Hochschule Esslingen
University of Applied Sciences

Hochschule Esslingen

CAN Bus communication with Arduino

Using CANdiy shield

Julio César Rodríguez Mejía
Proyect supervisor: Vikas Agrawal

27.06.2014

Index

Objective

Develop a prototype for communication between Arduino and a CAN Bus including wireless communication. In our prototype the wireless connection is achieved via Bluetooth.

Description

Arduino is a common rapid prototyping tool for electronic development, due to the major development of hardware extensions (shields) the main objective is to establish the communication between an Arduino and a CAN Bus with the help of an Arduino shield.

Since Arduino is a microcontroller-based board, after communication with the CAN bus is established, the data in the microcontroller can be manipulated, for instance, sent to another device through wireless or wired connection.

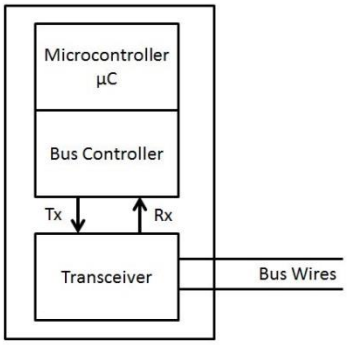


Figure 1 Network components

The general block diagram of the system represents the major network components (Figure 1):

In our prototype, the microcontroller unit will be the Arduino; it has been mentioned that with the information in the microcontroller can be sent or manipulated based on the application.

The Bus controller and the Transceiver will be present in the Arduino shield, the hardware extension to enable the connection and communication with the CAN bus.

Components

The main components for the network had been previously mentioned, nevertheless for testing purposes it is necessary to include other components, most of it, for monitoring purposes during the send and receive phase.

The corresponding components for this project are

- Arduino UNO
- CANdiy Shield for Arduino
- PCAN to USB
- Adaptors and physical connections

Arduino UNO

Defined itself as an “open source electronic prototyping platform”. This platform is popular for its “easy to use” code and development. The microcontroller board is based on the ATMEGA328, an 8-bit microcontroller from ATMEL.

Depending on the version of the board, it features a serial to USB converter with two different options; the previous versions use the FTDI USB-serial driver while the newer revisions use an ATMEGA 16U2 as a USB to serial converter. This USB to serial conversion is used to establish the connection between the computer and the board (to upload any written script).

The Arduino board is used as a programmer and can be delivered within the final model of the developed project; however, there is also the possibility to reduce the number of components (regarding space and cost) for a final product production.

The ATMEGA328 used in the Arduino board comes with a bootloader, in order to be able to understand the code written in the programming tool. Whenever an ATMEGA328 microcontroller is intended to be used as a standalone microcontroller and the application is developed with the Arduino programming software, it is necessary to build the minimum component circuit and load the bootloader in it; afterwards the sketch can be loaded.

Many versions of Arduino, most of them based on ATMEL microcontrollers can be found, the one used in this project is the Arduino UNO.

The following image corresponds to the used Arduino board and its features

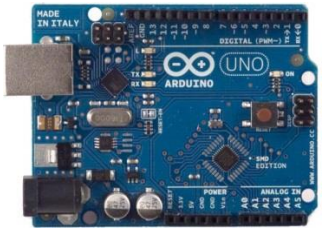


Figure 2 Arduino UNO board

Microcontroller	ATmega328
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz

Figure 3 Arduino UNO features

More information regarding the Arduino boards, software, libraries and bootloader can be found in:

http://arduino.cc/

http://arduino.cc/en/Tutorial/ArduinoToBreadboard

CANdiy shield for Arduino

"Shields" for Arduino are common hardware extensions to enable the Arduino boards perform a different task not featured in the board. In this case, the CANdiy shield enables the Arduino UNO use the CAN protocol for communication.

The CANdiy shield (named as shield in the document) has two main components, a CAN controller and a CAN transceiver. The third most important component is the connection socket, this component is just used for the physical connection to the CAN bus, later explained in detail.

An image of a previous version of the shield is shown here:

Figure 4 CANdiy Shield for Arduino

Documentation, schematics, libraries and examples can be found here:

https://github.com/watterott/CANdiy-Shield

CAN Controller

The CAN controller in the shield is the MCP2515, a standalone CAN controller from Microchip that communicates to a microcontroller through the SPI (Serial Peripheral Interface).

The controller implements CAN specification 2.0 B. Masks and filters can be configured to reduce the overhead in the microcontroller.

For detailed information regarding the CAN controller the datasheet is available in:

http://ww1.microchip.com/downloads/en/DeviceDoc/21801G.pdf

CAN Transceiver

The CAN transceiver is the MCP2551, a high speed transceiver, it is the interface between the controller and the physical bus, this device assures de voltage level corresponding to the specifications (fully compatible with the ISO-11898) in order to perform communication. It also works as a buffer between the peaks from external sources in the bus and the controller.

For detailed information regarding the CAN transceiver the datasheet is available in:

http://ww1.microchip.com/downloads/en/DeviceDoc/21667f.pdf

PCAN to USB

This device is used for testing purposes; the main task is to send and receive CAN messages from and to a USB device, for our testing purpose, a computer. This tool allows us to connect to any CAN network through the DB9 connector.

