ACAN_ESP32 library for ESP32
Version 3.0.3

Pierre Molinaro

June 16, 2025

Contents

1 Versions 3

2 Features 4

3 ESP32 builtin CAN Controller 4

4 Data flow 5

5 A simple example: `LoopBackDemo` 6

6 The `CANMessage` class 10

7 Driver instance naming 11

7.1 Microcontroller with one CAN module 11

7.2 Microcontroller with two CAN modules 11

8 Pin selection 12

9 Sending frames 12

9.1 Driver transmit buffer size 14

9.2 The `driverTransmitBufferSize` method 14

9.3 The `driverTransmitBufferCount` method 14

9.4 The `driverTransmitBufferPeakCount` method 15

9.5 The `resetDriverTransmitBufferPeakCount` method 15

10 Retrieving received messages using the `receive` method 15

10.1 The receive method . . . 15
10.2 The available method . . . 16
10.3 Driver receive buffer size . . . 17
10.4 The driverReceiveBufferSize method . . . 17
10.5 The driverReceiveBufferCount method . . . 17
10.6 The driverReceiveBufferPeakCount method . . . 17
10.7 The resetDriverReceiveBufferPeakCount method . . . 18

11 Filtering received messages 18
11.1 Accept only standard frames . . . 19
11.2 Accept only extended frames . . . 20
11.3 Standard frame single filter . . . 20
11.3.1 Example 1 . . . 20
11.3.2 Example 2 . . . 21
11.4 Extended frame single filter . . . 21
11.4.1 Example 1 . . . 22
11.4.2 Example 2 . . . 22
11.5 Standard frame dual filter . . . 23
11.5.1 Example . . . 23
11.6 Extended frame dual filter . . . 24
11.6.1 Example . . . 25
11.7 Defining your own filter . . . 25

12 The ACAN_ESP32::begin method reference 27
12.1 The ACAN_ESP32::begin method prototype . . . 27
12.2 The error code . . . 27
12.2.1 CAN Bit setting too far from desired rate . . . 28
12.2.2 CAN Bit inconsistent configuration error . . . 28

13 Other ACAN_ESP32 methods 28
13.1 The ACAN_ESP32::statusFlags method . . . 28
13.2 The ACAN_ESP32::recoverFromBusOff method . . . 29

14 ACAN_ESP32_Settings class reference 29
14.1 The ACAN_ESP32_Settings constructor: computation of the CAN bit settings . . . 29
14.2 The CANBitSettingConsistency method . . . 33
14.3 The actualBitRate method . . . 34
14.4 The exactBitRate method . . . 34
14.5 The ppmFromDesiredBitRate method . . . 35
14.6 The samplePointFromBitStart method . . . 35

14.7 Properties of the ACAN_ESP32_Settings class 35
14.7.1 The mRequestedCANMode property 36

1 Versions

Releases before 1.0.3 do not compile on `ESP32 Arduino` 2.x.x.

Releases 1.x.x do not compile on `ESP32 Arduino` 3.x.x.

The ESP32C6 is supported from release 3.0.0, see section 7 page 11.

Version	Date	Comment
3.0.3	June 16, 2025	Back compatibility with ESP32 2.0.17 board manager.
3.0.2	June 14, 2025	In ESP32 3.3.0-alpha1 board manager, `periph_interrput_t` has been deprecated in favor of `periph_interrupt_t` (thanks to `Simontpellier`).
3.0.1	March 11, 2025	Fixed bad received CAN frame identifier, bug introduced in release 3.0.0 (thanks to `AntonioFrog`).
3.0.0	March 6, 2025	Support of `ESP32C6` added.
2.0.1	March 22, 2024	`CANMessage.h` file renamed to `ACAN_ESP32_CANMessage.h`.
2.0.0	January 10, 2024	Updated for ESP32 Arduino 3.0.0-alpha3.
1.1.2	August 24, 2023	Added compatibility with `ESP32C3` and `ESP32S3`.
1.1.0	September 24, 2022	Added `available` method (section 10.2 page 16), thanks to `Modelfan`. Added `recoverFromBusOff` method (section 13.2 page 29), thanks to `matthew-mower`. Added `statusFlags` method (section 13.1 page 28). Control register names conform to ESP32 datasheet names.
1.0.6	February 14, 2022	Added `resetDriverTransmitBufferPeakCount` method (section 9.5 page 15) and `resetDriverReceiveBufferPeakCount` method (section 10.7 page 18).
1.0.5	October 1, 2021	Added `data_s64`, `data_s32`, `data_s16` and `data_s8` to `CANMessage` class union members, see section 6 page 10 (thanks to `tomtom0707`).
1.0.4	August 14, 2021	Corrected typo in library description.
1.0.3	August 13, 2021	Updated for ESP32 Arduino 2.0.0-rc1.

Version	Date	Comment
1.0.2	June 26, 2021	Fixed `tryToSend` bug (thanks to `DirkMeintjies`).
1.0.1	April 26, 2021	Adding reception filters.
		For some bit rate settings, RJW value was invalid.
		Error codes have been changed (section 12.2 page 27).
1.0.0	April 18, 2021	Initial release.

