference manual[13] presents a wonderful *Enhanced Rx FIFO*[14]. It stores up to 32 CANFD messages, and provides 128 32-bit registers for defining receive filters. Unfortunately, it doesn't work. Trying to access one of the dedicaced registers crashes the microcontroller. There are several posts relating this bug:

- *IMXRT1062 Hardfault Reading CAN3 ERFCR Register*, `https://community.nxp.com/thread/503656`
- `https://forum.pjrc.com/threads/54711-Teensy-4-0-First-Beta-Test/page119`
- ...

I haven't found a single post that explains how to do it. And surprisingly, this bug is not mentioned in the *Chip Errata* document[15]. So forget the *Enhanced Rx FIFO*.

Using the *Legacy Rx FIFO*? The section 44.4.8 page 2721 says *Legacy Rx FIFO must not be enabled when CAN FD feature is enabled.* So forget the *Legacy Rx FIFO* for CANFD: it works for CAN, but not for CANFD.

So we should use the legacy legacy way, filtering is done per receive *Message Buffer*.

30.1 Message Buffers in CANFD mode

First, we should present how the Message Buffers are handled in CANFD mode. The reference manual announces the chip implements 64 Message Buffers for FlexCAN3, however it is true only in CAN 2.0B mode.

We can consider that 2 blocks of 512 bytes of double-access RAM are reserved for Message Buffers. Theses blocks can be read and written by the CPU and by the CANFD protocol engine. A Message Buffer contains message data, identifier, and a control word[16]. In CAN 2.0B, the Message Buffer size is 16 bytes, so we have 64 Message Buffers. But in CANFD, a message can have up to 64 data bytes, so the Message Buffer size is up to 72 bytes, so the Message Buffer count goes down to 14.

30.2 The `mPayload` property

The `mPayload` of the `ACAN_T4FD_Settings` class sets the message maximum data size that the library can handle. This allows you to adjust the size of your Message Buffers according to the size of the messages in your application.

```
class ACAN_T4FD_Settings {
  ...
```

[12] The second step is to use the `dispatchReceivedMessageFD` method instead of the `receiveFD` method, see section 32 page 61.
[13] *i.MX RT1060 Processor Reference Manual*, Rev. 1, 12/2018, chapter 44, pages 2691-2846.
[14] section 44.4.7, page 2716.
[15] *Chip Errata for the i.MX RT1060*, Document Number: IMXRT1060CE, Rev. 1, 06/2019.
[16] See the reference manual, section 44.6.3, page 2829.

```
  public : typedef enum : uint8_t {
    PAYLOAD_8_BYTES  = 0,
    PAYLOAD_16_BYTES = 1,
    PAYLOAD_32_BYTES = 2,
    PAYLOAD_64_BYTES = 3
  } Payload ;
  ...
  public : Payload mPayload = PAYLOAD_64_BYTES ;
  ...
} ;
```

For example, if your application has no message with more than 32 bytes, you can set the `mPayload` property to `ACAN_T4FD_Settings::PAYLOAD_32_BYTES` : the Message Buffer count becomes 24. The table 17 gives the Message Buffer count according to the `mPayload` property.

By default, the `mPayload` property is set to `ACAN_T4FD_Settings::PAYLOAD_64_BYTES`, enabling send and receive CANFD frame of any size.

mPayload property value	Message Buffer size	Message Buffer count	mRxCANFDMBCount property range
PAYLOAD_8_BYTES	16 bytes	64	1 ... 62
PAYLOAD_16_BYTES	24 bytes	42	1 ... 40
PAYLOAD_32_BYTES	40 bytes	24	1 ... 22
PAYLOAD_64_BYTES (default)	72 bytes	14	1 ... 12

Table 17 – Available Message Buffer count according to the `mPayload` property

An Message Buffer can be used for:

- reception;
- sending a remote frame;
- sending a data frame.

30.3 The `MBCount` function

```
uint32_t MBCount (const ACAN_T4FD_Settings::Payload inPayload) ;
```

The `MBCount` standalone function is declared in the `ACAN_T4FD_Settings` header file. It returns the available Message Buffer count, according to a given payload, as shown in the table 17.

30.4 The `mRxCANFDMBCount` property

The `mRxCANFDMBCount` of the `ACAN_T4FD_Settings` class specifies the number of Message Buffers dedicaced to reception. Its valid ranges is one to the number of available Message Buffers minus two (see table 17); its default value is 11; its range depends from the `mPayload` property value.

The figure 4 shows the Message Buffer assignment, according to the `mRxCANFDMBCount` property value and the number of available Message Buffers:

- the Message Buffer #0 is always unused, as recommended in *Chip Errata for the i.MX RT1060*, section `ERR005829`, page 8;
- the last available Message Buffer is dedicated for sending data frames.

If your application does not send remote frames, it is safe to set the `mRxCANFDMBCount` property to the number of available Message Buffers minus two.

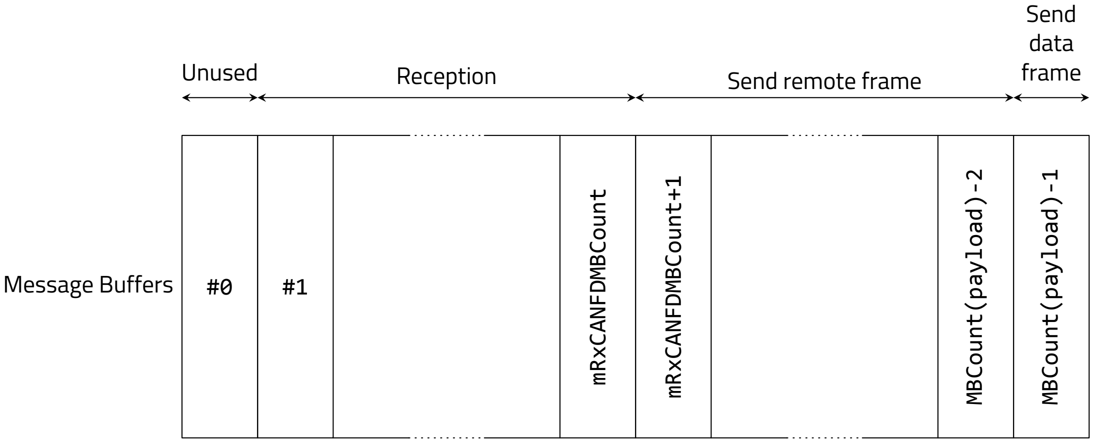

Figure 4 – FLEXCAN3 module Message Buffer assignment, in CANFD mode

By default, FLEXCAN3 is configured with 12 Message Buffers available for reception, 1 Message Buffer for sending remote data frames, and 1 Message Buffer for sending data frames (figure 5).