PCAN comes with software tools for Windows, more information is available in the manufacturer web page:

http://www.peak-system.com/PCAN-USB.199.0.html?&L=1

Figure 5 PCAN to USB adapter

CAN blue II

Figure 6 CAN blue II adapter

This element is a Bluetooth adapter used to connect CAN networks wirelessly. The adapter uses the SPP (Serial Port Profile) service to transmit messages. When two adapters are connected, one should act as a server and the other as a client; the role can be played simultaneously in order to enlarge the network.

In this proyect, the CAN blue II adapter is used as a Gateway for the Arduino; the Arduino is connected to the CAN blue II adapter in order to send and receive messages in a wireless connection.

Information is available in the website of the manufacturer:

http://www.ixxat.com/download/manuals/canblue-ii_manual.pdf

Adaptors and physical connection

The connection socket featured in the shield is the RJ45. It is necessary to check the pin order to establish the connection between a RJ45 CAN connector and a DB9 CAN connector.

We will use DB9 and RJ45 connectors due to the following scheme:

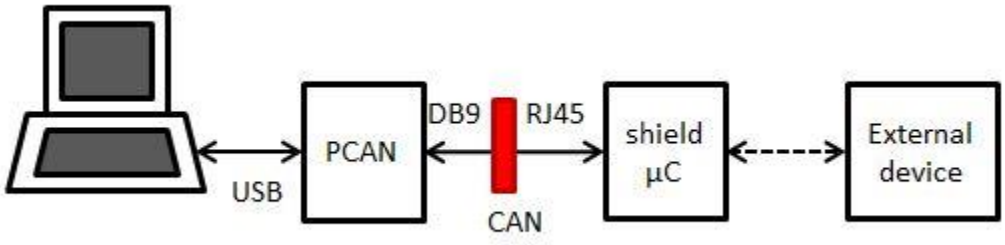
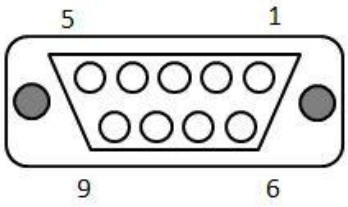


Figure 7 Block diagram, communication

- From the computer we send/receive the data using PCAN, we use USB communication between the computer and PCAN.
- At the output of PCAN we find a DB9 connector and at the input of the shield (with the microcontroller) we find a RJ45 connector, the adaptor is marked as a red block. This communication is through CAN Bus.
- After the µC, with dashed line, we can develop any further available communication protocol with an external device, for instance, UART.

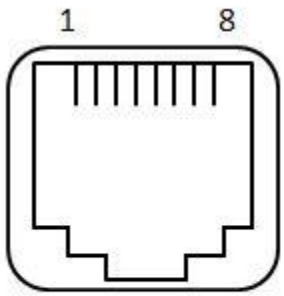
The CAN Bus is compound by two wires, the CAN_H and CAN_L, it can also be implemented by a one-wire Bus for low speed bit rates. The wires arrange in the sockets we will use (DB9 and RJ45) are shown in the following figure.



DB9 Male Connector

DB9 Pin #	Signal Name
1	Reserved
2	CAN_L
3	CAN_GND
4	Reserved
5	CAN_SHLD
6	GND
7	CAN_H
8	Reserved
9	CAN_V+

Figure 8 DB9 CAN pin arrangement



RJ45 Pin #	Signal Name
1	CAN_H
2	CAN_L
3	CAN_GND
4	Reserved
5	Reserved
6	CAN_SHLD
7	CAN_GND
8	CAN_V+

Figure 9 RJ45 CAN pin arrangement

Application

CAN Controller

The CAN controller is the main component in charge of receiving and sending tasks. This component must be configured to the specific behavior the application is aiming to.

Similar to a microcontroller, the configuration of the CAN controller is made by setting or clearing bits in specific registers. The complete data, description and configuration can be found in the datasheet.

Operation mode

As mentioned, the application requires this component to behave according to our objective, the available operation modes are the following:

- Configuration mode.
- Normal mode
- Sleep mode
- Listen-only mode
- Loopback mode

It is always necessary to start with the configuration mode; in this case we can “tune” the CAN controller. This indicates that it is necessary to develop an application with the CAN controller and a microcontroller in charge of the configuration.

Receiving messages

For message reception, the CAN controller features 3 buffers, one for the assembly of the messages (Message Assembly Buffer, MAB) and then the message is forward to the other two buffers.

In order to forward the message from the MAB to the reception buffers (RB0 and RB1), the messages should fulfill the mask and filters previously configured; this configuration to reject messages lowers the processing load of the MCU. The masks and filters for RB0 and RB1 can be different.

The message reception, when passed from MAB to RBx, sets a flag; it also triggers an interrupt and drives low a pin when configured. After the reception, the MCU must clear the flag in order to be able to receive another message.

One configuration mode allows the CAN controller to save a new arrived valid message (mask and filter accepted) for RB0 in RB1 when RB0 has not yet been processed by the MCU; this mode helps prevent message loss.

In order to receive messages the CAN controller should follow an algorithm. This procedure can be resumed by the following task (after configuration):

- Detect a SOF (start of frame) in the CAN bus
- Assembly the message in the MAB (Message Assembly Buffer)
- If the message is accepted (regarding mask and filters) the message is forward to the receive buffer.*
- The message in the receive buffer can be read by the microcontroller via SPI commands

*Details regarding configuration of mask, filters and message forwarding appears later in this document, the complete data is available in the datasheet.