2 Features

The `ACAN_ESP32` library is a CAN ("Controller Area Network") driver for ESP32 chips. It has been designed to make it easy to start and to be easily configurable:

- default configuration sends and receives any frame – no default filter to provide;
- efficient built-in CAN bit settings computation from user bit rate;
- user can fully define its own CAN bit setting values;
- reception filters are easily defined;
- 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.

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].

[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

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.

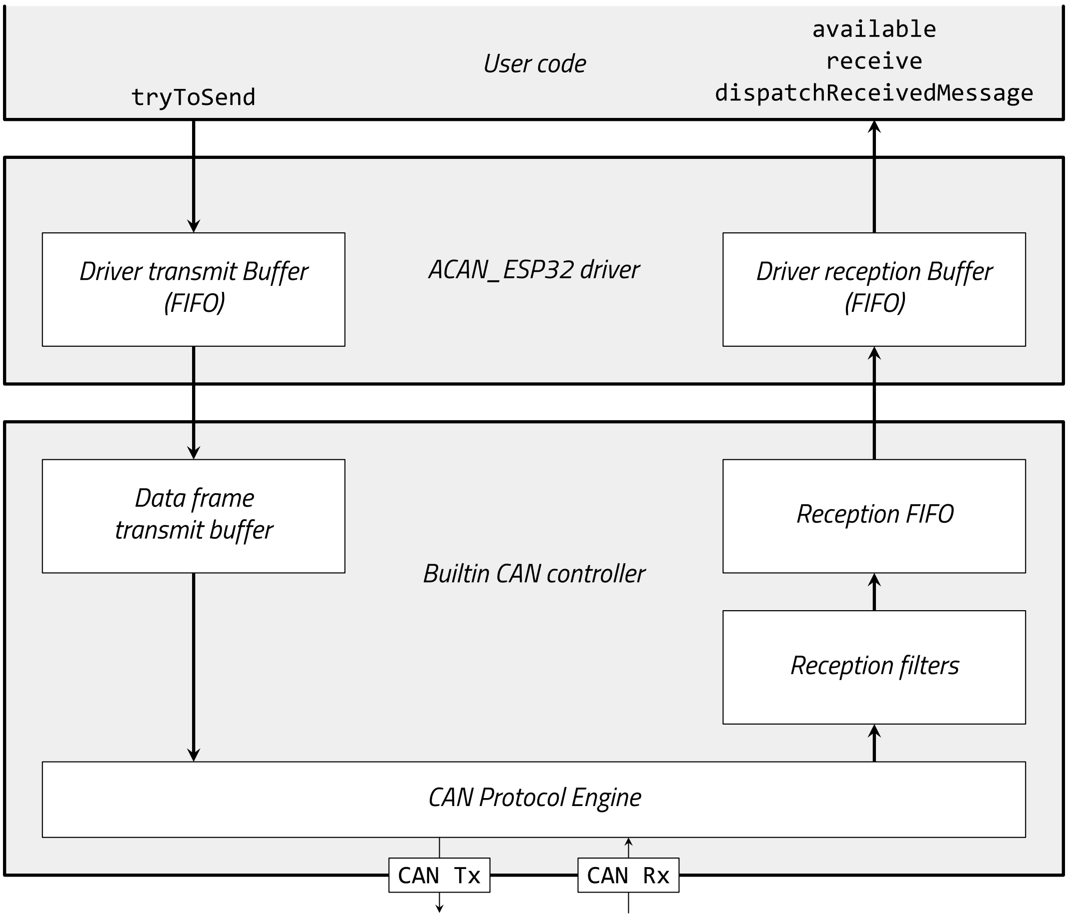

Figure 1 – Message flow in `ACAN_ESP32` driver and Builtin CAN controller

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;

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

- 8 8-bits registers for handling receive filters.

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 12. 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 14 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. 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 10.3 page 17 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 15;

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 12.

The `LoopBackDemo` sketch is:

```
#include <ACAN_ESP32.h>

```

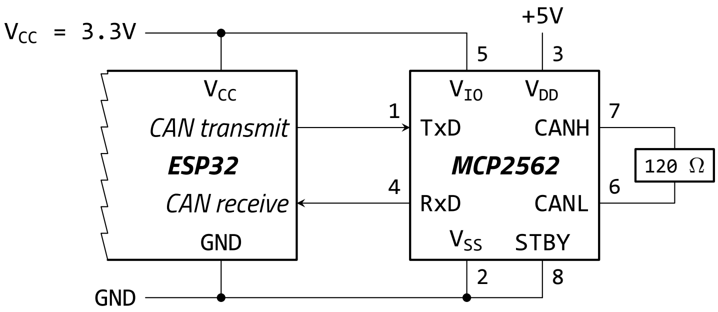

Figure 2 – Connecting an ESP32 to a MCP2562 CAN transceiver

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

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
// Select loopback mode
  settings.mRequestedCANMode = ACAN_ESP32_Settings::LoopBackMode;
//  settings.mRxPin = GPIO_NUM_4 ; // Optional, default Tx pin is GPIO_NUM_4
//  settings.mTxPin = GPIO_NUM_5 ; // Optional, default Rx pin is GPIO_NUM_5
  const uint32_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.mRJW) ;
    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 ("%") ;
    Serial.println ("Configuration␣OK!");
  }else {
    Serial.print ("Configuration␣error␣0x") ;
    Serial.println (errorCode, HEX) ;
  }
}