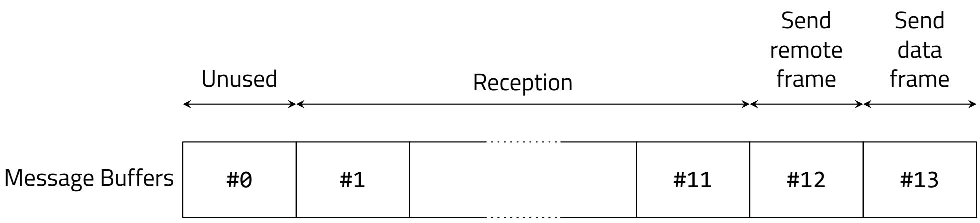

Figure 5 – FLEXCAN3 module Message Buffer default assignment, in CANFD mode

30.5 CANFD filters

To each Message Buffer in reception is associated a filter.

By default, each Message Buffer receives a pass-all filter, that is every frame received by the protocol engine can be assigned to any reception Message Buffer. More precisely, the matching process is:

1. the matching process starts with Message Buffer #1, until the `mRxCANFDMBCount`th Message Buffer;
2. if a Message Buffer is *empty* and its filter accepts the incomming frame, this frame is written to the Message buffer that becomes *full*;
3. if all the Message Buffers whose filter accepts the incoming frame are full, the last one is overwritten by the incoming frame; the previous message is lost.

If your application has somewhere an interrupt routine that lasts longer than the duration of receiving a CANFD frame, the FLEXCAN3 interrupt routine may not be able to release a Message Buffer until a new message arrives. If the reception filter is set only once, a message may be lost.

It is therefore consistent to define the same filter several times. It is very different from the CAN filters (section 13 page 19 and section 14 page 22).

31 Defining CANFD filters

The user can define up to `mRxCANFDMBCount` different filters. However, internally the library *always* defines `mRxCANFDMBCount` filters:

- if the user provides no filter, the pass-all filter is assigned to every reception Message Buffer;
- if the user provides exactly `mRxCANFDMBCount` filters, the first one is assigned to Message Buffer #1, ..., the last one is assigned to the `mRxCANFDMBCount`th Message Buffer;
- if the user provides less than `mRxCANFDMBCount` filters, the last filter is assigned to the remaining reception Message Buffers.

A filter acts on:

- remote / data information;
- standard / extended information;
- identifier value.

Note a filter cannot distinguish CANFD frames from CAN 2.0B frames.

31.1 CANFD filter example

In the following example, the `mRxCANFDMBCount` property has its default value (11). Note the two first filters have been duplicated.

```
void setup () {
  ACAN_T4FD_Settings settings (125 * 1000, DataBitRateFactor::x4) ;
  ...
  const ACANFDFilter filters [] = {
    ACANFDFilter (kData, kExtended, 0x123456), // Assigned to MB #1
    ACANFDFilter (kData, kExtended, 0x123456), // Assigned to MB #2
    ACANFDFilter (kData, kStandard, 0x234),    // Assigned to MB #3
    ACANFDFilter (kData, kStandard, 0x234),    // Assigned to MB #5
    ACANFDFilter (kRemote, kStandard, 0x542)   // Assigned to MB #6, ..., MB #11
  } ;
  const uint32_t errorCode = ACAN_T4::can3.beginFD (settings,
                                                    filters, // The filter array
                                                    5) ; // Filter array size
```

```
  ...
}

void loop () {
  CANFDMessage message ;
  if (ACAN_T4::can3.receiveFD (message)) { // Only frames that pass a filter are retrieved
    if ((message.type != CANFDMessage::CAN_REMOTE)
        && message.ext && (message.id == 0x123456)) {
      handle_myMessage_0 (message) ; // Extended data frame, id is 0x123456
    }else if ((message.type != CANFDMessage::CAN_REMOTE)
        && !message.ext && (message.id == 0x234)) {
      handle_myMessage_1 (message) ;  // Standard data frame, id is 0x234
    }else if ((message.type == CANFDMessage::CAN_REMOTE)
        && !message.ext && (message.id == 0x542)) {
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
    }
  }
  ...
}
```

Note there is a better way to handle received messages, with the `dispatchReceivedMessageFD` method, see section 32 page 61.

31.2 CANFD filter as pass-all filter

You can specify a CANFD filter that matches any frame:

```
    ACANFDFilter ()
```

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

```
void setup () {
  ...
  const ACANFDFilter primaryFilters [] = {
    ACANFDFilter (kData, kExtended, 0x123456), // Filter #0 -> MB #1
    ACANFDFilter (kData, kStandard, 0x234),    // Filter #1 -> MB #2
    ACANFDFilter (kRemote, kStandard, 0x542),  // Filter #2 -> MB #3
    ACANFDFilter ()                            // Filter #3 -> MB #4 to MB #11
  } ;
  ...
}
```

Note if a message that matches the #0 filter can be assigned to Message Buffer #4 to Message Buffer #11 if the Message Buffers #1 is full. And the same goes for #1 and #2 filters.

31.3 CANFD filter for matching several identifiers

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

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

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:

```
...
ACANFDFilter (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.

31.4 CANFD filter conformance

The pass-all primary filter (section 31.2 page 59) always conforms. For a filter for matching several identifiers, see section 31.3 page 59. For a filter for one single identifier:

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

If one or CANFD filters do not conform, the execution of the `beginFD` method returns an error – see table 18 page 64.

31.5 The `receiveFD` method revisited

The `receiveFD` method retrieves a received message. The value of the `idx` property of the `message` is the receiving Message Buffer index minus one. For example:

```
void setup () {
  ACAN_T4FD_Settings settings (125 * 1000, DataBitRateFactor::x4) ;
```

```
  ...
  const ACANFDFilter filters [] = {
    ACANFDFilter (kData, kExtended, 0x123456), // Filter #0 -> MB #1
    ACANFDFilter (kData, kStandard, 0x234),    // Filter #1 -> MB #2
    ACANFDFilter (kRemote, kStandard, 0x542)   // Filter #2 -> MB #3 to MB #11
  } ;
  const uint32_t errorCode = ACAN_T4::can3.begin (settings, filters, 3) ;
  ...
}

void loop () {
  CANFDMessage message ;
  if (ACAN_T4::can3.receiveFD (message)) { // Only frames that pass a filter are retrieved
    switch (message.idx) {
    case 0: // MB #1 match
      handle_myMessage_0 (message) ; // Extended data frame, id is 0x123456
      break ;
    case 1: // MB #2 match
      handle_myMessage_1 (message) ;  // Standard data frame, id is 0x234
      break ;
    default: // MB #3 to MB #11 match
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
      break ;
    }
  }
  ...
}
```

An improvement is to use the `dispatchReceivedMessageFD` method – see section 32 page 61.

32 The `dispatchReceivedMessageFD` method

The last improvement is to call the `dispatchReceivedMessageFD` method – do not call the `receiveFD` method any more. You can use it if you have defined CANFD filters that name a call-back function.

The CANFD 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 `dispatchReceivedMessageFD` method, you add to each filter definition as last argument the function that will handle the message. In the `loop` function, call the `dispatchReceivedMessageFD` method: it dispatches the messages to the call back functions.

