ACAN_ESP32 library for ESP32
Version 1.0.0

Pierre Molinaro

April 18, 2021

Contents

1 Versions

Version	Date	Comment
1.0.0	April 18, 2021	Initial release

2 Features

The ACAN_ESP32 library is a CAN ("Controller Area Network") driver for Teensy 3.1 / 3.2, 3.5, 3.6. It has been designed to make it easy to start and to be easily configurable:

- default configuration sends and receives any frame – no default filter to provide;
- efficient built-in CAN bit settings computation from user bit rate;
- user can fully define its own CAN bit setting values;
- 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* controller modes are selectable;
- Tx pin and Rx pins are selectable.

In version 1.0.0, reception filters are not documented: all network frames are received.

3 ESP32 builtin CAN Controller

ESP32 builtin CAN Controller is not official. In section 4.1.18 page 36 of the ESP32 datasheet[1], it is very shortly documented as a `TWAI`[2] controller. Actually, it is a CAN 2.0B controller. Specifically, this CAN module implements most of the functionality of an SJA1000[3].

This library is based upon the Mohamed Irfanulla MOHAMED ABDULLA Master[4]. You can find a copy of this thesis in the `extras` directory. The corresponding code is on the `https://github.com/irfanafa/ESP32ACAN` repository.

4 Data flow

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

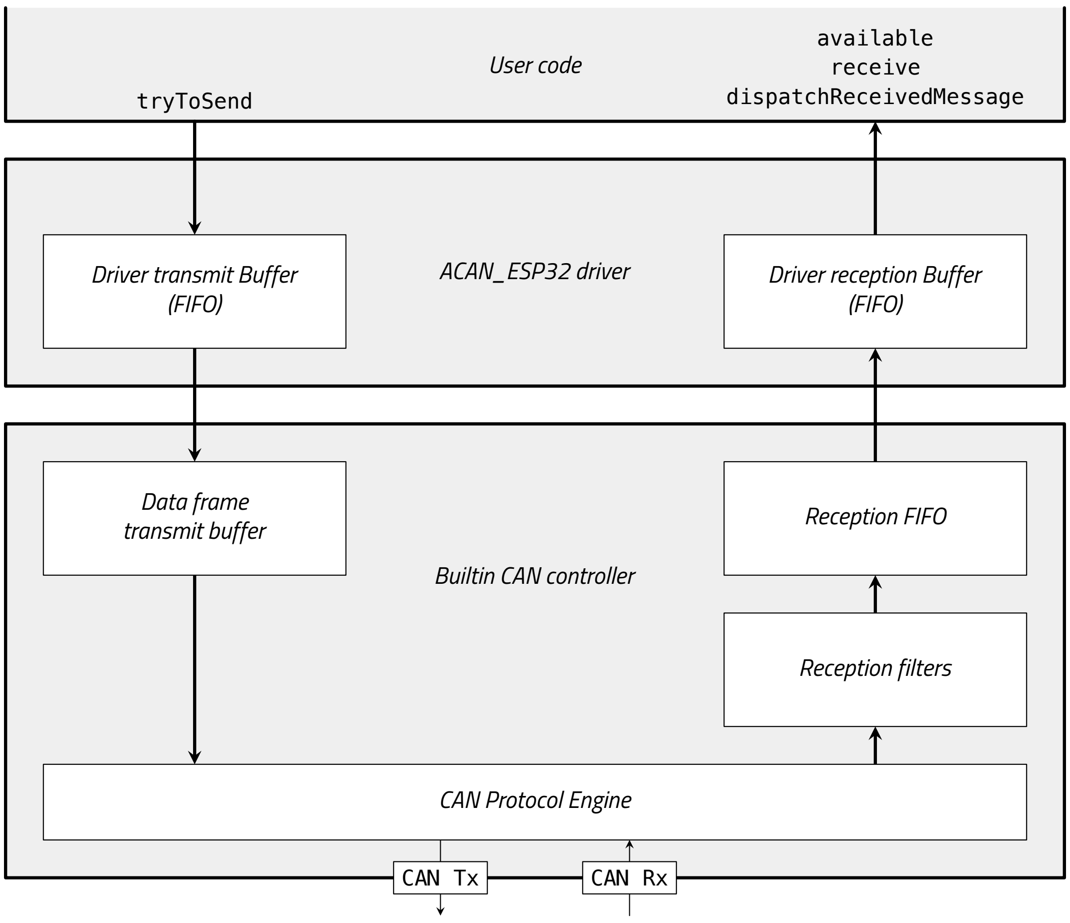

Figure 1 – Message flow in ACAN_ESP32 driver and Builtin CAN controller

[1]Espressif Systems, *ESP32 Series Datasheet*, Version 3.6, 2021, `https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf`

