





# User Manual

# CANBus API

86Duino Coding IDE 500+

CANBus Library

(Version1.0)

# REVISION

| DATE | VERSION | DESCRIPTION |
|---|---|---|
| 2026/1/28 | Version1.0 | New Release. |

# COPYRIGHT

The information in this manual is subject to change without notice for continuous improvement in the product. All rights are reserved. The manufacturer assumes no responsibility for any inaccuracies that may be contained in this document and makes no commitment to update or to keep current the information contained in this manual.

No part of this manual may be reproduced, copied, translated or transmitted, in whole or in part, in any form or by any means without the prior written permission of the ICOP Technology Inc.

©Copyright 2026 ICOP Technology Inc.
Ver.1.0 January, 2026

# TRADEMARKS ACKNOWLEDGMENT

ICOP® is the registered trademark of ICOP Corporation. Other brand names or product names appearing in this document are the properties and registered trademarks of their respective owners. All names mentioned herewith are served for identification purpose only.

For more detailed information or if you are interested in other ICOP products, please visit our official websites at:

- Global: www.icop.com.tw
- USA: www.icoptech.com
- Japan: www.icop.co.jp
- Europe: www.icoptech.eu
- China: www.icop.com.cn

For technical support or drivers download, please visit our websites at:

- https://www.icop.com.tw/resource_entrance

For EtherCAT solution service, support or tutorials, 86Duino Coding IDE 500+ introduction, functions, languages, libraries, etc. Please visit the QEC website:

- QEC: https://www.qec.tw/

This Manual is for the QEC series.

# SAFETY INFORMATION

- Read these safety instructions carefully.
- Please carry the unit with both hands and handle it with caution.
- Power Input voltage +19 to +50VDC Power Input (Typ. +24VDC)
- Make sure the voltage of the power source is appropriate before connecting the equipment to the power outlet.
- To prevent the QEC device from shock or fire hazards, please keep it dry and away from water and humidity.
- Operating temperature between -20 to +70°C/-40 to +85°C (Option).
- When using external storage as the main operating system storage, ensure the device's power is off before connecting and removing it.
- Never touch un-insulated terminals or wire unless your power adaptor is disconnected.
- Locate your QEC device as close as possible to the socket outline for easy access and avoid force caused by the entangling of your arms with surrounding cables from the QEC device.
- If your QEC device will not be used for a period of time, make sure it is disconnected from the power source to avoid transient overvoltage damage.

**WARNING!**



DO NOT ATTEMPT TO OPEN OR TO DISASSEMBLE THE CHASSIS (ENCASING) OF THIS PRODUCT. PLEASE CONTACT YOUR DEALER FOR SERVICING FROM QUALIFIED TECHNICIAN.

# CONTENT

Ch. 1 Introduction ..... 1
1.1 About CANBus Library ..... 2
1.1.1 What is 86Duino IDE? ..... 3
1.2 Protocol Versions ..... 4
1.2.1 CAN 2.0 (Classical CAN) ..... 5
1.3 Communication Overview ..... 6
1.4 Functions Table ..... 7
Ch. 2 Functions ..... 8
begin() ..... 10
init_Mask() ..... 11
init_Filt() ..... 13
beginTransmission() ..... 15
endTransmission() ..... 16
write() ..... 17
requestFrom() ..... 18
read() ..... 19
available() ..... 20
remoteExt() ..... 21
remoteID() ..... 22
sendMsgBuf() ..... 23
readMsgBuf() ..... 24
checkReceive() ..... 25
getCanId() ..... 26
Ch. 3 Example ..... 27
3.1 CANBus Example ..... 28
3.1.1 Receive Check ..... 28
3.1.2 Receive Check Wire ..... 29
3.1.3 Send ..... 30
3.1.4 Send Wire ..... 31
Warranty ..... 32

# Ch. 1

# Introduction

## 1.1 About CANBus Library

This function library is included as part of the 86Duino IDE Coding 103 and later, to support the CAN bus communication interface built-in to the 86Duino One controller board.