```
void setup () {
  ACAN_T4FD_Settings settings (125 * 1000, DataBitRateFactor::x4) ;
  ...
  const ACANFDFilter filters [] = {
    ACANFDFilter (kData, kExtended, 0x123456, handle_myMessage_0), // Filter #0
    ACANFDFilter (kData, kStandard, 0x234, handle_myMessage_1),    // Filter #1
    ACANFDFilter (kRemote, kStandard, 0x542, handle_myMessage_2)   // Filter #2
  } ;
  const uint32_t errorCode = ACAN_T4::can3.beginFD (settings,
                                                    filters, // The filter array
```

```
                                                          3) ; // Filter array size
  ...
}

void loop () {
  ACAN_T4::can3.dispatchReceivedMessageFD () ; // Do not use ACAN_T4::can3.receiveFD any more
  ...
}
```

The `dispatchReceivedMessageFD` 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_T4::can3.dispatchReceivedMessageFD ()) {
  }
  ...
}
```

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 ACANFDFilter filters [] = {
    ACANFDFilter (kData, kExtended, 0x123456, handle_myMessage_0), // Filter #0
    ACANFDFilter (kData, kStandard, 0x234),                        // Filter #1
    ACANFDFilter (kRemote, kStandard, 0x542, handle_myMessage_2)   // Filter #2
  } ;
  ...
}
```

The `dispatchReceivedMessageFD` 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_T4::can3.dispatchReceivedMessageFD (filterMatchFunction) ;
  ...
}
```

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

Note the filter index is the matching Message Buffer index minus one, in order to have a zero-based number.

As the library always defines `mRxCANFDMBCount` filters, the filter index value goes from `0` to `mRxCANFDMBCount-1`.

33 The ACAN_T4::beginFD method reference

33.1 The ACAN_T4::beginFD method prototype

The beginFD method prototype is:

```
uint32_t ACAN_T4::beginFD (const ACAN_T4_Settings & inSettings,
                           const ACANFDFilter inFilters [] = NULL,
                           const uint32_t inFilterCount = 0) ;
```

The two last arguments have default values.

Omitting the last two arguments makes no user filter is defined, all messages are received:

```
const uint32_t errorCode = ACAN_T4::can3.beginFD (settings) ;
```

33.2 The error code

The beginFD method returns an error code. The value 0 denotes no error. Otherwise, you consider every bit as an error flag, as described in table 18. An error code could report several errors. Bits from 0 to 11 are actually defined by the ACAN_T4_Settings class and are also returned by the CANFDBitSettingConsistency method (see section 34.2 page 68). Bits from 12 are defined by the ACAN_T4 class.

The ACAN_T4FD_Settings 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 kBitRatePrescalerIsGreaterThan1024               = 1 <<  1 ;
public: static const uint32_t kArbitrationPropagationSegmentIsZero             = 1 <<  2 ;
public: static const uint32_t kArbitrationPropagationSegmentIsGreaterThan64    = 1 <<  3 ;
public: static const uint32_t kArbitrationPhaseSegment1IsZero                  = 1 <<  4 ;
public: static const uint32_t kArbitrationPhaseSegment1IsGreaterThan32         = 1 <<  5 ;
public: static const uint32_t kArbitrationPhaseSegment2IsLowerThan2            = 1 <<  6 ;
public: static const uint32_t kArbitrationPhaseSegment2IsGreaterThan32         = 1 <<  7 ;
public: static const uint32_t kArbitrationRJWIsZero                            = 1 <<  8 ;
public: static const uint32_t kArbitrationRJWIsGreaterThan32                   = 1 <<  9 ;
public: static const uint32_t kArbitrationRJWIsGreaterThanPhaseSegment2        = 1 << 10 ;
public: static const uint32_t kArbitrationPhaseSegment1Is1AndTripleSampling    = 1 << 11 ;
public: static const uint32_t kDataPropagationSegmentIsZero                    = 1 << 12 ;
public: static const uint32_t kDataPropagationSegmentIsGreaterThan32           = 1 << 13 ;
public: static const uint32_t kDataPhaseSegment1IsZero                         = 1 << 14 ;
public: static const uint32_t kDataPhaseSegment1IsGreaterThan8                 = 1 << 15 ;
public: static const uint32_t kDataPhaseSegment2IsLowerThan2                   = 1 << 16 ;
public: static const uint32_t kDataPhaseSegment2IsGreaterThan8                 = 1 << 17 ;
public: static const uint32_t kDataRJWIsZero                                   = 1 << 18 ;
public: static const uint32_t kDataRJWIsGreaterThan8                           = 1 << 19 ;
public: static const uint32_t kDataRJWIsGreaterThanPhaseSegment2               = 1 << 20 ;
```

The ACAN_T4 class defines static constant properties that can be used as mask error:

```
public: static const uint32_t kCANBitConfiguration                = 1 << 25 ;
public: static const uint32_t kCANFDNotAvailableOnCAN1AndCAN2     = 1 << 24 ;
public: static const uint32_t kTooMuchCANFDFilters                = 1 << 23 ;
public: static const uint32_t kCANFDInvalidRxMBCountVersusPayload = 1 << 22 ;
```

Bit number	Comment	Link
0	`mBitRatePrescaler` == 0	
1	`mBitRatePrescaler` > 1024	
2	`mArbitrationPropagationSegment` == 0	
3	`mArbitrationPropagationSegment` > 64	
4	`mArbitrationPhaseSegment1` == 0	
5	`mArbitrationPhaseSegment1` > 32	
6	`mArbitrationPhaseSegment2` == 0	
7	`mArbitrationPhaseSegment2` > 32	
8	`mArbitrationRJW` == 0	
9	`mArbitrationRJW` > 32	
10	`mArbitrationRJW` > `mArbitrationPhaseSegment2`	
11	`mArbitrationPhaseSegment1` == 1 and *triple sampling*	
12	`mDataPropagationSegment` == 0	
13	`mDataPropagationSegment` > 32	
14	`mDataPhaseSegment1` == 0	
15	`mDataPhaseSegment1` > 8	
16	`mDataPhaseSegment2` < 2	
17	`mDataPhaseSegment2` > 8	
18	`mDataRJW` == 0	
19	`mDataRJW` > 32	
20	`mDataRJW` > `mArbitrationPhaseSegment2`	
22	CANFD is not available on CAN1 and CAN2	
23	More than `mRxCANFDMBCount` CANFD filters	
24	Invalid `mRxCANFDMBCount` setting	
25	Inconsistent CAN Bit configuration	section 33.2.2 page 65

Table 18 – The `ACAN_T4::beginFD` method error codes

33.2.1 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 `ACAN_T4_Settings` constructor cannot compute a CAN bit configuration close enough to the wished bitrate. When the `begin` is called with `settings.mBitConfigurationClosedToWishedRate` false, this error is reported. For example:

```
void setup () {
  ACAN_T4_Settings settings (1) ; // 1 bit/s !!!
  // Here, settings.mBitConfigurationClosedToWishedRate is false
  const uint32_t errorCode = ACAN_T4::can1.begin (settings) ;
  // Here, errorCode == ACAN_T4::kCANBitConfigurationTooFarFromWishedBitRateErrorMask
}
```

This error is a fatal error, the driver and the FLEXCAN module are not configured. See section 17.1 page 31 for a discussion about CAN bit setting computation.

33.2.2 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 34.2 page 68.

34 ACAN_T4FD_Settings class reference

Note. The `ACAN_T4FD_Settings` class is not Arduino specific. You can compile it on your desktop computer with your favorite C++ compiler.

34.1 The `ACAN_T4FD_Settings` constructor: computation of the CAN bit settings

The constructor of the `ACAN_T4FD_Settings` has two mandatory arguments:

1. the wished arbitration bitrate;
2. the data bitrate factor.

It tries to compute the CANFD bit settings for theses argument values. If it succeeds, the constructed object has its `mBitConfigurationClosedToWishedRate` property set to `true`, otherwise it is set to `false`. For example:

```
void setup () {
  ACAN_T4FD_Settings settings (1 * 1000 * 1000, // Arbitration bitrate: 1 Mbit/s
                              DataBitRateFactor::x4) ; // Data bitrate: 4 Mbit/s
  // Here, settings.mBitConfigurationClosedToWishedRate is true
  ...
}
```

The `DataBitRateFactor` enumeration type is declared in the `ACANFD_DataBitRateFactor.h` file:

```
enum class DataBitRateFactor : uint8_t {
    x1 = 1,
    x2 = 2,
    x3 = 3,
    x4 = 4,
    x5 = 5,
    x6 = 6,
    x7 = 7,
    x8 = 8,
    x9 = 9,
    x10 = 10
} ;
```

Of course, CAN bit computation always succeeds for classical arbitration bitrates: 1 Mbit/s, 500 kbit/s, 250 kbit/s, 125 kbit/s. Note all data bitrate factors cannot be used for a given arbitration bitrate. The FLEXCAN module uses an internal 60 MHz clock, that a data bitrate of 8 Mbit/s cannot be achieved.

Not that CAN bit computation can also succeed for some unusual bitrates, as 937500 bit/s and data bitrate factor of 8. You can check the result by computing actual bitrate, and the distance from the wished bitrate:

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4FD_Settings settings (937500, DataBitRateFactor::x8) ;
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual arbitration bitrate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  937 500 bit/s
  Serial.print ("actual data bitrate: ") ;
  Serial.println (settings.actualDataBitRate ()) ; //  7.5 Mbit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 0, exact bitrate
  ...
}
```

By default, a bitrate is accepted if the distance from the computed actual bitrate 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_T4FD_Settings` constructor:

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4FD_Settings settings (833000, DataBitRateFactor::x1, 200) ;
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual arbitration bitrate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  833 333 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 400 ppm
  ...
}
```

The third 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 bitrate:

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4FD_Settings settings (500 * 1000, DataBitRateFactor::x4, 0) ;
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual arbitration bitrate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  500,000 bit/s
  Serial.print ("actual data bitrate: ") ;
  Serial.println (settings.actualDataBitRate ()) ; //  2 Mbit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 0 ppm
  ...
}
```

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

```
void setup () {
```

```
  Serial.begin (9600) ;
  ACAN_T4FD_Settings settings (440 * 1000, DataBitRateFactor::x3) ;
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual arbitration bitrate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  444,444 bit/s
  Serial.print ("actual data bitrate: ") ;
  Serial.println (settings.actualDataBitRate ()) ; //  1,333,333 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 10,101 ppm
  ...
}
```

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4FD_Settings settings (1000 * 1000, DataBitRateFactor::x5) ;
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  Serial.print ("actual arbitration bitrate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  1,000,000 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 0 ppm
  Serial.print ("Bitrate prescaler: ") ;
  Serial.println (settings.mBitRatePrescaler) ; // 1
  Serial.print ("Arbitration propagation segment: ") ;
  Serial.println (settings.mArbitrationPropagationSegment) ; // 29
  Serial.print ("Arbitration phase segment 1: ") ;
  Serial.println (settings.mArbitrationPhaseSegment1) ; // 15
  Serial.print ("Arbitration phase segment 2: ") ;
  Serial.println (settings.mArbitrationPhaseSegment2) ; // 15
  Serial.print ("Arbitration resynchronization Jump Width: ") ;
  Serial.println (settings.mArbitrationRJW) ; // 15
  Serial.print ("Triple Sampling: ") ;
  Serial.println (settings.mTripleSampling) ; // 0, meaning single sampling
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.arbitrationSamplePointFromBitStart ()) ; // 75, meaning 75%
  Serial.print ("Data propagation segment: ") ;
  Serial.println (settings.mDataPropagationSegment) ; // 6
  Serial.print ("Data phase segment 1: ") ;
  Serial.println (settings.mDataPhaseSegment1) ; // 2
  Serial.print ("Data phase segment 2: ") ;
  Serial.println (settings.mDataPhaseSegment2) ; // 3
  Serial.print ("Data resynchronization Jump Width: ") ;
  Serial.println (settings.mDataRJW) ; // 3
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.DataSamplePointFromBitStart ()) ; // 75, meaning 75%
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANFDBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

The `arbitrationSamplePointFromBitStart` and the `dataSamplePointFromBitStart` method return the 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 bitrate, but it is always consistent. You can check this by calling the `CANFDBitSettingConsistency` 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) ;
  ACAN_T4FD_Settings settings (1000 * 1000, DataBitRateFactor::x5) ;
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 1 (--> is true)
  settings.mArbitrationPhaseSegment1 -- ; // 15 -> 14: safe, 1 <= PS1 <= 32
  settings.mArbitrationPhaseSegment2 ++ ; // 15 -> 16: safe, 2 <= PS2 <= 32 and RJW <= PS2
  Serial.print ("Arbitration Sample Point: ") ;
  Serial.println (settings.arbitrationSamplePointFromBitStart ()) ; // 73, meaning 73%
  Serial.print ("actual arbitration bitrate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; // 500000: ok, no change
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANFDBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

Be aware to always respect CANFD bit timing consistency!