An important feature regarding message reception is the operation mode; with this option it is possible to accept all messages, standard identifier messages, extended identifier messages and even all messages regardless of error. Normally, the reception of all messages regardless of error is used for debugging task, not for an actual system.

Sending messages

In order to send messages through the CAN bus, the CAN controller should receive the request for sending the message. To assembly the messages the controller has 3 buffers compound by 14 bytes. The first byte corresponds to the control bits, 5 bytes for the identifier (standard or extended) and 8 bytes available for data.

For transmission; the obligatory registers to be loaded are the identifier, standard (TXBnSIDH, TXBnSIDL) or extended (TXBnEIDm) and the data length (TXBnDLC).

Message transmission includes priority; when many messages are request to send, the priority in the control register is checked. 2 bits are available for priority assignment where 11 is the highest priority and 00 the lowest; in case two buffers have the same priority, the higher buffer number is selected.

In order to send a message the action must be request by the MCU, there are three ways the action can be performed:

- Writing the TXBnCRTL.TXREQ register via SPI (possible to set at same time as priority)
- Sending the SPI RTS command
- Setting TXnRTS pin, the corresponding buffer to be sent.

After the request, the message could not be immediately sent; the bit acts as a flag for the CAN controller waiting for the bus to be available and start sending the message. After sending the message the CANINTF.TnIF is set and an interrupt can be generated if configured.

There are specific registers to indicate when the message transmission has errors or is lost, in that case the send flag remains set, in order to retry to send the message; just in case of one-shot operation, the message is sent just once.

Message transmission can also be aborted; by means of the MCU, either clearing the corresponding buffer TXREQ flag or even aborting all messages setting CANCTRL.ABAT. When aborting all messages, a flag is triggered to announce the action.

Bit Timing

Can Controller

CAN communication uses NRZ-coding so it is necessary to use a PLL (Phase Lock Loop) to synchronize bit edges and adjust the clock. Bit-stuffing is also necessary to ensure the occurrence of an edge in a defined period of time and maintain synchronization. The MCP2515 uses a DPLL (Digital Phase Lock Loop) for transmission and reception clock synchronization.

CAN Bit time

In the CAN bus all nodes should have the same bit rate in order to ensure the communication. Since the clock signal is not transmitted the receiver nodes need to synchronize to the transmitter clock.

Common terms in the CAN communication are:

Nominal Bit Rate; number of bits per second transmitted by an ideal transmitter without resynchronization. NBR, frequency in bits/s.

$$NBR = \frac{1}{t_{bit}} = f_{bit}$$

Nominal Bit Time; the time necessary for a bit transmission, is formed by 4 segments.

$$t_{bit} = t_{SyncSeg} + PropSeg + PS1 + PS2$$

TQ; Time Quanta, individual minimal period of time. Each bit segment is formed by a defined number of TQ.

The length of a TQ depends on the oscillator period and the programmable Baud Rate preescaler (BRP).

$$TQ = \frac{2}{f_{osc}} * BRP$$

Synchronization Jump Width SJW; for synchronization purposes a defined amount of TQ to enlarge or shorten the bit time.

Information Processing Time IFT; the time needed to determine the logical value of a bit after being sampled.

CAN Bit Segments

The segments which form the CAN bit have a fixed length in order to maintain the synchronization and correct communication.

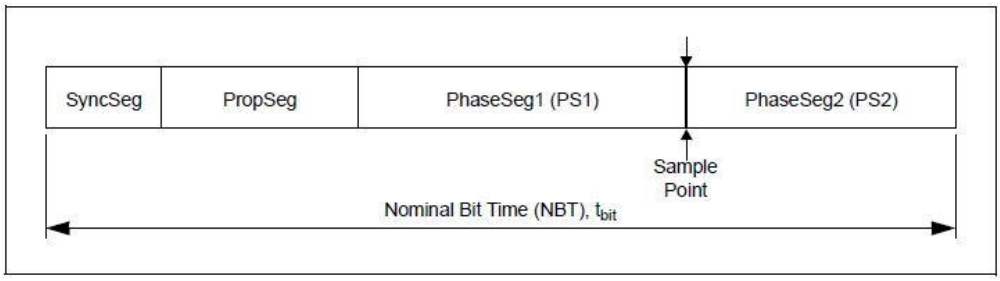


Figure 10 CAN Bit Segments

The Synchronization Segment consists of just one TQ; an edge is expected in this segment.

Propagation Segment to compensate physical delay within the nodes, duration between one and eight TQ.

Phase Segments (1 and 2) they are used to compensate phase errors and synchronize. The sample point is at the end of PS1 and determines the start of PS2. The duration of PS1 is between 1 and 8 TQ; PS2 between 2 and 8 TQ. When the sample is configured to be 3-time sample, 2 samples are taken TQ/2 before.

The IPT determines the value of PS2; for the MPC2515 IPT is two times TQ resulting in a minimum PS2 of 2 TQ, the value of PS2 should be at least equal to IPT.

Synchronization

Since we are expecting an edge in the Synchronization segment, after it, PS1 and PS2 are adjusted to match the specific transmission rate.

Synchronization is achieved by two forms; Hard Synchronization or resynchronization.

Hard Synchronization occurs with a recessive-to-dominant edge during a BUS-Idle state, it indicates the start of a message. By this way we force the edge to be in the Synchronization Segment and the time constrains for every following segment are met.