[2]TWAI: *Two-Wire Automotive Interface*.

[3]Philips, *SJA1000 Stand-alone CAN controller data sheet*, 2000 January 4, `https://www.nxp.com/docs/en/data-sheet/SJA1000.pdf`

[4]Mohamed Irfanulla MOHAMED ABDULLA, *Development of ESP32 CAN Driver*, École Centrale de Nantes, France, 28 August 2019.

Builtin CAN controller is hardware, a module of the ESP32 micro-controller. It is a CAN 2.0B controller, it implements most of the functionality of a SJA1000 controller :

- one transmit buffer;
- a 64-byte receive FIFO;
- 8 8-bits registers for handling receive filters (not documented in release 1.0.0).

Sending messages. The CAN hardware makes sending data frames different from sending remote frames. For both, user code calls the `tryToSend` method – see section 9 page 9. The 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 9.1 page 10 for changing the default value.

Receiving messages. The CAN *CAN Protocol Engine* transmits all correct frames to the *reception filters*. By default, they are configured as pass-all, and their configuration is not documented in release 1.0.0. Messages that pass the filters are stored in the 64-byte *Reception FIFO*. Its depth depends from the received message size: a standard frame with n data bytes occupies $n+3$ bytes in the FIFO; an extended frame with n data bytes occupies $n+5$ bytes in the FIFO. If, when receiving a frame that passes the filters, there is not enough room in the FIFO, the frame is lost. The message interrupt service routine transfers the messages from *Reception FIFO* to the *Driver Receive Buffer*. The size of the *Driver Receive Buffer* is 32 by default – see section 10.1 page 12 for changing the default value. Two user methods are available:

- the `available` method returns `false` if the *Driver Receive Buffer* is empty, and `true` otherwise;
- the `receive` method retrieves messages from the *Driver Receive Buffer* – see section 10 page 11;

Sequentiality. The ACAN_ESP32 driver and the configuration of the CAN controller ensures sequentiality of data messages. This means that if an user program calls `tryToSend` first for a message M_1 and then for a message M_2, the message M_1 will be always retrieved by `receive` or `dispatchReceivedMessage` before the message M_2.

5 A simple example: LoopBackDemo

The following code is a sample code for introducing the ACAN_ESP32 library. It demonstrates how to configure the driver, to send a CAN message, and to receive a CAN message.

Note that, unlike other microcontrollers, the loopback mode requires the connection with a transceiver. The figure 2 shows a connection with a `MCP2562` transceiver. The ACAN_ESP32 driver uses by default `GPIO5` as CAN transmit signal, and `GPIO4` as CAN receive signal. Other pins can be used, see section 8 page 8.

The `LoopBackDemo` sketch is:

```
#include <ACAN_ESP32.h>

static const uint32_t DESIRED_BIT_RATE = 1000UL * 1000UL ; // 1 Mb/s