34.2 The `CANFDBitSettingConsistency` method

This method checks the CANFD bit decomposition is consistent.

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4FD_Settings settings (1000 * 1000, DataBitRateFactor::x5) ;
  settings.mArbitrationPhaseSegment1 = 0 ; // Error, should be >= 1 (and <= 64)
  Serial.print ("Consistency: 0x") ;
  Serial.println (settings.CANFDBitSettingConsistency (), HEX) ; // 0x10, meaning error
  ...
}
```

The `CANFDBitSettingConsistency` method returns 0 if CANFD bit decomposition is consistent. Otherwise, the returned value is a bit field that can report several errors – see table 19.

The `ACAN_T4_Settings` 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 kBitRatePrescalerIsGreaterThan1024             = 1 <<  1 ;
public: static const uint32_t kArbitrationPropagationSegmentIsZero           = 1 <<  2 ;
public: static const uint32_t kArbitrationPropagationSegmentIsGreaterThan64 = 1 <<  3 ;
public: static const uint32_t kArbitrationPhaseSegment1IsZero                = 1 <<  4 ;
public: static const uint32_t kArbitrationPhaseSegment1IsGreaterThan32       = 1 <<  5 ;
public: static const uint32_t kArbitrationPhaseSegment2IsLowerThan2          = 1 <<  6 ;
public: static const uint32_t kArbitrationPhaseSegment2IsGreaterThan32       = 1 <<  7 ;
public: static const uint32_t kArbitrationRJWIsZero                          = 1 <<  8 ;
public: static const uint32_t kArbitrationRJWIsGreaterThan32                 = 1 <<  9 ;
public: static const uint32_t kArbitrationRJWIsGreaterThanPhaseSegment2      = 1 << 10 ;
```

Bit number	Error
0	mBitRatePrescaler == 0
1	mBitRatePrescaler > 1024
2	mArbitrationPropagationSegment == 0
3	mArbitrationPropagationSegment > 64
4	mArbitrationPhaseSegment1 == 0
5	mArbitrationPhaseSegment1 > 32
6	mArbitrationPhaseSegment2 == 0
7	mArbitrationPhaseSegment2 > 32
8	mArbitrationRJW == 0
9	mArbitrationRJW > 32
10	mArbitrationRJW > mArbitrationPhaseSegment2
11	mArbitrationPhaseSegment1 == 1 and *triple sampling*
12	mDataPropagationSegment == 0
13	mDataPropagationSegment > 32
14	mDataPhaseSegment1 == 0
15	mDataPhaseSegment1 > 8
16	mDataPhaseSegment2 == 0
17	mDataPhaseSegment2 > 8
18	mDataRJW == 0
19	mDataRJW > 8
20	mDataRJW > mDataPhaseSegment2

Table 19 – The ACAN_T4FD_Settings::CANFDBitSettingConsistency method error codes

```
public: static const uint32_t kArbitrationPhaseSegment1Is1AndTripleSampling = 1 << 11 ;
public: static const uint32_t kDataPropagationSegmentIsZero                = 1 << 12 ;
public: static const uint32_t kDataPropagationSegmentIsGreaterThan32       = 1 << 13 ;
public: static const uint32_t kDataPhaseSegment1IsZero                     = 1 << 14 ;
public: static const uint32_t kDataPhaseSegment1IsGreaterThan8             = 1 << 15 ;
public: static const uint32_t kDataPhaseSegment2IsLowerThan2               = 1 << 16 ;
public: static const uint32_t kDataPhaseSegment2IsGreaterThan8             = 1 << 17 ;
public: static const uint32_t kDataRJWIsZero                               = 1 << 18 ;
public: static const uint32_t kDataRJWIsGreaterThan8                       = 1 << 19 ;
public: static const uint32_t kDataRJWIsGreaterThanPhaseSegment2           = 1 << 20 ;
```

34.3 The actualArbitrationBitRate method

The actualArbitrationBitRate method returns the actual arbitration bitrate computed from mBitRatePrescaler, mArbitrationPropagationSegment, mArbitrationPhaseSegment1, mArbitrationPhaseSegment2 property values.

Note. If CANFD bit settings are not consistent (see section 34.2 page 68), the returned value is irrelevant.

34.4 The actualDataBitRate method

The `actualDataBitRate` method returns the actual data bitrate computed from `mBitRatePrescaler`, `mDataPropagationSegment`, `mDataPhaseSegment1`, `mDataPhaseSegment2` property values.

Note. If CANFD bit settings are not consistent (see section 34.2 page 68), the returned value is irrelevant.

34.5 The exactBitRate method

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

Note. If CANFD bit settings are not consistent (see section 34.2 page 68), the returned value is irrelevant.

With the default CAN root clock settings (60 MHz CAN root clock, CAN root clock divisor equal to 1, see section 37 page 74), there are 480 exact bitrates (table 20).