Resynchronization results in making PS1 bigger or PS2 smaller, the time of TQs to be modified is bounded by the SJW value. The bit-stuffing states that, there can be just a fixed number of bits with the same value; resynchronization is ensured.

To make the decision to enlarge or shorten, the moment when the edge occurs is compared to the Sync segment resulting in 3 different cases:

e = 0; edge in Sync Segment

e > 0; edge before Sample Point; PS1 lengthened (slower transmitter)

e < 0; edge after Sample Point; PS2 shortened (faster transmitter)

Arduino Uno

The microcontroller in charge of the configuration, control and processing of the CAN messages is the Arduino. As mentioned in the section of the CAN controller, it is necessary to combine the CAN controller with a MCU. Configuration actions and message processing are possible with the use of third-parity software libraries.

To establish the communication between the Arduino and the CAN controller, the SPI interface is used. Arduino has already an SPI library, intended to communicate with SPI devices; however, the use of the third-party library helps us saving time in code writing; they include the register values that should be modified for a specific task in the CAN controller.

Regarding the libraries; it is common to find two different types of files, one file with the .h extension and one file with the .cpp extension. The h file corresponds to the definition of all the methods to be implemented, just the definition with parameters and return type; it can also include some value definition for constants. The cpp file corresponds to the code, where the methods are fully implemented, so, whenever we use one of this libraries we will find at the beginning of the sketch (code) the #include <library.h>, indicating the library which methods will be used. A simplified explanation is; the #include statement is equivalent as copying the code at the beginning of the sketch.

The Arduino sketch, code to be executed, normally has the following elements:

- Headers
- Variable and constants declaration
- Setup method
- User defined functions
- Main loop

The headers section, previously mentioned, with the #include, is equivalent as copying the code of a library in the current sketch.

The variable and constants declaration is used to define values or pin numbers of the Arduino Board which will be used or modified in the code.

The setup method (void Setup ()) is the section of code which is only executed once at the start-up of the system. Here any configuration is included, regarding the configuration of any other external element of the Arduino (shields) or any peripheral from the microcontroller, for instance, the SPI interface or the UART interface.

User defined functions are functions, either with a returned value or, as the Setup function, with a void return value and with parameters, in the case they are needed. The declaration of the function is similar as the ones in Java. The reason this functions are declared before the main loop section is, at compilation time, if the functions are unknown we will receive an error message. Another solution is, similar to Java, include just the declaration of the function (firma) before the main loop section so at compilation time there is no error, afterwards the full function implementation is to be found.

The main section of the sketch is the main loop, this section is the one to be executed permanently, after configuration at start-up, and where the main processing or tasks should be done.

CAN Blue II

Since we aim to establish a wireless communication with Arduino we use Bluetooth (BT from now on). The CAN Blue adapter can establish a wireless connection with any Bluetooth device, in our testing environment the adapter is paired with a computer, in order to send CAN messages and verify the received messages sent by Arduino.

The communication method was mentioned before, the BT adapter uses a SSP (Serial Port Profile), a new COM Port is available in the debugging computer. The proper installation of the drivers and connection steps are available in the manufacturer's datasheet/manual.

CAN Blue II includes preprogrammed examples to test the communication. The most important point here is to achieve a stable connection with the BT adapter. The COM Port association differs in every computer and somehow it can represent an issue during testing phase. After establishing the serial communication with the Configuration Port (one of the two BT services offered by the adapter) we can use the example CanBlueCon_CAN_RX_TX_Demo; however regarding some configuration parameters for the CAN communication, is necessary to understand what the example is doing.

The main tasks of this example are:

- Disable any filter in order to receive all messages (standard and extended ID)
- Establish a communication Baudrate
- Start the CAN controller
- Send a predefined CAN message periodically

After checking this with the user manual, some minor modifications are needed to establish the communication with the Arduino board. Details can be found later in the test section.

Test

The following section shows a short description of the performed tests. A diagram is included to show a rough sketch of the communication scheme. The connection between the shield/µC and the computer is marked with dots because it is strictly not necessary; nevertheless it can be a debugging function in order to ensure the proper behavior of the prototype. During test phase this connection provides the power source for the Arduino/shield board.

Message transmission

The first test to be done with the system is to send a CAN message. Within the correct configuration of baudrate, we should be able to see the message in the PCAN software.

The corresponding block diagram of the test is shown

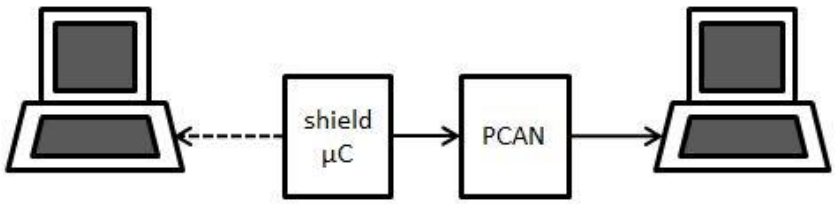


Figure 11 Connection for transmission test

The test corresponds to program the Arduino to send a message within a defined interval of time, and then the PCAN will receive the message and display it in the screen of the corresponding computer.

The code is basically the one included as example of the library in use. The main purpose of this test was to verify the communication between Arduino and the CAN controller; also the correct physical connection.