```

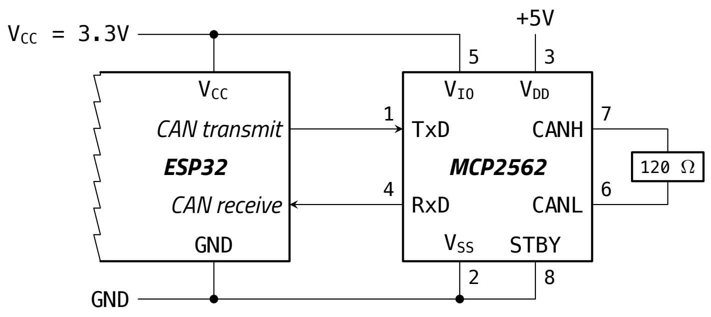

Figure 2 – Connecting an ESP32 to a MCP2562 CAN transceiver

```
void setup() {
//--- Configure builtin led
  pinMode (LED_BUILTIN, OUTPUT) ;
  digitalWrite (LED_BUILTIN, HIGH) ;
//--- Start serial
  Serial.begin (115200) ;
//--- Wait for serial (blink led at 10 Hz during waiting)
  while (!Serial) {
    delay (50) ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
  }
//--- Configure ESP32 CAN
  Serial.println ("Configure ESP32 CAN") ;
  ACAN_ESP32_Settings settings (DESIRED_BIT_RATE) ; // CAN bit rate
  settings.mRequestedCANMode = ACAN_ESP32_Settings::LoopBackMode; // Select loopback mode
  const uint16_t errorCode = ACAN_ESP32::can.begin (settings) ;
  if (errorCode == 0) {
    Serial.print ("Bit Rate prescaler: ") ;
    Serial.println (settings.mBitRatePrescaler) ;
    Serial.print ("Time Segment 1:     ") ;
    Serial.println (settings.mTimeSegment1) ;
    Serial.print ("Time Segment 2:     ") ;
    Serial.println (settings.mTimeSegment2) ;
    Serial.print ("SJW:                ") ;
    Serial.println (settings.mSJW) ;
    Serial.print ("Triple Sampling:    ") ;
    Serial.println (settings.mTripleSampling ? "yes" : "no") ;
    Serial.print ("Actual bit rate:    ") ;
    Serial.print (settings.actualBitRate ()) ;
    Serial.println (" bit/s") ;
    Serial.print ("Exact bit rate ?    ") ;
    Serial.println (settings.exactBitRate () ? "yes" : "no") ;
    Serial.print ("Sample point:       ") ;
    Serial.print (settings.samplePointFromBitStart ()) ;
    Serial.println ("%") ;
40      Serial.println ("Configuration OK!");
41    }else {
42      Serial.print ("Configuration error 0x") ;
43      Serial.println (errorCode, HEX) ;
44    }
45  }
46
47  static uint32_t gBlinkLedDate = 0;
48  static uint32_t gReceivedFrameCount = 0 ;
49  static uint32_t gSentFrameCount = 0 ;
50
51  void loop() {
52    CANMessage frame ;
53    if (gBlinkLedDate < millis ()) {
54      gBlinkLedDate += 500 ;
55      digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
56      Serial.print ("Sent: ") ;
57      Serial.print (gSentFrameCount) ;
58      Serial.print ("\t") ;
59      Serial.print ("Receive: ") ;
60      Serial.print (gReceivedFrameCount) ;
61      Serial.print ("\t") ;
62      Serial.print (" STATUS 0x") ;
63      Serial.print (CAN_STATUS, HEX) ;
64      Serial.print (" RXERR ") ;
65      Serial.print (CAN_RX_ECR) ;
66      Serial.print (" TXERR ") ;
67      Serial.println (CAN_TX_ECR) ;
68      frame.len = 8 ;
69      const bool ok = ACAN_ESP32::can.tryToSend (frame) ;
70      if (ok) {
71        gSentFrameCount += 1 ;
72      }
73    }
74    while (ACAN_ESP32::can.receive (frame)) {
75      gReceivedFrameCount += 1 ;
76    }
77  }
```

Line 1. This line includes the ACAN_ESP32 library.

Line 3. Declaration of the baud rate, in bit/s.

Line 18. Configuration is a four-step operation. This line is the first step. It instanciates the `settings` object of the `ACAN_ESP32_Settings` class. The constructor has one parameter: the wished CAN bit rate. It returns a `settings` object fully initialized with CAN bit settings for the wished bit rate, and default values for other configuration properties.

Line 19. This is the second step. You can override the values of the properties of `settings` object. Here, the mRequestedCANMode properties is set to ACAN_ESP32_Settings::LoopBackMode – it is NormalMode by default. If you want to change CAN transmit and receive pins, write here the new settings (see section 8 page 8). The section 12.7 page 20 lists all properties you can override.