Arbitration bitrate (bit/s)	Available Data bitrate Factors	Arbitration bitrate (bit/s)	Available Data bitrate Factors	Arbitration bitrate (bit/s)	Available Data bitrate Factors
480	x5	500	x3 x4 x5 x6 x8 x10	600	x4 x5 x10
625	x2 x3 x4 x5 x6 x8 x10	640	x5	750	x2 x4 x5 x8 x10
768	x5	800	x3 x4 x5 x6 x8 x10	960	x4 x5 x10
1 000	x2 x3 x4 x5 x6 x8 x10	1 200	x2 x4 x5 x8 x10	1 250	x1 x2 x3 x4 x5 x6 x8 x10
1 280	x3 x5	1 500	x1 x2 x4 x5 x8 x10	1 600	x2 x3 x4 x5 x6 x10
1 875	x1 x2 x4 x5 x8 x10	1 920	x2 x5 x10	2 000	x1 x2 x3 x4 x5 x6 x8 x10
2 400	x1 x2 x4 x5 x8 x10	2 500	x1 x2 x3 x4 x5 x6 x8 x10	3 000	x1 x2 x4 x5 x8 x10
3 125	x1 x2 x3 x4 x5 x6 x8 x10	3 200	x1 x2 x3 x5 x6 x10	3 750	x1 x2 x4 x5 x8 x10
3 840	x1 x5	4 000	x1 x2 x3 x4 x5 x6 x8 x10	4 800	x1 x2 x4 x5 x10
5 000	x1 x2 x3 x4 x5 x6 x8 x10	6 000	x1 x2 x4 x5 x8 x10	6 250	x1 x2 x3 x4 x5 x6 x8 x10
6 400	x1 x3 x5	7 500	x1 x2 x4 x5 x8 x10	8 000	x1 x2 x3 x4 x5 x6 x10
9 375	x1 x2 x4 x5 x8 x10	9 600	x1 x2 x5 x10	10 000	x1 x2 x3 x4 x5 x6 x8 x10
12 000	x1 x2 x4 x5 x8 x10	12 500	x1 x2 x3 x4 x5 x6 x8 x10	15 000	x1 x2 x4 x5 x8 x10
15 625	x1 x2 x3 x4 x5 x6 x8 x10	16 000	x1 x2 x3 x5 x6 x10	18 750	x1 x2 x4 x5 x8 x10
19 200	x1 x5	20 000	x1 x2 x3 x4 x5 x6 x8 x10	24 000	x1 x2 x4 x5 x10
25 000	x1 x2 x3 x4 x5 x6 x8 x10	30 000	x1 x2 x4 x5 x8 x10	31 250	x1 x2 x3 x4 x5 x6 x8 x10
32 000	x1 x3 x5	37 500	x1 x2 x4 x5 x8 x10	40 000	x1 x2 x3 x4 x5 x6 x10
46 875	x1 x2 x4 x5 x8 x10	48 000	x1 x2 x5 x10	50 000	x1 x2 x3 x4 x5 x6 x8 x10
60 000	x1 x2 x4 x5 x8 x10	62 500	x1 x2 x3 x4 x5 x6 x8 x10	75 000	x1 x2 x4 x5 x8 x10
78 125	x1 x2 x3 x4 x6 x8	80 000	x1 x2 x3 x5 x6 x10	93 750	x1 x2 x4 x5 x8 x10
96 000	x1 x5	100 000	x1 x2 x3 x4 x5 x6 x8 x10	120 000	x1 x2 x4 x5 x10
125 000	x1 x2 x3 x4 x5 x6 x8 x10	150 000	x1 x2 x4 x5 x8 x10	156 250	x1 x2 x3 x4 x6 x8
160 000	x1 x3 x5	187 500	x1 x2 x4 x5 x8 x10	200 000	x1 x2 x3 x4 x5 x6 x10
234 375	x1 x2 x4 x8	240 000	x1 x2 x5 x10	250 000	x1 x2 x3 x4 x5 x6 x8 x10
300 000	x1 x2 x4 x5 x8 x10	312 500	x1 x2 x3 x4 x6 x8	375 000	x1 x2 x4 x5 x8 x10
400 000	x1 x2 x3 x5 x6 x10	468 750	x1 x2 x4 x8	480 000	x1 x5
500 000	x1 x2 x3 x4 x5 x6 x8 x10	600 000	x1 x2 x4 x5 x10	625 000	x1 x2 x3 x4 x6 x8
750 000	x1 x2 x4 x5 x8 x10	800 000	x1 x3 x5	937 500	x1 x2 x4 x8
1 000 000	x1 x2 x3 x4 x5 x6 x10				

Table 20 – The 480 CANFD exact bitrates (60 MHz CAN root clock, divisor equal to 1)

With the 24 MHz CAN root clock, CAN root clock divisor equal to 1 (see section 37 page 74), there are 551 exact bitrates (table 21).

Arbitration bitrate (bit/s)	Available Data bitrate Factors	Arbitration bitrate (bit/s)	Available Data bitrate Factors	Arbitration bitrate (bit/s)	Available Data bitrate Factors
192	x5	200	x3 x4 x5 x6 x8 x10	240	x4 x5 x10
250	x2 x3 x4 x5 x6 x8 x10	256	x5	300	x2 x4 x5 x8 x10
320	x3 x4 x5 x6 x8 x10	375	x2 x4 x5 x8 x10	384	x4 x5 x10
400	x2 x3 x4 x5 x6 x8 x10	480	x2 x4 x5 x8 x10	500	x1 x2 x3 x4 x5 x6 x8 x10
512	x3 x5	600	x1 x2 x4 x5 x8 x10	625	x1 x2 x3 x4 x5 x6 x8 x10
640	x2 x3 x4 x5 x6 x10	750	x1 x2 x4 x5 x8 x10	768	x2 x5 x10
800	x1 x2 x3 x4 x5 x6 x8 x10	960	x1 x2 x4 x5 x8 x10	1 000	x1 x2 x3 x4 x5 x6 x8 x10
1 200	x1 x2 x4 x5 x8 x10	1 250	x1 x2 x3 x4 x5 x6 x8 x10	1 280	x1 x2 x3 x5 x6 x10
1 500	x1 x2 x4 x5 x8 x10	1 536	x1 x5	1 600	x1 x2 x3 x4 x5 x6 x8 x10
1 875	x1 x2 x4 x5 x8 x10	1 920	x1 x2 x4 x5 x10	2 000	x1 x2 x3 x4 x5 x6 x8 x10
2 400	x1 x2 x4 x5 x8 x10	2 500	x1 x2 x3 x4 x5 x6 x8 x10	2 560	x1 x3 x5
3 000	x1 x2 x4 x5 x8 x10	3 125	x1 x2 x3 x4 x5 x6 x8 x10	3 200	x1 x2 x3 x4 x5 x6 x10
3 750	x1 x2 x4 x5 x8 x10	3 840	x1 x2 x5 x10	4 000	x1 x2 x3 x4 x5 x6 x8 x10
4 800	x1 x2 x4 x5 x8 x10	5 000	x1 x2 x3 x4 x5 x6 x8 x10	6 000	x1 x2 x4 x5 x8 x10
6 250	x1 x2 x3 x4 x5 x6 x8 x10	6 400	x1 x2 x3 x5 x6 x10	7 500	x1 x2 x4 x5 x8 x10
7 680	x1 x5	8 000	x1 x2 x3 x4 x5 x6 x8 x10	9 375	x1 x2 x4 x5 x8 x10
9 600	x1 x2 x4 x5 x10	10 000	x1 x2 x3 x4 x5 x6 x8 x10	12 000	x1 x2 x4 x5 x8 x10
12 500	x1 x2 x3 x4 x5 x6 x8 x10	12 800	x1 x3 x5	15 000	x1 x2 x4 x5 x8 x10
15 625	x1 x2 x3 x4 x6 x8	16 000	x1 x2 x3 x4 x5 x6 x10	18 750	x1 x2 x4 x5 x8 x10
19 200	x1 x2 x5 x10	20 000	x1 x2 x3 x4 x5 x6 x8 x10	24 000	x1 x2 x4 x5 x8 x10
25 000	x1 x2 x3 x4 x5 x6 x8 x10	30 000	x1 x2 x4 x5 x8 x10	31 250	x1 x2 x3 x4 x6 x8
32 000	x1 x2 x3 x5 x6 x10	37 500	x1 x2 x4 x5 x8 x10	38 400	x1 x5
40 000	x1 x2 x3 x4 x5 x6 x8 x10	46 875	x1 x2 x4 x8	48 000	x1 x2 x4 x5 x10
50 000	x1 x2 x3 x4 x5 x6 x8 x10	60 000	x1 x2 x4 x5 x8 x10	62 500	x1 x2 x3 x4 x6 x8
64 000	x1 x3 x5	75 000	x1 x2 x4 x5 x8 x10	80 000	x1 x2 x3 x4 x5 x6 x10
93 750	x1 x2 x4 x8	96 000	x1 x2 x5 x10	100 000	x1 x2 x3 x4 x5 x6 x8 x10
120 000	x1 x2 x4 x5 x8 x10	125 000	x1 x2 x3 x4 x6 x8	150 000	x1 x2 x4 x5 x8 x10
160 000	x1 x2 x3 x5 x6 x10	187 500	x1 x2 x4 x8	192 000	x1 x5
200 000	x1 x2 x3 x4 x5 x6 x8 x10	240 000	x1 x2 x4 x5 x10	250 000	x1 x2 x3 x4 x6 x8
300 000	x1 x2 x4 x5 x8 x10	320 000	x1 x3 x5	375 000	x1 x2 x4 x8
400 000	x1 x2 x3 x4 x5 x6 x10	480 000	x1 x2 x5 x10	500 000	x1 x2 x3 x4 x6 x8
600 000	x1 x2 x4 x5 x8	750 000	x1 x2 x4	800 000	x1 x2 x3 x5 x6
960 000	x1 x5	1 000 000	x1 x2 x3 x4		