The code is show:

```
// demo: CAN-BUS Shield, send data

#include <mcp_can.h>
#include <SPI.h>


unsigned char txBuf[8] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0x0F, 0x0F};
long unsigned int txId = 0x7FF;



void setup()
{
    Serial.begin(9600);

    CAN.begin(CAN_100KBPS);                          // init can bus : baudrate = 100k

}


void loop()
{


    CAN.sendMsgBuf(txId, 0, 8, txBuf);   // send data:  id = 0x7FF, standard frame,
data len = 8, txBuf: data buf
    delay(1000);                         // delay in uS to send the message again,every
1 second.
}
```

Code 1 Message Transmission

Message reception

In order to complete the communication we need also to receive messages an here we will find one of the most important task regarding message handling in the CAN Bus system; message filtering.

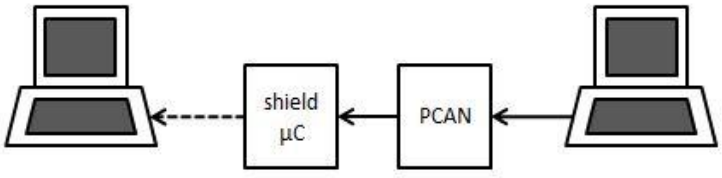


Figure 12 Message reception test

The main purpose of filtering the messages is to reduce the processing load to the microcontroller. Filtering can be done by software in the microcontroller; nevertheless this will result in a higher load for it. To reduce this load and avoid the software filtering, we can configure our CAN controller to accept or reject defined messages via ID filtering.

Message filtering

The MCP2515 controller has two different receive buffers; this buffers accept a mask configuration and certain filter configurations. Buffer 0 has priority and it has only 2 filters, buffer 1 has 4 filters.

Regarding the message filtering with the CAN-controller we need to remark two different terms; mask and filter.

The mask, value configured in the CAN-controller indicating which bits in the 11-bit identifier should be compared with the value in the filters. The bits to be compared are set.

The filter, value configured in the CAN-controller indicating the expected value ID of the incoming message. With to filters we can configure a range of IDs to be accepted; one filter indicates the lower ID to be accepted while the other indicates the higher bound.

A truth table can be found in the CAN controller datasheet in order to show the behavior regarding masks and filters configuration; the following figure was directly taken from the datasheet.

TABLE 4-2: FILTER/MASK TRUTH TABLE

Mask Bit n	Filter Bit n	Message Identifier bit	Accept or Reject bit n
0	x	x	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

Note: x = don't care

Figure 13 Message Filtering Truth Table

The corresponding tests are:

- Configure masks and filters to accept only a defined ID
- Configure masks and filters to accept a range of IDs
- Configure masks and filters to accept defined IDs, one for each filter, 6 IDs in total

Acceptance of defined ID (ID 123 and 111)

Buffer 0	11-bit Standard Identifier										
Mask	1	1	1	1	1	1	1	0	0	0	0
Filter 0	0	0	1	0	0	1	0	0	0	1	1
Filter 1	0	0	1	0	0	0	1	0	0	0	1

Buffer 1	11-bit Standard Identifier										
Mask	1	1	1	1	1	1	1	1	1	1	1
Filter 2	0	0	0	0	0	0	0	0	0	0	0
Filter 3	0	0	0	0	0	0	0	0	0	0	0
Filter 4	0	0	0	0	0	0	0	0	0	0	0
Filter 5	0	0	0	0	0	0	0	0	0	0	0

Figure 14 Masks and Filters for specific IDs

With this Mask and Filter configuration, the controller only accept the defined specific IDs for buffer 0 and any message with ID 0x000 in buffer 1, practically, no messages.

The code corresponding to this test is shown:

```
// demo: CAN-BUS Shield, receive data
// and send it back, just for desired ID
// echo for the desired message ID
#include <mcp_can.h>
#include <SPI.h>

long unsigned int rxId;
unsigned char len = 0;
unsigned char rxBuf[8];

void setup()
{
  Serial.begin(9600);

  CAN.begin(CAN_500KBPS);                       // init can bus : baudrate = 500k
  pinMode(2, INPUT);                            // Setting pin 2 for /INT input

  //Configure the CAN controller
  //configure mask and filters for the messages
  //we receive only std id with id 123 or 111
  CAN.init_Mask(0, 0, 0x07FF);

  //Buffer 0
  CAN.init_Filt(0, 0, 0x0123);
  CAN.init_Filt(1, 0, 0x0111);

  //Buffer 1
  CAN.init_Mask(1, 0, 0x07FF);
  CAN.init_Filt(2, 0, 0x0000);
  CAN.init_Filt(3, 0, 0x0000);
  CAN.init_Filt(4, 0, 0x0000);
  CAN.init_Filt(5, 0, 0x0000);
}

void loop()
{
  //checkReceive fcn, return 3 when message is availalbe
  //returns 4 when no message arrived (check library code)
  if(CAN.checkReceive()==3){
     CAN.readMsgBuf(&len, rxBuf);    // Read data: len = data length, buf = data
byte(s)
      rxId = CAN.getCanId();         // Get message ID
      Serial.print("ID: ");
      Serial.print(rxId, HEX);
      Serial.print("  Data: ");
      for(int i = 0; i<len; i++)     // Print each byte of the data
      {
        if(rxBuf[i] < 0x10)          // If data byte is less than 0x10, add a
leading zero
        {
          Serial.print("0");
        }
        Serial.print(rxBuf[i], HEX);
        Serial.print(" ");
      }
      Serial.println();
      CAN.sendMsgBuf(rxId, 0, 8, rxBuf);   // send data:  id = rxID, standrad frame,
data len = 8, stmp: data buffer (received data)
    }
}
```