Line 20. This is the third step, configuration of the ACAN_ESP32::can driver with settings values. You cannot change the ACAN_ESP32::can name – see section 7 page 8. The driver is configured for being able to send any (standard / extended, data / remote) frame, and to receive all (standard / extended, data / remote) frames.

Lines 21 to 44. Last step: the configuration of the ACAN_ESP32::can driver returns an error code, stored in the errorCode constant. It has the value 0 if all is ok – see section 11.2 page 13.

Line 47. The gBlinkLedDate global variable is used for sending a CAN message every 0.5 s.

Line 48. The gReceivedFrameCount global variable counts the number of received messages.

Line 49. The gSentFrameCount global variable counts the number of sent messages.

Line 52. 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 6 page 7.

Line 53. It tests if it is time to blink the led, print send and receive counters, and to send a message.

Line 68. Set the message length. In a real code, we set here message data, identifier, and for an extended frame the ext boolean property.

Line 69. 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 70 to 72. 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 74. As the CAN controller is configured in *loop back* mode (see lines 7 and 8), all sent messages are received. The receive method returns false if no message is available from the *driver reception buffer*. It returns true if a message has been successfully removed from the *driver reception buffer*. This message is assigned to the message object.

Line 75. It a message has been received, the gReceivedFrameCount is incremented and displayed.

6 The CANMessage class

Note. The CANMessage class is declared in the CANMessage.h header file. The class declaration is protected by an include guard that causes the macro GENERIC_CAN_MESSAGE_DEFINED to be defined. The ACAN2515 driver 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 ;  // Used by the ACAN driver
  public : uint8_t len = 0 ;  // Length of data (0 ... 8)
  public : union {
    uint64_t data64        ; // Caution: subject to endianness
    uint32_t data32 [2]    ; // Caution: subject to endianness
    uint16_t data16 [4]    ; // Caution: subject to endianness
    float    dataFloat [2] ; // Caution: subject to endianness
    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 (ESP32 processor is little-endian).

7 Driver instance

The driver instance name is `ACAN_ESP32::can`. You cannot choose its name, it is defined by the library.

Note. The driver variable is an ACAN_ESP32 class static property. This choice may seem strange. However, a common error is to declare its own driver variable:

```
ACAN_ESP32 myCAN ; // Don't do that, it is an error !!!
```

Declaring a driver variable as ACAN_ESP32 class static property[5] enables the compiler to raise an error if you try to declare your own driver variable.

8 Pin selection

By default, CAN transmit pin is `GPIO5`, and CAN receive pin is `GPIO4`.

For using other pins, just set `mTxPin` and / or `mRxPin` properties of `settings` object. For example:

```
ACAN_ESP32_Settings settings (125 * 1000) ;
settings.mTxPin = GPIO_NUM_2 ;
settings.mRxPin = GPIO_NUM_13 ;
const uint32_t errorCode = ACAN_ESP32::can.begin (settings) ;
```

The `mTxPin` and `mRxPin` properties type is `gpio_num_t`, so you should use the `GPIO_NUM_`n names.

[5]The ACAN_ESP32 constructor is declared `private`.

Note. Particular care must be taken in the choice of pins. Indeed, some pins ouput a PWM at boot, others require a high or low level, ... The https://randomnerdtutorials.com/esp32-pinout-reference-gpios/ *shows what pins are best to use as inputs, outputs and which ones you need to be cautious.*

For example, it is a bad choice to use `GPIO0` as CAN transmit pins: it outputs PWM signal at boot, disturbing the CAN bus. Using `GPIO12` as CAN receive pin provide a boot failure: if the CAN bus is recessive, the transceiver outputs a high level on its `RxD` pin, and boot fails if `GPIO12` is pulled high.