Table 21 – The 551 CANFD exact bitrates (24 MHz CAN root clock, divisor equal to 1)

34.6 The ppmFromWishedBitRate method

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_T4FD_Settings settings (440 * 1000, DataBitRateFactor::x3) ;
  Serial.print ("mBitConfigurationClosedToWishedRate: ") ;
  Serial.println (settings.mBitConfigurationClosedToWishedRate) ; // 0 (--> is false)
  Serial.print ("actual arbitration bitrate: ") ;
  Serial.println (settings.actualArbitrationBitRate ()) ; //  444,444 bit/s
  Serial.print ("actual data bitrate: ") ;
  Serial.println (settings.actualDataBitRate ()) ; //  1,333,333 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromWishedBitRate ()) ; // 10,101 ppm
  ...
}
```

Note. If CAN bit settings are not consistent (see section 34.2 page 68), the returned value is irrelevant.

34.7 The arbitrationSamplePointFromBitStart method

The arbitrationSamplePointFromBitStart method returns the distance of sample point from the start of the CANFD arbitration bit, expressed in part-per-cent (ppc): 1 ppc $= 1\% = 10^{-2}$.

Note. If CANFD bit settings are not consistent (see section 34.2 page 68), the returned value is irrelevant.

34.8 The dataSamplePointFromBitStart method

The dataSamplePointFromBitStart method returns the distance of sample point from the start of the CANFD data bit, expressed in part-per-cent (ppc): 1 ppc $= 1\% = 10^{-2}$.

Note. If CANFD bit settings are not consistent (see section 34.2 page 68), the returned value is irrelevant.

34.9 Properties of the ACAN_T4FD_Settings class

All properties of the ACAN_T4FD_Settings class are declared public and are initialized (table 22) by the constructor.

Property (computed by the constructor)	**Type**	**Valid Range**
mWhishedBitRate	uint32_t	1 ... 1000000
mBitRatePrescaler	uint16_t	1 ... 1024
mArbitrationPropagationSegment	uint8_t	1 ... 64
mArbitrationPhaseSegment1	uint8_t	1 ... 32
mArbitrationPhaseSegment2	uint8_t	1 ... 32
mArbitrationRJW	uint8_t	1 ... 32
mTripleSampling	bool	false, true
mDataPropagationSegment	uint8_t	1 ... 32
mDataPhaseSegment1	uint8_t	1 ... 8
mDataPhaseSegment2	uint8_t	2 ... 8
mDataRJW	uint8_t	1 ... 8
mBitConfigurationClosedToWishedRate	bool	false, true
Initialized Property	**Type**	**Initial value**
mListenOnlyMode	bool	false
mSelfReceptionMode	bool	false
mLoopBackMode	bool	false
mReceiveBufferSize	uint16_t	32
mTransmitBufferSize	uint16_t	16
mPayload	Payload	PAYLOAD_8_BYTES
mRxCANFDMBCount	uint8_t	11 (<= MBCount (mPayload) - 2)
mTxPinOutputBufferImpedance	TxPinOutputBufferImpedance	IMPEDANCE_R0_DIVIDED_BY_6
mTxPinIsOpenCollector	bool	false
mRxPinConfiguration	RxPinConfiguration	PULLUP_47k

Table 22 – Properties of the ACAN_T4FD_Settings class

34.9.1 The `mListenOnlyMode` property

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

34.9.2 The `mSelfReceptionMode` property

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

34.9.3 The `mLoopBackMode` property

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

Part III

Setting the CAN Root Clock

35 The three CAN Root Clocks

The Teensy 4.x processor implements three clocks that can be used as *root clock* for the `CAN1`, `CAN2` and `CAN3` FlexCAN controllers. The selected root clock is used by all FlexCAN controllers.

The three available frequencies are 24 MHz, 60 MHz and 80 MHz. However, using 80 MHz root clock is problematic, it is subject to the `ERR050235` Silicon Bug.

36 The `ERR050235` Silicon Bug

The `ERR050235` Silicon bug concerns `0N00X` mask (document `RT1060_0N00X` Rev 1.2), and `1N00X` mask (document `RT1060_1N00X` Rev 1.0). The mask number is written on the chip, it can easily be read (third line, in figure 6).

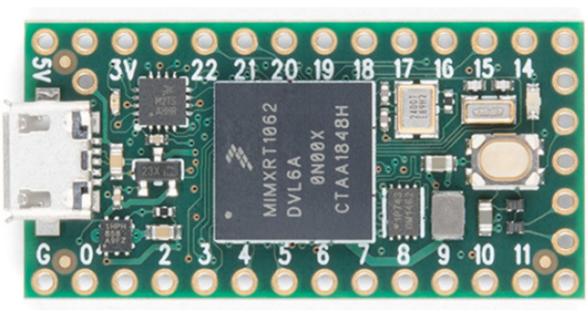

Figure 6 – Teensy 4.0, `0N00X` mask

The ERR050235 Silicon bug is described as follow in theses documents.

Description: *When selecting the CCM CAN clock source with CAN_CLK_SEL set to 2, the UART clock gate will not open and CAN_CLK_ROOT will be off. To avoid this issue, set CAN_CLK_SEL to 0 or 1 for CAN clock selection, or open the UART clock gate by configuring the CCM_CCGRx register.*

Workaround: *There are two workarounds for this issue:*

- *Set CAN_CLK_SEL to 0 or 1 for CAN clock selection.*
- *If CAN_CLK_SEL is set to 2, then the CCM must open any of UART clock gate by configuring the CCM_CCGRx register.*

Note: CAN_CLK_SEL is a 2-bit field of the CCM_CSCMR2 control register[17]:

- 0 -> CAN root Clock is 60 MHz;
- 1 -> CAN root Clock is 24 MHz;
- 2 -> CAN root Clock is 80 MHz;
- 0 -> CAN root Clock is disabled.

As the use of an UART cannot be assumed, the ACAN_T4 library does not use the 80 MHz frequency setting, only 24 MHz and 60 MHz.

37 CAN Root Clock API

The Teensy 4.x micro-controller supports two CAN root clocks, 24 MHz and 60 MHz. In addition, a CAN Root Clock divisor between 1 and 64 can be applied to this clock.

By default, the CAN Root Clock is 60 MHz, and the CAN Root Clock divisor is set to 1.

37.1 The ACAN_CAN_ROOT_CLOCK enumeration