Code 2 Filter configuration for message reception

The configuration regarding a range of IDs to be accepted is shown:

Buffer 0	11-bit Standard Identifier										
Mask	1	1	1	1	1	1	1	0	0	0	0
Filter 0	0	0	1	0	0	0	1	0	0	0	0
Filter 1	0	0	1	0	0	1	0	0	0	0	0

Buffer 1	11-bit Standard Identifier										
Mask	1	1	1	1	1	1	1	1	1	1	1
Filter 2	0	0	0	0	0	0	0	0	0	0	0
Filter 3	0	0	0	0	0	0	0	0	0	0	0
Filter 4	0	0	0	0	0	0	0	0	0	0	0
Filter 5	0	0	0	0	0	0	0	0	0	0	0

Figure 15 Masks and Filters for a range of IDs

The configuration, compared to the previous one, differs in the last for bits of the identifier. We set the Mask not to compare this last 4 bits of the ID, resulting in a range from 110 to 12F. Filter 0 ensures 11X to be accepted; Filter 1 ensures 12X to be accepted. Buffer 1 is again configured to reject all messages.

The code is shown:

```
// demo: CAN-BUS Shield, receive data
// recieve IDs from 110 to 12F
// the last 4 bits are not filtered/checked
#include <mcp_can.h>
#include <SPI.h>

long unsigned int rxId;
unsigned char len = 0;
unsigned char rxBuf[8];

void setup()
{
  Serial.begin(9600);

  CAN.begin(CAN_100KBPS);                       // init can bus : baudrate = 100k
  pinMode(2, INPUT);                            // Setting pin 2 for /INT input

  //Configure the CAN controller
  //configure mask and filters for the messages
  //we receive only std id with id from 110 to 12F
  CAN.init_Mask(0, 0, 0x07F0);

  //Buffer 0
  CAN.init_Filt(0, 0, 0x0110);
  CAN.init_Filt(1, 0, 0x0120);

  //Buffer 1
  CAN.init_Mask(1, 0, 0x07F0);
  CAN.init_Filt(2, 0, 0x0110);
  CAN.init_Filt(3, 0, 0x0110);
  CAN.init_Filt(4, 0, 0x0120);
  CAN.init_Filt(5, 0, 0x0120);

}

void loop()
{
  //checkReceive fcn, return 3 when message is availalbe
  //returns 4 when no message arrived (check library code
  if(CAN.checkReceive()==3){
     CAN.readMsgBuf(&len, rxBuf);      // Read data: len = data length, buf = data
byte(s)
      rxId = CAN.getCanId();           // Get message ID
      Serial.print("ID: ");
      Serial.print(rxId, HEX);
      Serial.print("  Data: ");
      for(int i = 0; i<len; i++)       // Print each byte of the data
      {
        if(rxBuf[i] < 0x10)            // If data byte is less than 0x10, add a leading
zero
        {
          Serial.print("0");
        }
        Serial.print(rxBuf[i], HEX);
        Serial.print(" ");
      }
      Serial.println();
      CAN.sendMsgBuf(rxId, 0, 8, rxBuf);// send data:  id = rxID, standard frame, data
len = 8, stmp: data buffer (received)
    }
}
```

Code 3 Group Filtering Code

To maximize the use of the Masks and Filters in the CAN controller, we can specify 6 different IDs to be accepted; each of them corresponding to a Filter. The specified IDs are 110, 111; for Buffer 0. Buffer 1 will accept 115, 120, 123 and 130. We can specify any ID just writing its value in the filter.

Buffer 0	11-bit Standard Identifier										
Mask	1	1	1	1	1	1	1	0	0	0	0
Filter 0	0	0	1	0	0	0	1	0	0	0	0
Filter 1	0	0	1	0	0	0	1	0	0	0	1

Buffer 1	11-bit Standard Identifier										
Mask	1	1	1	1	1	1	1	1	1	1	1
Filter 2	0	0	1	0	0	0	1	0	1	0	1
Filter 3	0	0	1	0	0	1	0	0	0	0	0
Filter 4	0	0	1	0	0	1	0	0	0	1	1
Filter 5	0	0	1	0	0	1	1	0	0	0	0

Figure 16 Configuration for 6 different IDs

As the previous codes, the task is to verify if the ID corresponds to the ones to be accepted, when the ID is accepted, we return the message with the same data (echo). The code is shown just for the configuration section; libraries, variables and the sending and receiving procedure remains the same:

```
void setup()
{
  Serial.begin(9600);

  CAN.begin(CAN_100KBPS);                    // init can bus : baudrate = 100k
  pinMode(2, INPUT);                         // Setting pin 2 for /INT input

  //Configure the CAN controller
  //configure mask and filters for the messages
  //we receive only std id with id 110, 111, 115, 120, 123, 130
  CAN.init_Mask(0, 0, 0x07FF);

  //Buffer 0
  CAN.init_Filt(0, 0, 0x0110);
  CAN.init_Filt(1, 0, 0x0111);

  //Buffer 1
  CAN.init_Mask(1, 0, 0x07FF);
  CAN.init_Filt(2, 0, 0x0115);
  CAN.init_Filt(3, 0, 0x0120);
  CAN.init_Filt(4, 0, 0x0123);
  CAN.init_Filt(5, 0, 0x0130);
}
```

Code 4 Filter for 6 different IDs

Message reception with CAN Blue II

The first test with the Bluetooth adapter corresponds to send a message via BT and display with the Arduino UART. The BT adapter is paired with a computer and via the virtual COM port we send the corresponding command to send a CAN message.