9 Sending frames

Call the method `tryToSend` for sending 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 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_ESP32::can.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_ESP32::can.tryToSend (message) ;
    if (ok) {
      gSendMessage = false ;
    }
  }
  ...
}
```

9.1 Driver transmit buffer size

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

```
ACAN_ESP32_Settings settings (125 * 1000) ;
settings.mDriverTransmitBufferSize = 30 ;
const uint32_t errorCode = ACAN_ESP32::can.begin (settings) ;
...
```

As the size of CAN`Message` class is 16 bytes, the actual size of the driver transmit buffer is the value of `settings.mDriverTransmitBufferSize * 16`.

9.2 The `driverTransmitBufferSize` method

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

```
const uint32_t s = ACAN_ESP32::can.driverTransmitBufferSize () ;
```

9.3 The `driverTransmitBufferCount` method

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

```
const uint32_t n = ACAN_ESP32::can.driverTransmitBufferCount () ;
```

9.4 The `driverTransmitBufferPeakCount` method

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

```
const uint32_t max = ACAN_ESP32::can.driverTransmitBufferPeakCount () ;
```

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

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

10 Retrieving received messages using the `receive` method

This is a basic example:

```
void setup () {
  ACAN_ESP32_Settings settings (125 * 1000) ;
  ...
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings) ; // No receive filter
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN_ESP32::can.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_ESP32_Settings settings (125 * 1000) ;
  ...
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings) ; // No receive filter
  ...
}

void loop () {
  CANMessage message ;
  if (ACAN_ESP32::can.receive (message)) {
    if (!message.rtr && message.ext && (message.id == 0x123456)) {
      handle_myMessage_0 (message) ; // Extended data frame, id is 0x123456
    }else if (!message.rtr && !message.ext && (message.id == 0x234)) {
      handle_myMessage_1 (message) ;  // Standard data frame, id is 0x234
    }else if (message.rtr && !message.ext && (message.id == 0x542)) {
      handle_myMessage_2 (message) ;  // Standard remote frame, id is 0x542
    }
  }
  ...
}
```

The `handle_myMessage_0` function has the following header:

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

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

10.1 Driver receive buffer size

By default, the driver receive buffer size is 32.

You can change this default value by setting the `mDriverReceiveBufferSize` property of `settings` variable:

```
ACAN_ESP32_Settings settings (125 * 1000) ;
settings.mDriverReceiveBufferSize = 100 ;
const uint32_t errorCode = ACAN_ESP32::can.begin (settings) ;
...
```

The actual size of the driver receive buffer is the value of `settings.mDriverReceiveBufferSize` * 16 (the size of `CANMessage` class is 16 bytes).

10.2 The `driverReceiveBufferSize` method

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

```
const uint32_t s = ACAN_ESP32::can.receiveBufferSize () ;
```

10.3 The `driverReceiveBufferCount` method

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

```
const uint32_t n = ACAN_ESP32::can.driverReceiveBufferCount () ;
```

10.4 The `driverReceiveBufferPeakCount` method

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

```
const uint32_t max = ACAN_ESP32::can.driverReceiveBufferPeakCount () ;
```

Note the driver receive buffer may overflow, if messages are not retrieved (by calls of the `receive` method or the `dispatchReceivedMessage` method). If an overflow occurs, further calls of the `ACAN_ESP32::can.receiveBufferPeakCount ()` method return `ACAN_ESP32::can.receiveBufferSize ()`+1.

11 The ACAN_ESP32::begin method reference

11.1 The ACAN_ESP32::begin method prototype

The begin method prototype is:

```
uint32_t ACAN_ESP32::begin (const ACAN_ESP32_Settings & inSettings) ;
```

11.2 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. An error code could report several errors. Bits from 0 to 8 are actually defined by the ACAN_ESP32_Settings class and are also returned by the CANBitSettingConsistency method (see section 12.2 page 17). Bits from 9 are defined by the ACAN_ESP32 class.

The ACAN_ESP32_Settings class defines static constant properties that can be used as mask error:

```
  public: static const uint16_t kBitRatePrescalerIsLowerThan2     = 1 <<  0 ;
  public: static const uint16_t kBitRatePrescalerIsGreaterThan128 = 1 <<  1 ;
  public: static const uint16_t kTimeSegment1IsZero               = 1 <<  2 ;
  public: static const uint16_t kTimeSegment1IsGreaterThan16      = 1 <<  3 ;
  public: static const uint16_t kTimeSegment2IsZero               = 1 <<  4 ;
  public: static const uint16_t kTimeSegment2IsGreaterThan8       = 1 <<  5 ;
  public: static const uint16_t kTimeSegment1Is1AndTripleSampling = 1 <<  6 ;
  public: static const uint16_t kSJWIsZero                        = 1 <<  7 ;
  public: static const uint16_t kSJWIsGreaterThan4                = 1 <<  8 ;
```

The ACAN_ESP32 class defines static constant properties that can be used as mask error:

```
  public: static const uint32_t kNotInRestModeInConfiguration       = 1 <<  9 ;
  public: static const uint32_t kCANRegistersError                  = 1 << 10 ;
  public: static const uint32_t kTooFarFromDesiredBitRate           = 1 << 11 ;
  public: static const uint32_t kInconsistentBitRateSettings        = 1 << 12 ;
  public: static const uint32_t kCannotAllocateDriverReceiveBuffer  = 1 << 13 ;
  public: static const uint32_t kCannotAllocateDriverTransmitBuffer = 1 << 14 ;
```

For example, you can write:

```
const uint32_t errorCode = ACAN_ESP32::can.begin (settings) ;
if (errorCode != 0) {
  ...
  if ((errorCode & ACAN_ESP32::kTooFarFromDesiredBitRate) != 0) {
    // Error: too far from desired bit rate
  }
  ...
}
```

11.2.1 CAN Bit setting too far from desired rate

This error is raised when the `mBitRateClosedToDesiredRate` of the `settings` object is false. This means that the `ACAN_ESP32_Settings` constructor cannot compute a CAN bit configuration close enough to the wished bit rate. When the `begin` is called with `settings.mBitRateClosedToDesiredRate` false, this error is reported. For example:

```
void setup () {
  ACAN_ESP32_Settings settings (1) ; // 1 bit/s !!!
  // Here, settings.mBitRateClosedToDesiredRate is false
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings) ;
  // Here, errorCode == ACAN_ESP32::kCANBitConfigurationTooFarFromWishedBitRateErrorMask
}
```

This error is a fatal error, the driver and the CAN module are not configured. See section 12.1 page 14 for a discussion about CAN bit setting computation.

11.2.2 CAN Bit inconsistent configuration error

This error is raised when you have changed the CAN bit properties (`mBitRatePrescaler`, `mTimeSegment1`, `mTimeSegment2`, `mSJW`), and one or more resulting values are inconsistent. See section 12.2 page 17.

12 ACAN_ESP32_Settings class reference

12.1 The ACAN_ESP32_Settings constructor: computation of the CAN bit settings

The constructor of the `ACAN_ESP32_Settings` has one mandatory argument: the wished bit rate. It tries to compute the CAN bit settings for this bit rate. If it succeeds, the constructed object has its `mBitConfigurationClosed-ToWishedRate` property set to `true`, otherwise it is set to `false`. For example:

```
void setup () {
  ACAN_ESP32_Settings settings (1 * 1000 * 1000) ; // 1 Mbit/s
  // Here, settings.mBitRateClosedToDesiredRate is true
  ...
}
```

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (842 * 1000) ; // 842 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 125 ppm
  ...
}
```

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

By default, a wished bit rate is accepted if the distance from the computed actual bit rate is lower or equal to $1,000$ ppm $= 0.1$ %. You can change this default value by adding your own value as second argument of `ACAN_ESP32_Settings` constructor:

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (842 * 1000, 100) ; // 842 kbit/s, max distance is 100 ppm
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 125 ppm
  ...
}
```

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (500 * 1000, 0) ; // 500 kbit/s, max distance is 0 ppm
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  500,000 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 0 ppm
  ...
}
```

The slowest exact bit rate is 25 kbit/s.