CAN is a common communication protocol in the industrial field, able to support highly secured and timing critical control systems. CAN bus is a common interface for automobile and industrial automation equipment.

The CAN bus included in the 86Duino platform includes the following features:

- Support CAN 2.0A and 2.0B standards
- Support up to 1 Mbps data transmission speed
- Support standard (11-bit) and extended (29-bit) data/remote frame transmission
- Provide 3 set of frame buffers
- Extensive error detection and handling mechanism

To simplify the effort to learn and become familiar with this library, the API design for this library is closely resembling the Wire library and CAN bus library from Seeed Studio.




Figure 1.1: CAN bus logo. Source: Wikipedia – *Controller Area Network (CAN)*.
Used for technical reference and identification purposes only.
https://en.wikipedia.org/wiki/CAN_bus

### 1.1.1 What is 86Duino IDE?

The 86Duino integrated development environment (IDE) software makes it easy to write and upload code to 86Duino boards and QEC MDevice. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Arduino IDE, Processing, DJGPP, and other open-source software, which can be downloaded from https://www.qec.tw/software/.




For more information, please visit 86Duino or QEC Website.

## 1.2 Protocol Versions

**Controller Area Network (CAN)** is a multi-master serial bus protocol designed for deterministic and fault-tolerant communication. Originally developed by Robert Bosch GmbH, CAN has evolved into standardized specifications including CAN 2.0A (11-bit identifier) and CAN 2.0B (29-bit identifier), which are widely adopted in automotive and industrial control systems.

This overview is summarized from publicly available information on Wikipedia (*Controller Area Network*): https://en.wikipedia.org/wiki/CAN_bus

A controller area network (CAN) is a vehicle bus standard designed to enable efficient communication primarily between electronic control units (ECUs). Originally developed to reduce the complexity and cost of electrical wiring in automobiles through multiplexing, the CAN bus protocol has since been adopted in various other contexts, such as 3D printing. This broadcast-based, message-oriented protocol ensures data integrity and prioritization through a process called arbitration, allowing the highest priority device to continue transmitting if multiple devices attempt to send data simultaneously, while others back off. Its reliability is enhanced by differential signaling, which mitigates electrical noise.

Common versions of the CAN protocol include CAN 2.0, CAN FD, and CAN XL which vary in their data rate capabilities and maximum data payload sizes. **The CAN bus included in the 86Duino platform support CAN 2.0A and 2.0B standards**

### 1.2.1 CAN 2.0 (Classical CAN)

Due to its legacy, CAN 2.0 is the most widely used protocol with a maximum payload size of eight bytes and a typical baud rate of 500 kbit/s. Classical CAN, which includes CAN 2.0A (Standard CAN) and CAN 2.0B (Extended CAN), primarily differs in identifier field lengths: CAN 2.0A uses an 11-bit identifier, while CAN 2.0B employs a 29-bit identifier. The longer identifier in CAN 2.0B allows for a greater number of unique message identifiers, which is beneficial in complex systems with many nodes and data types. However, this increase in unique message identifiers also increases frame length, which in turn reduces the maximum data rate. Additionally, the extended identifier provides finer control over message prioritization due to more available identifier values. This, however, may introduce compatibility issues; CAN 2.0A devices can generally communicate with CAN 2.0B devices, but not vice versa, due to potential errors in handling longer identifiers. High-speed CAN 2.0 supports bit rates from 40 kbit/s to 1 Mbit/s and is the basis for higher-layer protocols. In contrast, low-speed CAN 2.0 supports bit rates from 40 kbit/s to 125 kbit/s and offers fault tolerance by allowing communication to continue despite a fault in one of the two wires, with each node maintaining its own termination.