The Arduino executed sketch already has the filter configuration, so the resulting behavior is the same as the one with the test with the PCAN. In fact, the only difference is that the source of the message is not a wired CAN interface, it is wireless.

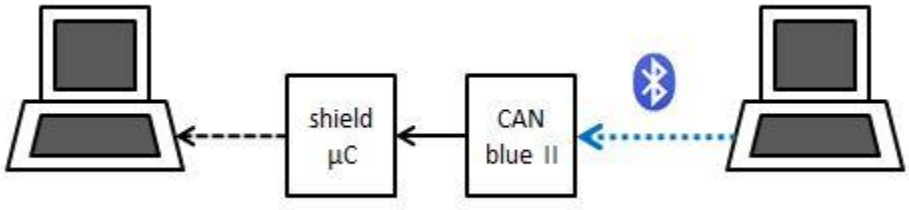


Figure 17 Message reception with BT

The message transmission with the BT adapter can be achieved by two different ways. The first one is to open the serial communication tool provided by the manufacturer and manually type the command to send the CAN message. The second is to use the preprogrammed example so the message will be sent periodically.

As mentioned in the application section, a few modifications should be made to achieve the expected behavior, in concrete, the Baudrate configuration and the message.

These modifications are made in a Batch file in the examples folder after the installation of the drivers. A batch file is the automatic way of typing commands in the command line in Windows.

Some useful commands for the use of the CAN Blue are:

- c config show shows the current CAN configuration
- c can_init xxx configures the baudrate for the CAN communication where xxx is the speed kbps
- c can_start starts the CAN controller
- m sdx IDX 00 00 00 00 00 00 00 00 sends a CAN message where sd indicates standard frame, x indicates the number of data bytes, IDX the 11-bit ID and the data field is represented with 00 for each byte. The number of data bytes should correspond to the x in sdx (1 to 8)

Due to the application, the configuration and programming of the CAN Blue does not go deeper, just to send a standard identifier CAN message and receive all the available messages in the Bus.

The corresponding code used in the CAN Blue adapter is shown:

```
#print ########################################
#print ## Demo: CANblue CAN RX / TX Demo     ##
#print ########################################

#print
#print Showing CANblue Generic Version
d version
#delay 0.5

#print
#print Erasing STD filter list
c filter_clear STD

#print
#print Erasing EXT filter list
c filter_clear EXT

#print
#print Disabling STD filter list
c filter_disable STD

#print
#print Disabling EXT filter list
c filter_disable EXT

#print
#print Initializing CAN with 1000 kbaud
c can_init 500

#print
#print Starting CAN controller
c can start

#print Start loop demo
#pause

#label L1
#print
m sd8 123 1 2 3 0 0 0 0 0
#delay 1
#goto L1
```

Code 5 CAN Blue Example text file

```
@echo off

echo Start CanBlueCon on COM Port 5 with CanBlueCon_CAN_RX_TX_Demo.txt file
..\CanBlueCon 5 CanBlueCon CAN RX TX Demo.txt
Pause
```

Code 6 CAN Blue Example bat file

The code was modified just in the baudrate, keeping the default labels, to 500 kbps and the message to be sent configured to be ID: 123 data 1 2 3 0 0 0 0, so this message is accepted by the Arduino due to the previously explained tests. The corresponding .bat file should indicate the COM port in which the Config SPP (Serial Port Profile) of the CAN Blue II adapter was mapped in the computer. This number is indicated before the name of the text file.

Arduino sending control commands

Until this section the test consider the proper connection, configuration and communication over CAN; nevertheless with a microcontroller, like Arduino, we are able to use another communication tools; the debugging part was always supported by the UART interface, that means the UART interface was already running at the same time as the CAN interface.

With this advantage and the Bluetooth communication (wireless) we aim to the following:

- A mobile or generic BT device establishes connection with the BT CAN adapter
- The mobile device sends information via CAN messages
- Arduino is in charge of the message processing
- Arduino is in charge of sending control commands to another component via UART

The performed test with Arduino was the correct transmission of control commands via UART interface. The important situation here is regarding the transmitted bits.

Whenever a bit is transmitted with Arduino over serial it is automatically converted to its ASCII value. Normally the serial communication aims to text transmission and with the ASCII values is easier to visualize the transmitted data.