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 1001 ppm
  ...
}
```

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 1001 ppm
  Serial.print ("Bit rate prescaler: ") ;
  Serial.println (settings.mBitRatePrescaler) ; // BRP = 9
  Serial.print ("Time segment 1: ") ;
  Serial.println (settings.mTimeSegment1) ; // 15
  Serial.print ("Time segment 2: ") ;
  Serial.println (settings.mTimeSegment2) ; // 4
  Serial.print ("Resynchronization Jump Width: ") ;
  Serial.println (settings.mSJW) ; // SJW = 4
  Serial.print ("Triple Sampling: ") ;
  Serial.println (settings.mTripleSampling) ; // 0, meaning single sampling
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 80, meaning 80%
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

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

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

You can change the property values for adapting to the particularities of your CAN network propagation time. By example, you can increment the `mTimeSegment1` value, and decrement the `mTimeSegment2` value in order to sample the `CAN_Rx` pin later.

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  settings.mTimeSegment1 -- ; // 15 -> 14: safe, 1 <= TS1 <= 16
  settings.mTimeSegment2 ++ ; // 4 -> 5: safe, 2 <= TS2 <= 8 and SJW <= PS2
  Serial.print ("Sample Point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 75, meaning 75%
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  500000: ok, bit rate did not change
  Serial.print ("Consistency: ") ;
  Serial.println (settings.CANBitSettingConsistency ()) ; // 0, meaning Ok
  ...
}
```

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

$$\begin{aligned}
2 &\leqslant \texttt{mBitRatePrescaler} \leqslant 128\\
1 &\leqslant \texttt{mSJW} \leqslant 4\\
\text{Single sampling: } 1 &\leqslant \texttt{mTimeSegment1} \leqslant 16\\
\text{Triple sampling: } 2 &\leqslant \texttt{mTimeSegment1} \leqslant 8\\
2 &\leqslant \texttt{mTimeSegment2} \leqslant 8\\
\texttt{mSJW} &\leqslant \texttt{mTimeSegment2}
\end{aligned}$$

Resulting actual bit rate is given by:

$$\text{Actual bit rate} = \frac{80\text{ MHz}}{\texttt{mBitRatePrescaler} \cdot (1 + \texttt{mTimeSegment1} + \texttt{mTimeSegment2})}$$

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

$$\text{Sampling point \textit{(single sampling)}} = 100 \cdot \frac{1 + \texttt{mTimeSegment1}}{1 + \texttt{mTimeSegment1} + \texttt{mTimeSegment2}}$$

$$\text{Sampling first point \textit{(triple sampling)}} = 100 \cdot \frac{\texttt{mTimeSegment1}}{1 + \texttt{mTimeSegment1} + \texttt{mTimeSegment2}}$$

12.2 The CANBitSettingConsistency method

This method checks the CAN bit decomposition (given by mBitRatePrescaler, mTimeSegment1, mTimeSegment2, mSJW property values) is consistent.

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  settings.mTimeSegment1 = 0 ; // Error, mTimeSegment1 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 1.

Bit number	**Error**
0	mBitRatePrescaler < 2
1	mBitRatePrescaler > 128
2	mTimeSegment1 == 0
3	mTimeSegment1 > 16
4	mTimeSegment2 == 0
5	mTimeSegment2 > 8
6	mTimeSegment1 == 1 and *triple sampling*
7	mSJW == 0
8	mSJW > 4

Table 1 – The ACAN_ESP32_Settings::CANBitSettingConsistency method error codes

The ACAN_ESP32_Settings class defines static constant properties that can be used as mask error:

```
  public: static const uint16_t kBitRatePrescalerIsLowerThan2     = 1 <<  0 ;
  public: static const uint16_t kBitRatePrescalerIsGreaterThan128 = 1 <<  1 ;
  public: static const uint16_t kTimeSegment1IsZero               = 1 <<  2 ;
  public: static const uint16_t kTimeSegment1IsGreaterThan16      = 1 <<  3 ;
  public: static const uint16_t kTimeSegment2IsZero               = 1 <<  4 ;
  public: static const uint16_t kTimeSegment2IsGreaterThan8       = 1 <<  5 ;
  public: static const uint16_t kTimeSegment1Is1AndTripleSampling = 1 <<  6 ;
  public: static const uint16_t kSJWIsZero                        = 1 <<  7 ;
  public: static const uint16_t kSJWIsGreaterThan4                = 1 <<  8 ;
```

12.3 The actualBitRate method

The actualBitRate method returns the actual bit computed from mBitRatePrescaler, mTimeSegment1, mTimeSegment2, mSJW property values.