static uint32_t gBlinkLedDate = 0;
static uint32_t gReceivedFrameCount = 0 ;
static uint32_t gSentFrameCount = 0 ;

void loop() {
  CANMessage frame ;
  if (gBlinkLedDate < millis ()) {
    gBlinkLedDate += 500 ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
    Serial.print ("Sent:␣") ;
    Serial.print (gSentFrameCount) ;
    Serial.print ("\t") ;
    Serial.print ("Receive:␣") ;
    Serial.print (gReceivedFrameCount) ;
    Serial.print ("\t") ;
    Serial.print ("␣STATUS␣0x") ;
    Serial.print (CAN_STATUS, HEX) ;
    Serial.print ("␣RXERR␣") ;
    Serial.print (CAN_RX_ECR) ;
    Serial.print ("␣TXERR␣") ;
    Serial.println (CAN_TX_ECR) ;
    frame.len = 8 ;
    const bool ok = ACAN_ESP32::can.tryToSend (frame) ;
    if (ok) {
      gSentFrameCount += 1 ;
    }
  }
  while (ACAN_ESP32::can.receive (frame)) {
```

```
78       gReceivedFrameCount += 1 ;
79     }
80   }
```

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 12). The section 14.7 page 35 lists all properties you can override.

Line 20, 21. This is the third step, configuration of the `ACAN_ESP32::can` driver with `settings` values. Default CAN `Tx` pin is `GPIO_NUM_5`, default `Rx` pin is `GPIO_NUM_4`; here, you can choose your own pins (see section 8 page 12).

Line 22. 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 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 11 page 18.

Lines 23 to 46. 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 12.2 page 27.

Line 49. The `gBlinkLedDate` global variable is used for sending a CAN message every 0.5 s.

Line 50. The `gReceivedFrameCount` global variable counts the number of received messages.

Line 51. The `gSentFrameCount` global variable counts the number of sent messages.

Line 54. 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 10.

Line 55. It tests if it is time to blink the led, print send and receive counters, and to send a message.

Line 70. Set the message length. In a real code, we set here message data, identifier, and for an extended frame the `ext` boolean property.

Line 71. 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 72 to 74. 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 76. 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 77. 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 ;  // 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 (ESP32 processor is little-endian).

7 Driver instance naming

The ESP32, ESP32S2, ESP32S3 and ESP32C3 microcontroller contain one CAN module, the driver instance name is ACAN_ESP32::can, see section 7.1 page 11. The ESP32C6 contains two CAN modules, see section 7.2 page 11

7.1 Microcontroller with one CAN module

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.

7.2 Microcontroller with two CAN modules

The ESP32C6 contains two CAN modules, named TWAI0 and TWAI1 in the reference manual.

For TWAI0, the driver instance name is ACAN_ESP32::can. The ensures compatibility with ESP32x microcontrollers with one CAN module.

For TWAI1, the driver instance name is ACAN_ESP32::can1. So you write for example:

```
const uint32_t errorCode = ACAN_ESP32::can1.begin (settings) ;
```

If you have written a code for the TWAI0 module and you want to use the TWAI1 module instead, you have to change in your code all ACAN_ESP32::can to ACAN_ESP32::can1.

For making swap between TWAI0 and TWAI1 modules easier, you can declare a C++ reference to the selected driver:

```
ACAN_ESP32 & myCAN = ACAN_ESP32::can1 ; // Do not forget the '&'!
```

[5] The ACAN_ESP32 constructor is declared private.

Then you change all `ACAN_ESP32::can.`x to `myCAN.`x. For example:

```
const uint32_t errorCode = myCAN.begin (settings) ;
```

For using `TWAI0` again, just change:

```
ACAN_ESP32 & myCAN = ACAN_ESP32::can ; // Do not forget the '&'!
```

See for example the `LoopBackDemo-esp32c6` and `LoopBackDemo-Intensive-esp32c6` demo sketchs.

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.

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 Web page[6] *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.

Some boards define both `D`x and `GPIO`y names, as for example `XIAO ESP32-S3` (figure 3). Always use `GPIO_NUM_`y notation. if you want `D4` to be the Tx pin, write:

```
...
settings.mTxPin = GPIO_NUM_5 ; // D4 for XIAO ESP32-S3
...
```

9 Sending frames

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

[6] https://randomnerdtutorials.com/esp32-pinout-reference-gpios/

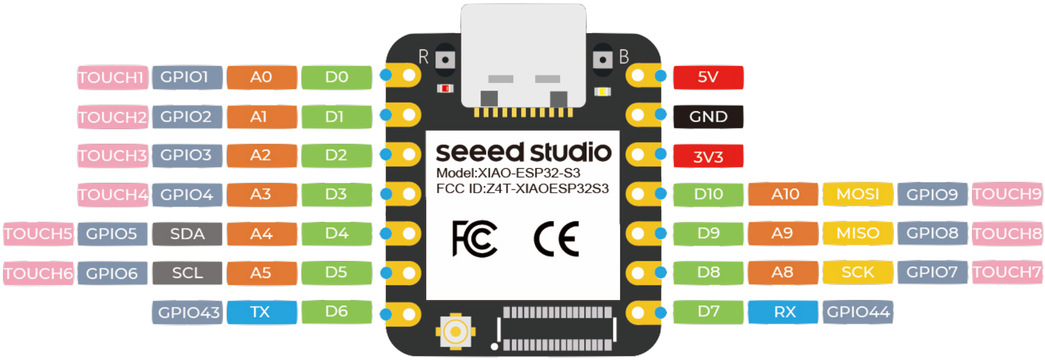

Figure 3 – XIAO ESP32-S3 pins

- `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 `mDriverTransmitBuffer` 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 `CANMessage` 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`.

9.5 The `resetDriverTransmitBufferPeakCount` method

This method assign the current number of messages in the transmit buffer to the peak value of message count in the transmit buffer.

```
ACAN_ESP32::can.resetDriverTransmitBufferPeakCount () ;
```

10 Retrieving received messages using the `receive` method

10.1 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.

10.2 The `available` method

```
  public: bool available (void) const ;
```

The `available` returns `true` if the driver receive FIFO is not empty, and `false` if it is empty.

10.3 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).

10.4 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 () ;
```

10.5 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 () ;
```

10.6 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.re` `BufferPeakCount ()` method return `ACAN_ESP32::can.receiveBufferSize ()+1`.

10.7 The `resetDriverReceiveBufferPeakCount` method

This method assign the current number of messages in the receive buffer to the peak value of message count in the receive buffer.

```
ACAN_ESP32::can.resetDriverReceiveBufferPeakCount () ;
```

11 Filtering received messages

By default, no filtering of received message occurs, that is all network CAN frames are captured and transferred into the hardware CAN 64-byte RxFIFO, and then transferred info the driver receive buffer by the driver.

As `SJA1000`, ESP32 CAN module has 8 bytes dedicaced to received message filtering. This is very little, so the filtering possibilities are very limited.

Six different filters are defined:

- accept only standard frames (section 11.1 page 19, demo sketch: `ESP32CANAcceptOnlyStandardFilterDemo`
- accept only extended frames (section 11.2 page 20, demo sketch: `ESP32CANAcceptOnlyExtendedFilterDemo`
- standard frame single filter (section 11.3 page 20, demo sketch: `ESP32CANSingleStandardFilterDemo`);
- extended frame single filter (section 11.4 page 21, demo sketch: `ESP32CANSingleExtendedFilterDemo`);
- standard frame dual filter (section 11.5 page 23, demo sketch: `ESP32CANDualStandardFilterDemo`);
- extended frame dual filter (section 11.6 page 24, demo sketch: `ESP32CANDualExtendedFilterDemo`).

If none of the above filters work for you, you can set your own (section 11.7 page 25).

A filter demo sketch iterates over:

- all standard data frames with no data (2^{11} frames);
- all standard remote frames (2^{11} frames);
- all extended data frames with no data (2^{29} frames);
- all extended remote frames (2^{29} frames).

The frames are transmitted in this order:

- standard data frame with identifier `0x000`;

- standard remote frame with identifier `0x000`;
- standard data frame with identifier `0x001`;
- standard remote frame with identifier `0x001`;
- ...
- standard data frame with identifier `0x7FF`;
- standard remote frame with identifier `0x7FF`;
- extended data frame with identifier `0x00000000`;
- extended remote frame with identifier `0x00000000`;
- extended data frame with identifier `0x00000001`;
- extended remote frame with identifier `0x00000001`;
- ...
- extended data frame with identifier `0x1FFFFFFF`;
- extended remote frame with identifier `0x1FFFFFFF`.

So it takes a while!

Every minute a progress message is printed.

Every accepted frame is printed. **So a huge number of lines can be printed!** For example, the `ESP32CANAcceptOnlyExtendedFilterDemo` sketch accepts all extended frames, so 2^{29} data frames and 2^{29} remote frames are received and printed.

11.1 Accept only standard frames

This filter accepts any (data and remote) standard frames, and rejects any extended frame.

Demo sketch: `ESP32CANAcceptOnlyStandardFilterDemo`.

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
  ...
  const ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::acceptStandardFrames () ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

11.2 Accept only extended frames

This filter accepts any (data and remote) extended frames, and rejects any standard frame.

Demo sketch: `ESP32CANAcceptOnlyExtendedFilterDemo`.

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
  ...
  const ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::acceptExtendedFrames () ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

11.3 Standard frame single filter

This filter accepts standard frames that pass filter, and rejects any extended frame.

Demo sketch: `ESP32CANSingleStandardFilterDemo`.

The `ACAN_ESP32_Filter::singleStandardFilter` static function prototype is:

```
ACAN_ESP32_Filter singleStandardFilter (const ACAN_ESP32_Filter::Type inType,
                                        const uint16_t inIdentifier,
                                        const uint16_t inDontCareMask) ;
```

The three parameters are:

1. `inType`: you can choose to receive only the data frames (`ACAN_ESP32_Filter::data`), to receive only the remote frames (`ACAN_ESP32_Filter::remote`), or both (`ACAN_ESP32_Filter::dataAndRemot`
2. `inIdentifier`: the value of the identifier of the frames you want to receive; note: as a standard identifier consists of 11 bits, bits 11 to 15 of the supplied value are ignored;
3. `inDontCareMask`: here you specify the `inIdentifier` bits that are ignored for filtering (see examples below); a zero value means only frame with identifier equal to `inIdentifier` matches.

11.3.1 Example 1

I only want to receive standard data frames with an identifier of `0x123`.

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
```

```
  ...
  const ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::singleStandardFilter (
    ACAN_ESP32_Filter::data, 0x123, 0
  ) ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

11.3.2 Example 2

From the previous example, the last parameter is changed to `0x404`.

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
  ...
  const ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::singleStandardFilter (
    ACAN_ESP32_Filter::data, 0x123, 0x404
  ) ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

The 1-bits of `0x404` are the #2 and #10: `0x123` bits 2 and 10 are ignored for filtering. Therefore frames of identifier `0x123`, `0x127`, `0x523` and `0x527` are received.

11.4 Extended frame single filter

This filter accepts extended frames that pass filter, and rejects any standard frame.

Demo sketch: `ESP32CANSingleExtendedFilterDemo`.

The `ACAN_ESP32_Filter::singleExtendedFilter` static function prototype is:

```
ACAN_ESP32_Filter singleExtendedFilter (const ACAN_ESP32_Filter::Type inType,
                                        const uint32_t inIdentifier,
                                        const uint32_t inDontCareMask) ;
```

The three parameters are:

1. `inType`: you can choose to receive only the data frames (`ACAN_ESP32_Filter::data`), to receive only the remote frames (`ACAN_ESP32_Filter::remote`), or both (`ACAN_ESP32_Filter::dataAndRemot`

2. `inIdentifier`: the value of the identifier of the frames you want to receive; note: as a extended identifier consists of 29 bits, bits 29 to 31 of the supplied value are ignored;

3. `inDontCareMask`: here you specify the `inIdentifier` bits that are ignored for filtering (see examples below); a zero value means only frame with identifier equal to `inIdentifier` matches.

11.4.1 Example 1

I only want to receive extended data frames with an identifier of `0x12345678`.

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
  ...
  const ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::singleExtendedFilter (
    ACAN_ESP32_Filter::data, 0x12345678, 0
  ) ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

11.4.2 Example 2

From the previous example, the last parameter is changed to `0x20202`.

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
  ...
  const ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::singleExtendedFilter (
    ACAN_ESP32_Filter::data, 0x12345678, 0x20202
  ) ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

The 1-bits of `0x20202` are the #1 , #9 and #17: `0x12345678` bits 1, 9 and 17 are ignored for filtering. Therefore frames of identifier `0x12345478`, `0x1234547A`, `0x12345678`, `0x1234567A`, `0x12365478`, `0x1236547A`, `0x12365678` and `0x1236567A` are received. The table 1 shows how theses identifier values can be found.

Parameter	Hex value	Binary value
`inIdentifier`	`0x12345678`	`1 0010 0011 0100 0101 0110 0111 1000`
`inDontCareMask`	`0x00020202`	`0 0000 0000 0010 0000 0010 0000 0010`
Accepted identifiers		`1 0010 0011 01x0 0101 01x0 0111 10x0`

Table 1 – `ACAN_ESP32Filter::singleExtendedFilter` filter example

11.5 Standard frame dual filter

This filter accepts standard frames that pass one of the filters, and rejects any extended frame.

The `ACAN_ESP32_Filter::dualStandardFilter` static function prototype is:

```
ACAN_ESP32_Filter dualStandardFilter (const ACAN_ESP32_Filter::Type inType0,
                                      const uint16_t inIdentifier0,
                                      const uint16_t inDontCareMask0,
                                      const ACAN_ESP32_Filter::Type inType1,
                                      const uint16_t inIdentifier1,
                                      const uint16_t inDontCareMask1) ;
```

The six parameters are:

1. `inType0`, `inType1`: you can choose to receive only the data frames (`ACAN_ESP32_Filter::data`), to receive only the remote frames (`ACAN_ESP32_Filter::remote`), or both (`ACAN_ESP32_Filter::dataAndR`
2. `inIdentifier0`, `inIdentifier1`: the value of the identifier of the frames you want to receive; note: as a standard identifier consists of 11 bits, bits 11 to 15 of the supplied value are ignored;
3. `inDontCareMask0`, `inDontCareMask1`: here you specify the `inIdentifier` bits that are ignored for filtering (see examples below); a zero value means only frame with identifier equal to `inIdentifier` matches.

The first three parameters `inType0`, `inIdentifier0` and `inDontCareMask0` define the first filter. The last three parameters `inType1`, `inIdentifier1` and `inDontCareMask1` define the second one. The two filters are independant. A frame is received if it passes one filter (or both).

11.5.1 Example

Demo sketch: `ESP32CANDualStandardFilterDemo`.

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
  ...
```

```
  const ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::dualStandardFilter (
    ACAN_ESP32_Filter::data, 0x123, 0x110,
    ACAN_ESP32_Filter::remote, 0x456, 0x022
  ) ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

For the first filter, the 1-bits of `0x110` are the #4 and #8: `0x123` bits 4 and 8 are ignored for filtering. Therefore standard data frames of identifier `0x023`, `0x033`, `0x123` and `0x133` are received. For the second one, the 1-bits of `0x022` are the #1 and #5: `0x456` bits 1 and 5 are ignored for filtering. Therefore remote data frames of identifier `0x454`, `0x456`, `0x474` and `0x476` are also received.

11.6 Extended frame dual filter

This filter accepts extended frames that pass one of the filters, and rejects any standard frame.

The `ACAN_ESP32_Filter::dualExtendedFilter` static function prototype is:

```
ACAN_ESP32_Filter dualExtendedFilter (const uint32_t inIdentifier0,
                                      const uint32_t inDontCareMask0,
                                      const uint32_t inIdentifier1,
                                      const uint32_t inDontCareMask1 ;
```

The four parameters are:

1. `inIdentifier0`, `inIdentifier1`: the value of the identifier of the frames you want to receive; note: as a standard identifier consists of 29 bits, bits 29 to 31 of the supplied value are ignored; **special case for this filter, the 13 lower bits are also ignored**;
2. `inDontCareMask0`, `inDontCareMask1`: here you specify the `inIdentifier` bits that are ignored for filtering (see examples below); a zero value means only frame with identifier equal to `inIdentifier` matches; as for the previous parameter, bits 0 to 12 and bits 29 to 31 are ignored.

The first two parameters `inIdentifier0` and `inDontCareMask0` define the first filter. The last two parameters `inIdentifier1` and `inDontCareMask1` define the second one. The two filters are independant. A frame is received if it passes one filter (or both).

Unlike other filters, it is not possible to filter by the type (data, remote) of the received frame. Both data and remote extended frames with a given identifier are either accepted, either rejected.

11.6.1 Example

Demo sketch: ESP32CANDualExtendedFilterDemo.

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
  ...
  const ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::dualExtendedFilter (
    0x12345678, 0x00060000, // First filter
    0x19876543, 0x0000A000  // Second filter
  ) ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

The details of the filter operations are shown in table 2. Note that parameter bits 0 to 12 are always ignored by this filter. For example, `inIdentifier0` can have any value between `0x12344000` and `0x12345FFF` without modifying filtering result. Note also that `inDontCareMask1` value is `0x00009000`, the two 1-bits are #12 and #15. The bit #12 is always ignored this filter, therefore `inDontCareMask1` can have any value between `0x00008000` and `0x00009FFF` without modifying filtering result. Finally, this filter accepts data and remote extended frames whose identifiers are of the form `1 0010 0011 0xx0 010x xxxx xxxx xxxx` (2^{15} extended data frames, 2^{15} extended remote frames) or `1 1001 1000 0111 x00x xxxx xxxx xxxx` (2^{14} extended data frames, 2^{14} extended remote frames).

Parameter	**Hex value**	**Binary value**
inIdentifier0	0x12345678	0001 0010 0011 0100 0101 0110 0111 1000
inDontCareMask0	0x00060000	0000 0000 0000 0110 0000 0000 0000 0000
Ignored bits		xxx x xxxx xxxx xxxx
Accepted identifiers		1 0010 0011 0xx0 010x xxxx xxxx xxxx
Parameter	**Hex value**	**Binary value**
inIdentifier1	0x19876543	0001 1001 1000 0111 0110 0101 0100 0011
inDontCareMask1	0x00009000	0000 0000 0000 0000 1001 0000 0000 0000
Ignored bits		xxx x xxxx xxxx xxxx
Accepted identifiers		1 1001 1000 0111 x00x xxxx xxxx xxxx

Table 2 – `ACAN_ESP32Filter::dualExtendedFilter` filter example

11.7 Defining your own filter

If none of the previous filters satisfy you, you can define your own filter. The properties of `ACAN_ESP32Filter` are public, so you can set them as yout want:

```
void setup () {
  ...
  ACAN_ESP32_Settings settings (...) ;
  ...
  ACAN_ESP32_Filter filter = ACAN_ESP32_Filter::acceptAll () ; // Providing a default val
  filter.mACR0 = ... ;
  ...
  filter.mAMR3 = ... ;
  filter.mAMFSingle = ... ;
  filter.mFormat = ... ;
  const uint32_t errorCode = ACAN_ESP32::can.begin (settings, filter) ;
  ...
}
```

Read the *SJ1000 Data sheet*[7], section 6.4.15 from page 44. Section 6.3.9 from page 19 is irrelevant because it is for *basic* mode, but the driver sets the SJA1000 to *pelican* mode.

When the `ACAN_ESP32::begin` method is executed:

- the `mACR0` property of the `filter` parameter is set to the `ACR0` control register;
- ...
- the `mAMR3` property of the `filter` parameter is set to the `AMR3` control register;
- the `mAMFSingle` boolean property of the `filter` parameter is set to the `AFM` bit `MOD` control register.

The `mFormat` boolean property of the `filter` parameter is particular. It does not correspond to any control register, it is handled by the driver. The key point is that the SJA1000 filters are not designed to accept or reject a frame based on its standard or extended format. The contents of the `AMR0`, ..., `ACR3` registers are interpreted differently depending on whether the received frame is standard or extended. Thus, a filter setting always accepts, whatever the value of the `AMR0`, ..., `ACR3` registers, *some* standard frames and *some* extended frames.

When the `ACAN_ESP32::begin` method is executed, the `mFormat` boolean property of the `filter` parameter is set to the `mAcceptedFrameFormat` property of the `ACAN_ESP32` class. This property is only used in the `ACAN_ESP32::handleRXInterrupt` method for accepting or rejecting data or remote frames.

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

12 The ACAN_ESP32::begin method reference

12.1 The ACAN_ESP32::begin method prototype

The begin method prototype is:

```
uint32_t ACAN_ESP32::
begin (const ACAN_ESP32_Settings & inSettings,
       const ACAN_ESP32_Filter & inFilter = ACAN_ESP32_Filter::acceptAll ()) ;
```

The second parameter defines the receive filter and is optional; by default, the pass-all filter is provided.

12.2 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 9 are actually defined by the ACAN_ESP32_Settings class and are also returned by the CANBitSettingConsistency method (see section 14.2 page 33). Bits from 16 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 kBitRatePrescalerIsZero            = 1 <<  0 ;
  public: static const uint16_t kBitRatePrescalerIsGreaterThan64   = 1 <<  1 ;
  public: static const uint16_t kTimeSegment1IsZero                = 1 <<  2 ;
  public: static const uint16_t kTimeSegment1IsGreaterThan16       = 1 <<  3 ;
  public: static const uint16_t kTimeSegment2IsLowerThan2          = 1 <<  4 ;
  public: static const uint16_t kTimeSegment2IsGreaterThan8        = 1 <<  5 ;
  public: static const uint16_t kTimeSegment2Is2AndTripleSampling  = 1 <<  6 ;
  public: static const uint16_t kRJWIsZero                         = 1 <<  7 ;
  public: static const uint16_t kRJWIsGreaterThan4                 = 1 <<  8 ;
  public: static const uint16_t kRJWIsGreaterThanTimeSegment2      = 1 <<  9 ;
```

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

```
  public: static const uint32_t kNotInRestModeInConfiguration        = 1 << 16 ;
  public: static const uint32_t kCANRegistersError                   = 1 << 17 ;
  public: static const uint32_t kTooFarFromDesiredBitRate            = 1 << 18 ;
  public: static const uint32_t kInconsistentBitRateSettings         = 1 << 19 ;
  public: static const uint32_t kCannotAllocateDriverReceiveBuffer   = 1 << 20 ;
  public: static const uint32_t kCannotAllocateDriverTransmitBuffer  = 1 << 21 ;
```

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
  }
  ...
}
```

12.2.1 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 14.1 page 29 for a discussion about CAN bit setting computation.

12.2.2 CAN Bit inconsistent configuration error

This error is raised when you have changed the CAN bit properties (`mBitRatePrescaler`, `mTimeSegment1`, `mTimeSegment2`, `mRJW`), and one or more resulting values are inconsistent. See section 14.2 page 33.

13 Other ACAN_ESP32 methods

13.1 The ACAN_ESP32::statusFlags method

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

This method returns four status flags:

- bit 0: `1` if the hardware receive buffer did overflow, `0` otherwise;

- bit 1: `1` if the driver receive buffer did overflow, `0` otherwise;
- bit 2: `1` if the CAN controller is *bus-off*, `0` otherwise;
- bit 3: `1` if the CAN controller is in *reset* mode, `0` otherwise;
- bits 4-31: always 0.

The value returned is therefore zero when there is no error.

13.2 The `ACAN_ESP32::recoverFromBusOff` method