## 1.3 Communication Overview

Controller Area Network (CAN) is a multi-master, message-oriented serial communication protocol designed for reliable and real-time data exchange between electronic control units (ECUs). CAN communication follows a layered architecture consisting of the application layer, object layer, transfer layer, and physical layer. The transfer layer defines the core CAN protocol behavior, including message framing, bit timing and synchronization, arbitration, acknowledgement, error detection, and fault confinement. CAN uses a lossless bitwise arbitration mechanism based on dominant and recessive signal states, allowing multiple nodes to transmit simultaneously while ensuring that the message with the highest priority (lowest identifier) gains bus access without data loss.

At the physical layer, CAN employs a differential two-wire bus (CANH and CANL) with termination resistors at both ends to ensure signal integrity and noise immunity. Depending on the application, CAN supports multiple physical layer standards, such as high-speed CAN (ISO 11898-2) and low-speed fault-tolerant CAN (ISO 11898-3).

CAN frames consist of an identifier, control fields, up to 8 data bytes (or up to 64 bytes with CAN FD), and error-checking mechanisms. This architecture makes CAN well suited for deterministic, prioritized, and fault-tolerant communication in automotive, industrial, and embedded control systems.

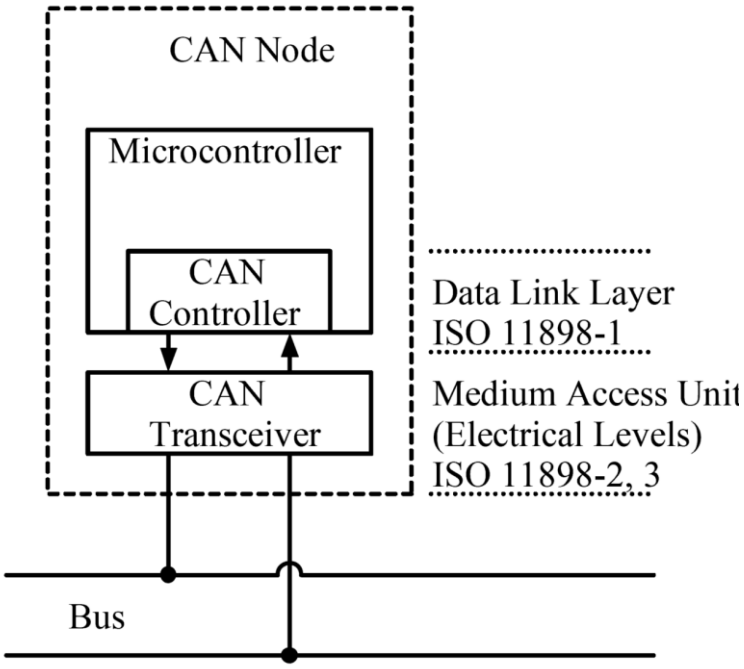



Figure 1.3. CAN node architecture. A CAN node consists of a microcontroller, a CAN controller, and a CAN transceiver. Source: Wikipedia – Controller Area Network (CAN).
https://en.wikipedia.org/wiki/CAN_bus

## 1.4 Functions Table

Following are the briefly description for the CANBus APIs.

- `begin()`: Initialize CANBus function library.
- `init_Mask()`: On an 86Duino platform, there are 32 group of CAN ID mask.
- `init_Filt()`: Function to filter CAN ID.
- `beginTransmission()`: Initialize transmission state.
- `endTransmission()`: End CAN bus data transmission.
- `write()`: Transmit designated data.
- `requestFrom()`: Receive data from CAN bus.
- `read()`: Function to read data from the CAN FIFO buffer.
- `available()`: Function to check whether there are data to read from the CAN interface's FIFO buffer.
- `remoteExt()`: Retrieve data format from incoming data on the CAN bus.
- `remoteID()`: Retrieve the remote CAN bus ID from receiving data.
- `sendMsgBuf()`: Transmit data to remote CAN device.
- `readMsgBuf()`: Function to read data from the CAN bus.
- `checkReceive()`: Check CAN bus' FIFO buffer to see whether there are data to read.
- `getCanId()`: Retrieve CAN bus ID from receiving data, similar to the `remoteID()` function.

# Ch. 2

# Functions

86Duino Coding IDE 500+.

CANBus Library.