```
enum class ACAN_CAN_ROOT_CLOCK { CLOCK_24MHz, CLOCK_60MHz } ;
```

This enumeration defines the two implemented CAN root clocks, 24 MHz and 60 MHz.

37.2 The setCANRootClock function

```
bool setCANRootClock (const ACAN_CAN_ROOT_CLOCK inCANRootClock,
                      const uint32_t inCANRootClockDivisor) ;
```

The effect of calling this function depends from the inCANRootClockDivisor value:

[17] i.MX RT1060 Processor Reference Manual, Rev. 2, 12/2019, section 14.7.8, pages 1059-1060.

- if inCANRootClockDivisor ⩾ 1 and inCANRootClockDivisor ⩽ 64, the inCANRootClock and inCANRootClockDivisor values are stored and will be used for all bitrate calculations; the function returns true;
- if inCANRootClockDivisor < 1 or inCANRootClockDivisor > 64, the inCANRootClock and inCANRootClockDivisor values are ignored and will not be used for all bitrate calculations; the function returns false; in other words, the call has no effect.

Note: Calling this function affects CAN1, CAN2 and CAN3 (CAN2.0B and CANFD). You **must** call this function before any instantiation of the ACAN_T4_Settings and ACAN_T4FD_Settings classes. The constructors of these classes use the CAN root clock settings to compute the parameters of the requested bitrates.

Note: There is no benefit in choosing inCANRootClockDivisor > 1, unless you want to achieve a low bitrate.

37.3 The getCANRootClock function

```
ACAN_CAN_ROOT_CLOCK getCANRootClock (void) ;
```

This function returns the current CAN root clock setting, either ACAN_CAN_ROOT_CLOCK::CLOCK_24MHz or ACAN_CAN_ROOT_CLOCK::CLOCK_60MHz.

37.4 The getCANRootClockFrequency function

```
uint32_t getCANRootClockFrequency (void) ; // 24 000 000, 60 000 000
```

This function returns the current CAN root clock frequency, either 24 000 000 or 60 000 000.

37.5 The getCANRootClockDivisor function

```
uint32_t getCANRootClockDivisor (void) ; // 1 ... 64
```

This function returns the current CAN root clock divisor, a value between 1 and 64.

38 An example: the 615 kbit/s bitrate

See LoopBackDemoCAN1-615kbit-s demo sketch.

The 615 kbit/s bitrate cannot be achieved with the default settings of the CAN root clock (60 MHz, divisor equal to 1). The closest bitrate is 612 244 bit/s, too far from 615 kbit/s to be accepted: the begin method returns the 0x2000000 error code, see table 7 page 29. The distance between actual bitrate and required bitrate is 4479 ppm, and the default maximum value is 1 000 ppm.

The error can be removed by specifying a larger tolerance for acceptance of the actual bitrate, for example 5 000 ppm:

```
ACAN_T4_Settings settings (615000, 5000) ; // 615 kbit/s, tolerance 5000 ppm
```

But this only silences the error, and does not affect the actual bitrate which is 612 244 bit/s.

If you want to get a closer bitrate, you should try the 24 MHz clock.

```
setCANRootClock (ACAN_CAN_ROOT_CLOCK::CLOCK_24MHz, 1) ;
```

Note you **must** call this function before any instantiation of the `ACAN_T4_Settings` and `ACAN_T4FD_Settings` classes.

Now, the actual bitrate is 615 384 bit/s, closer than the previous one. The distance between actual bitrate and required bitrate is 625 ppm, compatible with the default maximum value (1 000 ppm): the `begin` method returns the `0` error code, meaning no error.

39 Low bitrate: the 100 bit/s bitrate

See `LoopBackDemoCAN1-100bit-s` demo sketch.

With default CAN root clock settings (60 MHz, divisor equal to 1), the lowest bitrate is $\frac{60 \text{ MHz}}{256 \cdot 25} = 9\,375$ bit/s. With the 24 MHz root clock with a divisor equal to 64, the lowest bitrate becomes $\frac{24 \text{ MHz}}{64 \cdot 256 \cdot 25} = 58.59$ bit/s.

Thus, with the settings 24 MHz and divisor equal to 64, we can try a bitrate of 100 bit/s: success, it is an exact rate, reached with a bitrate prescaler equal to 150, and `mPropagationSegment` = `mPhaseSegment1` = `mPhaseSegment2` = 8 (see section 17.2 page 34).