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (440 * 1000) ; // 440 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 0 (--> is false)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  444,444 bit/s
  ...
}
```

Note. If CAN bit settings are not consistent (see section 12.2 page 17), the returned value is irrelevant.

12.4 The exactBitRate method

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (842 * 1000) ; // 842 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 125 ppm
  Serial.print ("Exact: ") ;
  Serial.println (settings.exactBitRate ()) ; // 0 (---> false)
  ...
}
```

Note. If CAN bit settings are not consistent (see section 12.2 page 17), the returned value is irrelevant.

12.5 The ppmFromDesiredBitRate method

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

```
void setup () {
  Serial.begin (9600) ;
  ACAN_ESP32_Settings settings (842 * 1000) ; // 842 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("actual bit rate: ") ;
  Serial.println (settings.actualBitRate ()) ; //  842105 bit/s
  Serial.print ("distance: ") ;
  Serial.println (settings.ppmFromDesiredBitRate ()) ; // 125 ppm
  ...
}
```

Note. If CAN bit settings are not consistent (see section 12.2 page 17), the returned value is irrelevant.

12.6 The samplePointFromBitStart method

The `samplePointFromBitStart` method returns the distance of sample point from the start of the CAN bit, expressed in part-per-cent (ppc): 1 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_ESP32_Settings settings (500 * 1000) ; // 500 kbit/s
  Serial.print ("mBitRateClosedToDesiredRate: ") ;
  Serial.println (settings.mBitRateClosedToDesiredRate) ; // 1 (--> is true)
  Serial.print ("Sample point: ") ;
  Serial.println (settings.samplePointFromBitStart ()) ; // 80 --> 80%
  ...
}
```

Note. If CAN bit settings are not consistent (see section 12.2 page 17), the returned value is irrelevant.

12.7 Properties of the ACAN_ESP32_Settings class

All properties of the ACAN_ESP32_Settings class are declared public and are initialized (table 2).

Property	Type	Initial value	Comment
mTxPin	gpio_num_t	GPIO_NUM_5	See section 8 page 8
mRxPin	gpio_num_t	GPIO_NUM_4	See section 8 page 8
mDesiredBitRate	uint32_t	*Initialized by constructor*	See section 12.1 page 14
mBitRatePrescaler	uint16_t	*Initialized by constructor*	See section 12.1 page 14
mTimeSegment1	uint8_t	*Initialized by constructor*	See section 12.1 page 14
mTimeSegment2	uint8_t	*Initialized by constructor*	See section 12.1 page 14
mSJW	uint8_t	*Initialized by constructor*	See section 12.1 page 14
mTripleSampling	bool	*Initialized by constructor*	See section 12.1 page 14
mBitRateClosedToDesiredRate	bool	*Initialized by constructor*	See section 12.1 page 14
mRequestedCANMode	CANMode	NormalMode	See section 12.7.1 page 20
mDriverReceiveBufferSize	uint16_t	32	See section 10.1 page 12
mDriverTransmitBufferSize	uint16_t	16	See section 9.1 page 10

Table 2 – Properties of the ACAN_ESP32_Settings class

12.7.1 The mRequestedCANMode property

This property has three possible values, as described in the table 3. It corresponds to the LOM and STM bits of the MODE control register. The default value is ACAN_ESP32_Settings::NormalMode.

Value	**Comment, from SJA1000 datasheet**
`ACAN_ESP32_Settings::NormalMode`	*An acknowledge is required for successful transmission.*
`ACAN_ESP32_Settings::ListenOnlyMode`	*In this mode the CAN controller would give no acknowledge to the CAN-bus, even if a message is received successfully; the error counters are stopped at the current value. This mode of operation forces the CAN controller to be error passive. Message transmission is not possible. The listen only mode can be used e.g. for software driven bit rate detection and 'hot plugging'. All other functions can be used like in normal mode.*
`ACAN_ESP32_Settings::LoopBackMode`	*In this mode a full node test is possible without any other active node on the bus using the self reception request command; the CAN controller will perform a successful transmission, even if there is no acknowledge received.*

Table 3 – Values of the `mRequestedCANMode` property of the `ACAN_ESP32_Settings` class