The following are the examples and the usage of the APIs of the CANBus library that can work on the 86Duino boards and QEC MDevice.

Functions:

- `begin()`: Initialize CANBus function library.
- `init Mask()`: On an 86Duino platform, there are 32 group of CAN ID mask.
- `init Filt()`: Function to filter CAN ID.
- `beginTransmission()`: Initialize transmission state.
- `endTransmission()`: End CAN bus data transmission.
- `write()`: Transmit designated data.
- `requestFrom()`: Receive data from CAN bus.
- `read()`: Function to read data from the CAN FIFO buffer.
- `available()`: Function to check whether there are data to read from the CAN interface's FIFO buffer.
- `remoteExt()`: Retrieve data format from incoming data on the CAN bus.
- `remoteID()`: Retrieve the remote CAN bus ID from receiving data.
- `sendMsgBuf()`: Transmit data to remote CAN device.
- `readMsgBuf()`: Function to read data from the CAN bus.
- `checkReceive()`: Check CAN bus' FIFO buffer to see whether there are data to read.
- `getCanId()`: Retrieve CAN bus ID from receiving data, similar to the `remoteID()` function.

## begin()

**Description**

Initialize CANBus function library

**Syntax**

`CAN.begin()`

`CAN.begin(speed)`

**Parameters**

- `[in] speed`: Data transmission speed for CAN bus. When not set, the default 10KBPS setting is used.
  - CAN_10KBPS
  - CAN_20KBPS
  - CAN_50KBPS
  - CAN_83K3BPS
  - CAN_100KBPS
  - CAN_125KBPS
  - CAN_250KBPS
  - CAN_500KBPS
  - CAN_833KBPS
  - CAN_1000KBPS

**Returns**

None.

**Example Code**

```
#include <CANBus.h>

unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS); // Configure CAN bus transmission speed to
500KBPS
}

void loop() {
    CAN.beginTransmission(0x00, CAN_STDID);    // Set the CAN device ID to
0x00, using standard data grame.
    CAN.write(buf, 8);                        // Sent 8 bytes of data
    CAN.endTransmission();                    // End transmission
    delay(10);
}
```

## init_Mask()

**Description**

On an 86Duino platform, there are 32 group of CAN ID mask. When there are data from external CAN device, the mask is used to determine the range of device ID (0 ~ N) which their data is to be handled.

Combine with the `init_Filt()` function, this function can limit the range of CAN ID to be handled, within the A ~ B range. When the `init_Mask()` and `ini_Filt()` function are not called within a sketch, the CAN function library is set to accept data from all CAN ID.

**Syntax**

`CAN.init_Mask(num, ext, ulData)`

**Parameters**

- `[in] num`: Mask ID, within the 0 ~ 31 range.
- `[in] ext`: One of the following 4 data transmission format.
  - CAN_STDID: Standard data frame, comply with CAN 2.0A standard. ID range: 0 ~ 0x7FF
  - CAN_EXTID: Extended data frame, comply with CAN 2.0B standard. ID range: 0 ~ 0x1FFFFFFF
  - CAN_STDID_REMOTE: 3. CAN_STDID_REMOTE: Standard remote data frame, comply with CAN 2.0A standard. ID range: 0 ~ 0x7FF
  - CAN_EXTID_REMOTE: 4. CAN_EXTID_REMOTE: Extended remote data frame, comply with CAN 2.0B standard. ID range: 0 ~ 0x1FFFFFFF
- `[in] ulData`: Mask value.
  Refer to the following examples:
  - Without mask:
    0 0 0 0 0 0 0 0 Filter ID value: 0x00
    0 0 0 0 0 0 1 1 Mask: 0x03
    Range of CAN ID to receive data: 0x00 ～ 0x03
  - With mask:
    0 0 0 1 1 0 0 1 Filter ID value: 0x19
    0 0 0 0 0 0 0 1 Mask: 0x01
    Range of CAN ID to receive data: 0x18、0x19