```
public: bool recoverFromBusOff (void) const ;
```

If the CAN Controller is *bus-off* and in *reset* mode, the method starts recovery and returns `true`. *To return to the Error Active state, the TWAI controller must undergo Bus-Off recovery. Bus-Off recovery requires the TWAI controller to observe 128 occurrences of 11 consecutive Recessive bits on the bus.* For example at 500 kbit/s, The recovery time is therefore greater than or equal to 11 * 128 * 2 μs = 2.816 ms. During recovery, the CAN controller is *bus-off*, but not in *reset* mode. You can test the *bus-off* state using the `statusFlags` method (section 13.1 page 28).

If the CAN Controller is not *bus-off* or not in *reset* mode, the method has no effect and returns `false`.

14 `ACAN_ESP32_Settings` class reference

14.1 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.mRJW) ; // 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:

$$2 \leqslant \texttt{mBitRatePrescaler} \leqslant 128$$

$$\texttt{mBitRatePrescaler} \text{ is even}$$

$$1 \leqslant \texttt{mTimeSegment1} \leqslant 16$$

$$\text{Single sampling: } 2 \leqslant \texttt{mTimeSegment2} \leqslant 8$$

$$\text{Triple sampling: } 3 \leqslant \texttt{mTimeSegment2} \leqslant 8$$

$$1 \leqslant \texttt{mRJW} \leqslant 4$$

$$\texttt{mRJW} \leqslant \texttt{mTimeSegment2}$$

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:

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

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

14.2 The CANBitSettingConsistency method

This method checks the CAN bit decomposition (given by `mBitRatePrescaler`, `mTimeSegment1`, `mTimeSegment2`, `mRJW` 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.

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

```
  public: static const uint16_t kBitRatePrescalerIsZero           = 1 <<  0 ;
  public: static const uint16_t kBitRatePrescalerIsGreaterThan64  = 1 <<  1 ;
  public: static const uint16_t kTimeSegment1IsZero               = 1 <<  2 ;
  public: static const uint16_t kTimeSegment1IsGreaterThan16      = 1 <<  3 ;
  public: static const uint16_t kTimeSegment2IsLowerThan2         = 1 <<  4 ;
  public: static const uint16_t kTimeSegment2IsGreaterThan8       = 1 <<  5 ;
  public: static const uint16_t kTimeSegment2Is2AndTripleSampling = 1 <<  6 ;
  public: static const uint16_t kRJWIsZero                        = 1 <<  7 ;
  public: static const uint16_t kRJWIsGreaterThan4                = 1 <<  8 ;
  public: static const uint16_t kRJWIsGreaterThanTimeSegment2     = 1 <<  9 ;
```

14.3 The actualBitRate method

The actualBitRate method returns the actual bit computed from mBitRatePrescaler, mTimeSegment1, mTimeSegment2, mRJW 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 14.2 page 33), the returned value is irrelevant.

14.4 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 14.2 page 33), the returned value is irrelevant.

There are 22 exact bit rates: 25 kbit/s, 31250 bit/s, 32 kbit/s, 40 kbit/s, 50 kbit/s, 62500 bit/s, 64 kbit/s, 78125 bit/s, 80 kbit/s, 100 kbit/s, 125 kbit/s, 156250 bit/s, 160 kbit/s, 200 kbit/s, 250 kbit/s, 312500 bit/s, 320 kbit/s, 400 kbit/s, 500 kbit/s, 625 kbit/s, 800 kbit/s, 1 Mbit/s.

14.5 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 14.2 page 33), the returned value is irrelevant.

14.6 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 14.2 page 33), the returned value is irrelevant.

14.7 Properties of the ACAN_ESP32_Settings class

All properties of the ACAN_ESP32_Settings class are declared public and are initialized (table 3).

Property	**Type**	**Initial value**	**Comment**
mTxPin	gpio_num_t	GPIO_NUM_5	See section 8 page 12
mRxPin	gpio_num_t	GPIO_NUM_4	See section 8 page 12
mDesiredBitRate	uint32_t	*Initialized by constructor*	See section 14.1 page 29
mBitRatePrescaler	uint8_t	*Initialized by constructor*	See section 14.1 page 29
mTimeSegment1	uint8_t	*Initialized by constructor*	See section 14.1 page 29
mTimeSegment2	uint8_t	*Initialized by constructor*	See section 14.1 page 29
mRJW	uint8_t	*Initialized by constructor*	See section 14.1 page 29
mTripleSampling	bool	*Initialized by constructor*	See section 14.1 page 29
mBitRateClosedToDesiredRate	bool	*Initialized by constructor*	See section 14.1 page 29
mRequestedCANMode	CANMode	NormalMode	See section 14.7.1 page 36
mDriverReceiveBufferSize	uint16_t	32	See section 10.3 page 17
mDriverTransmitBufferSize	uint16_t	16	See section 9.1 page 14

Table 3 – Properties of the ACAN_ESP32_Settings class

14.7.1 The mRequestedCANMode property

This property has three possible values, as described in the table 4. 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 4 – Values of the mRequestedCANMode property of the ACAN_ESP32_Settings class