    0 1 0 1 1 1 0 0 Filter ID value: 0x5C
    0 0 0 0 0 1 1 1 Mask: 0x07
    Range of CAN ID to receive data: 0x58 ～ 0x5F

**Returns**

- CAN_FAIL: Setting failed
- CAN_OK: Setting success

**Example Code**

```
#include <CANBus.h>

unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS); // Configure CAN bus transmission speed to
500KBPS
    CAN.init_Filt(0, CAN_STDID, 0x019); // Configure filter ID 0 to 0x019,
using Standard data frame
    CAN.init_Mask(0, CAN_STDID, 0x01); // Configure mask for filter ID 0 to
0x01, using standard data frame
}

void loop() {
    CAN.beginTransmission(0x00, CAN_STDID);    // Set external CAN ID to 0x00,
using standard data frame
    CAN.write(buf, 8);                        // Transmit 8 bytes of data
    CAN.endTransmission();                    // End transmission
    delay(10);
}
```

## init_Filt()

**Description**

Function to filter CAN ID. On the 86Duino platform, there are 32 groups of CAN ID filters. When receiving data from the CAN bus, these filter ID can be used to determine whether the data need to be handled.

Combining with the `init_Mask()` function, receiving data can be limited to CAN ID within the A ~ B range.

When this function is used in a sketch without calling the `init_Mask()` or `init_Filt()` function first, data from all CAN ID will be accepted.

**Syntax**

`CAN.init_Filt(num, ext, ulData)`

**Parameters**

- `[in] num`: Identify filter ID to be used, within the 0 ~ 31 range.
- `[in] ext`: data transmission format. There are 4 different options:
  - CAN_STDID: Standard data frame, comply with CAN 2.0A standard. ID range: 0 ~ 0x7FF
  - CAN_EXTID: Extended data frame, comply with CAN 2.0B standard. ID range: 0 ~ 0x1FFFFFFF
  - CAN_STDID_REMOTE: Standard remote frame, comply with CAN 2.0A standard. ID range: 0 ~ 0x7FF
  - CAN_EXTID_REMOTE: Extended remote frame, comply with CAN 2.0B standard. ID range: 0 ~ 0x1FFFFFFF
- `[in] ulData`: Filter value. Refer to the following example:
  - Without mask:
    0 0 0 0 0 0 1 1 Filter ID value: 0x03
    0 0 0 0 0 0 0 0 Mask: 0x00
    Range of CAN ID to receive data:: 0x03
  - With mask
    0 0 0 1 1 0 0 1 Filter ID value: 0x19
    0 0 0 0 0 0 0 1 Mask: 0x01
    Range of CAN ID to receive data: 0x18、0x19

    0 1 0 1 1 1 0 0 Filter ID value: 0x5C
    0 0 0 0 0 1 1 1 Mask: 0x07
    Range of CAN ID to receive data: 0x58 ～ 0x5F

**Returns**

- CAN_FAIL: Setting failed
- CAN_OK: Setting success

**Example Code**

```
#include <CANBus.h>

unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS); // Configure CAN bus transmission speed to
500KBPS
    CAN.init_Filt(0, CAN_STDID, 0x019); // Configure filter ID 0 to 0x019,
using Standard data frame
    CAN.init_Mask(0, CAN_STDID, 0x01); // Configure mask for filter ID 0 to
0x01, using standard data frame
}

void loop() {
    CAN.beginTransmission(0x00, CAN_STDID);    // Set external CAN ID to 0x00,
using standard data frame
    CAN.write(buf, 8);                        // Transmit 8 bytes of data
    CAN.endTransmission();                    // End transmission
    delay(10);
}
```

## beginTransmission()

**Description**

Initialize transmission state.

*Note: Usage for this function is similar to the `beginTransmission()` function for Wire. Immediately after calling this function, the `write()` and `endTransmission()` functions must be called to complete the transmission task.

**Syntax**

`CAN.beginTransmission(id)`

`CAN.beginTransmission(id, ext)`

**Parameters**

- `[in] id`: External CAN device ID
- `[in] ext`: Data transmission format. This parameter is not needed for the function to work. When not set, the default, CAN_STDID, is used.
  - CAN_STDID: Standard data frame, comply with CAN 2.0A standard. ID range: 0 ~ 0x7FF
  - CAN_EXTID: Extended data frame, based on CAN 2.0B standard. ID range: 0 ~ 0x1FFFFFFF
  - CAN_STDID_REMOTE: Standard data frame, comply with CAN 2.0A standard. ID range: 0 ~ 0x7FF
  - CAN_EXTID_REMOTE: Extended data frame, comply with CAN 2.0B standard. ID range: 0 ~ 0x1FFFFFF

**Returns**

None

**Example Code**

```
#include <CANBus.h>
unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS); // Configure CAN bus transmission speed to
500KBPS
}

void loop() {
    CAN.beginTransmission(0x00); // Set the CAN device ID to 0x00, using
standard data grame.
    CAN.write(buf, 8);           // Sent 8 bytes of data
    CAN.endTransmission();      // End transmission
    delay(10);
}
```

## endTransmission()

**Description**

End CAN bus data transmission.

**Syntax**

`CAN.endTransmission()`

**Parameters**

None

**Returns**

- 0: Indicate function to end transmission is success.
- -1: Indicates function to end transmission failed.

**Example Code**

```
#include <CANBus.h>

unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS); // Configure CAN bus transmission speed to
500KBPS
}

void loop() {
    CAN.beginTransmission(0x00); // Set the CAN device ID to 0x00, using
standard data grame.
    CAN.write(buf, 8);           // Sent 8 bytes of data
    CAN.endTransmission();      //End transmission    delay(10);
}
```

## write()

**Description**

Transmit designated data.

**Syntax**

`CAN.write(val)`

`CAN.write(buf, len)`

**Parameters**

- `[in] val`: data to transmit (unsigned char variable)
- `[in] buf`: data array to transmit (unsigned char array)
- `[in] len`: size of array

**Returns**

Number of byte transmitted

**Example Code**

```
#include <CANBus.h>

unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS); // Set CAN bus transmission speed to 500KBPS
}

void loop() {
    CAN.beginTransmission(0x00); // Set remote CAN device ID to 0x00, using
standard data format
    CAN.write(buf, 8);           // Transmit 8 bytes of data
    CAN.endTransmission();      // End transmission
    delay(10);
}
```

## requestFrom()

**Description**

Receive data from CAN bus.

**Syntax**

`CAN.requestFrom()`

**Parameters**

None

**Returns**

Number of byte received

**Example Code**

```
#include <CANBus.h>
unsigned char ch;

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}

void loop() {
    if (CAN.requestFrom() > 0)         // Check to see whether data received.
    {
        while (CAN.available() > 0)    // Check FIFO to see whether there are
data to retrieve.
        {
            ch = CAN.read();           // Read data from FIFO.
            Serial.print(ch);          // Output data value.
            Serial.print("\t");
        }
        Serial.println();
    }
}
```

## read()

**Description**

Function to read data from the CAN FIFO buffer.

**Syntax**

`CAN.read()`

**Parameters**

None

**Returns**

When there are data in the FIFO buffer, return 1 byte of data.

When there is no data in the FIFO, return -1.

**Example Code**

```
#include <CANBus.h>
unsigned char ch;

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}

void loop() {
    if (CAN.requestFrom() > 0)         // Check to see whether data is
received.
    {
        while (CAN.available() > 0)    // Check to see whether there are data
in the FIFO
        {
            ch = CAN.read();          // Read data from FIFO
            Serial.print(ch);         // Output data value
            Serial.print("\t");
        }
        Serial.println();
    }
}
```

## available()

**Description**

Function to check whether there are data to read from the CAN interface's FIFO buffer.

**Syntax**

`CAN.available()`

**Parameters**

None

**Returns**

Number of byte of data to read from the FIFO buffer.

**Example Code**

```
#include <CANBus.h>
unsigned char ch;

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}

void loop() {
    if (CAN.requestFrom() > 0)         // Check to see whether data is
received.
    {
        while (CAN.available() > 0)    // Check to see whether there are data
in the FIFO
        {
            ch = CAN.read();           // Read data from FIFO
            Serial.print(ch);          // Output data value
            Serial.print("\t");
        }
        Serial.println();
    }
}
```

## remoteExt()

**Description**

Retrieve data format from incoming data on the CAN bus.

**Syntax**

`CAN.remoteExt()`

**Parameters**

None

**Returns**

Received data format.

There are 4 different formats:

1. CAN_STDID: standard data format
2. CAN_EXTID: Extended data format
3. CAN_STDID_REMOTE: standard remote data format
4. CAN_EXTID_REMOTE: Extended remote data format

**Example Code**

```
#include <CANBus.h>

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);

    Serial.print(" CAN data format:");
    Serial.println(CAN.remoteExt());
}

void loop() {
    // ...
}
```

## remoteID()

**Description**

Retrieve the remote CAN bus ID from receiving data.

**Syntax**

`CAN.remoteID()`

**Parameters**

None

**Returns**

CAN bus ID transmitted the data.

**Example Code**

```
#include <CANBus.h>

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);

    Serial.print(" CAN Remote ID:");
    Serial.println(CAN.remoteID());
}

void loop() {
    // ...
}
```

## sendMsgBuf()

**Description**

Transmit data to remote CAN device. (This function is created referencing the CAN bus library from Seeed Studio, Depending on your preferences, you can use this function to transmit data instead of the `beginTransmission()` and `write()` functions.)

**Syntax**

`CAN.sendMsgBuf(id, ext, len, buf)`

**Parameters**

- `[in] id`: Device ID for the remote CAN device
- `[in] ext`: Data transmission format
  - CAN_STDID: Standard data frame, comply with CAN 2.0A standard. ID range: 0 ~ 0x7FF
  - CAN_EXTID: Extended data frame, based on CAN 2.0B standard. ID range: 0 ~ 0x1FFFFFFF
  - CAN_STDID_REMOTE: Standard data frame, comply with CAN 2.0A standard. ID range: 0 ~ 0x7FF
  - CAN_EXTID_REMOTE: Extended data frame, comply with CAN 2.0B standard. ID range: 0 ~ 0x1FFFFFF
- `[in] len`: size of the array
- `[in] buf`: data array

**Returns**

- CAN_OK: Data transmitted
- CAN_FAIL: Data transmission failed.

**Example Code**

```
#include <CANBus.h>
unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup() {
  Serial.begin(115200);
  CAN.begin(CAN_500KBPS);
}

void loop() {
  CAN.sendMsgBuf(0x00, CAN_STDID, 8, buf);   // Transmit data
  // Set remote CAN device ID to 0x00
  // Using standard data format
  // 8 bytes of data
  // buf:  data array
  delay(10);
}
```

## readMsgBuf()

**Description**

Function to read data from the CAN bus. (This function is created referencing the CAN bus library from Seeed Studio, Depending on your preferences, you can use this function to read data instead of the `requestFrom()` and `read()` function.)

*Note: This function is used in conjunction with the `checkReceive()` function.

**Syntax**

`CAN.readMsgBuf(&len, buf)`

**Parameters**

- `[in] len`: Length of data in the buffer (number of byte), an unsigned char variable.
- `[in] buf`: Received data (character array)

**Returns**

- CAN_OK: Data is read
- CAN_NOMSG: Did not read any data

**Example Code**

```
#include <CANBus.h>
unsigned char len = 0;
unsigned char buf[8];

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}

void loop() {
    if(CAN_MSGAVAIL == CAN.checkReceive()) // Check to see whether data is
read
    {
        CAN.readMsgBuf(&len, buf);    // Read data
        for(int i = 0; i<len; i++)    // Output data value
        {
            Serial.print(buf[i]);
            Serial.print("\t");
        }
        Serial.println();
    }
}
```

## checkReceive()

**Desription**

Check CAN bus' FIFO buffer to see whether there are data to read. This function is created referencing the CAN Bus library from Seeed Studio.

*Note: This function must be used with the `readMsgBuf()` function.

**Syntax**

`CAN.checkReceive(void)`

**Parameters**

None.

**Returns**

- CAN_MSGAVAIL: Indicates there are data in FIFO buffer to be read.
- CAN_NOMSG: Indicates there is no data in the FIFO buffer.

**Example Code**

```
#include <CANBus.h>
unsigned char len = 0;
unsigned char buf[8];

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}

void loop() {
    if(CAN_MSGAVAIL == CAN.checkReceive())           // Check to see whether
data is received.
    {
        CAN.readMsgBuf(&len, buf);    // Read data

        for(int i = 0; i<len; i++)    // Output data value
        {
            Serial.print(buf[i]);Serial.print("\t");
        }
        Serial.println();
    }
}
```

## getCanId()

**Description**

Retrieve CAN bus ID from receiving data, similar to the `remoteID()` function.
(This function is created referencing the CAN bus library from Seeed Studio.)

**Syntax**

`CAN.getCanId()`

**Parameters**

None

**Returns**

Device ID.

**Example Code**

```
#include <CANBus.h>

void setup() {
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);

    Serial.print("CAN bus ID:");
    Serial.println(CAN.getCanId());
}

void loop() {
    // ...
}
```

# Ch. 3

# Example

# 3.1 CANBus Example

The following are examples of the CANBus library that can work on the 86Duino boards.

## 3.1.1 Receive Check

```
#include <CANBus.h>

unsigned char len = 0;
unsigned char buf[8];


void setup()
{
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}


void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive())           // check if data coming
    {
        CAN.readMsgBuf(&len, buf);    // read data,  len: data length, buf:
data buf

        for(int i = 0; i<len; i++)    // print the data
        {
            Serial.print(buf[i]);Serial.print("\t");
        }
        Serial.println();
    }
}
```

### 3.1.2 Receive Check Wire

```
#include <CANBus.h>

unsigned char ch;

void setup()
{
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}



void loop()
{
    if (CAN.requestFrom() > 0)         // check if data coming
    {
        while (CAN.available() > 0)
        {
            ch = CAN.read();           // read data
            Serial.print(ch);          // print the data
            Serial.print("\t");
        }
        Serial.println();
    }
}
```

### 3.1.3 Send

```
#include <CANBus.h>

unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup()
{
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}

void loop()
{
    CAN.sendMsgBuf(0x00, CAN_STDID, 8, buf);   // send data:  id = 0x00,
                                               // standrad frame,
                                               // data len = 8,
                                               // buf: data buffer
    delay(10);
}
```

### 3.1.4 Send Wire

```
#include <CANBus.h>

unsigned char buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

void setup()
{
    Serial.begin(115200);
    CAN.begin(CAN_500KBPS);
}

void loop()
{
    CAN.beginTransmission(0x00, CAN_STDID);    // id = 0x00, standrad frame
    CAN.write(buf, 8);                        // buf: data buffer, data len =
8,
    CAN.endTransmission();                    // send data
    delay(10);
}
```

# Warranty

This product is warranted to be in good working order for a period of one year from the date of purchase. Should this product fail to be in good working order at any time during this period, we will, at our option, replace or repair it at no additional charge except as set forth in the following terms. This warranty does not apply to products damaged by misuse, modifications, accident or disaster. Vendor assumes no liability for any damages, lost profits, lost savings or any other incidental or consequential damage resulting from the use, misuse of, originality to use this product. Vendor will not be liable for any claim made by any other related party. Return authorization must be obtained from the vendor before returned merchandise will be accepted. Authorization can be obtained by calling or faxing the vendor and requesting a Return Merchandise Authorization (RMA) number. Returned goods should always be accompanied by a clear problem description.

All Trademarks appearing in this manuscript are registered trademark of their respective owners. All Specifications are subject to change without notice.

©ICOP Technology Inc. 2026