However the control commands were not intended to be read, they are just a bit stream which can be understood by the other device. Taking this into account, the correct Arduino function is ***Serial.write*** instead of ***Serial.print.***

The code is shown:

```
//Send control command over RS232

unsigned char start_cmd[10]= {0x0D, 0x01, 0x53, 0x31, 0x6E, 0x0D, 0x01, 0x53, 0x31, 0x6E};

unsigned char stop_cmd[10] = {0x0D, 0x01, 0x43, 0x31, 0x7E, 0x0D, 0x01, 0x43, 0x31, 0x7E};


void setup()
{
  //baudrate configuration for Serial communication
  Serial.begin(2400);
}

void loop()
{
  //sending the fist command
  for(int i = 0; i<10; i++){
    Serial.write(start_cmd[i]);
  }
  delay(3000);

  //after a 3s delay, send second command
  for(int i = 0; i<10; i++){
    Serial.write(stop_cmd[i]);
  }
  delay(3000);
}
```

Code 7 Arduino Control Commands over Serial

The serial communication in Arduino at the same time as the CAN communication can be shown as the following:

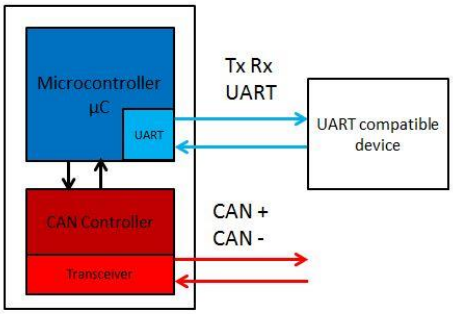


Figure 18 Arduino with CAN and UART communication

A remark in the physical connection of the UART serial is, the Tx and Rx pins are crossed. That it, for instance, the Arduino Tx is connected to the Rx pin in the other device and vice versa.

Integration

The final task in order to apply the previously performed test is to integrate them to a final example application. Also this integration task requires the review of the different test codes and, where it is possible; create function calls to show a cleaner main task for execution.

The final Arduino sketch includes the following:

- CAN controller configuration
  - Baudrate
  - Masks and filters.
- Message processing
  - Verify which messages is to be sent over UART
- UART configuration
  - Baudrate
  - String of characters of the control message

The configuration for CAN communication, as most of the configuration in any Arduino sketch, is included at the beginning of the sketch. Due to this, the code does not create a function to perform this configuration.

Message processing is included in the “main” section of the sketch, at the moment is principal task, after the processing we know which control message should be sent.

Finally, to send the control messages, tests were performed with a finite, pre-defined for loop to print (send) the characters over the UART. Since the length of the control messages is fixed and already known, functions can be created and called and send each message. In this stage of the project we are using two pre-defined messages, start and stop, as global variables. This string of

characters is declared as an array and the function call is in charge of sending each of the elements of the array.

Figure 19 Bluetooth communication for control

Prior to the final application code integration, two different scripts for CAN Blue II were written, each of them in charge of triggering a different serial command. The scripts (batch files) that can be executed by the computer to send a CAN message over Bluetooth to the CAN Blue II.

The messages will be received and processed by the Arduino (with the CAN shield) and the Arduino is in charge of sending the corresponding control commands.

```
d version
#delay 0.5

#print
c filter clear STD

#print
c filter_clear EXT

#print
c filter disable STD

#print
c filter_disable EXT

#print
c can_init 500

#print
c can start

#label L1
m sd8 123 1 2 3 4 5 66 77 88
```

```
d version
#delay 0.5

#print
c filter clear STD

#print
c filter_clear EXT

#print
c filter disable STD

#print
c filter_disable EXT

#print
c can_init 500

#print
c can start

#label L1
m sd8 111 0 2 3 4 5 66 77 88
```

Code 8 Start and Stop batch files for CAN Blue II

The batch files shown above differ in just one aspect, the ID of the CAN message to be sent. The first approach to send different serial commands considered checking the value of the ID. With different IDs it is possible to send different serial commands.

The corresponding code for the final application is shown:

```
// Final sketch


// including configuration for the CAN communication
// and the corresponding action regarding the ID
// of the incomming message in order to control
// a relee in an external board.

#include <mcp_can.h>
#include <SPI.h>

long unsigned int rxId;
long unsigned int txId = 0x7FF;
unsigned char len = 0;
unsigned char rxBuf[8];
unsigned char txBuf[8] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0x0F, 0x0F};
unsigned char start_cmd[10]= {0x0D, 0x01, 0x53, 0x31, 0x6E, 0x0D, 0x01, 0x53, 0x31,
0x6E};
unsigned char stop_cmd[10] = {0x0D, 0x01, 0x43, 0x31, 0x7E, 0x0D, 0x01, 0x43, 0x31,
0x7E};


void setup()
{
  //Serial configuration for relee control
  Serial.begin(2400);

  //Serial configuration for computer communication (debugging)
  //Serial.begin(9600);

  //init can bus : baudrate = 500k
  CAN.begin(CAN_500KBPS);

  //Configure the CAN controller
  //configure mask and filters for the messages
  //we receive only std id with id 123 or 111
  CAN.init_Mask(0, 0, 0x07FF);

  //Buffer 0
  CAN.init_Filt(0, 0, 0x0123);
  CAN.init_Filt(1, 0, 0x0111);

  //Buffer 1
  CAN.init_Mask(1, 0, 0x07FF);
  CAN.init_Filt(2, 0, 0x0000);
  CAN.init_Filt(3, 0, 0x0000);
  CAN.init_Filt(4, 0, 0x0000);
  CAN.init_Filt(5, 0, 0x0000);
}

void printCANmsg_serial(){
  Serial.println("Received ID: ");
  Serial.print(rxId, HEX);
  Serial.print("  Data: ");
  //print each byte of data
  for(int i = 0; i<len; i++){
    //add a 0 for 0x00 representation when data is less than 0x10
    if(rxBuf[i] < 0x10){
      Serial.print("0");
    }
    Serial.print(rxBuf[i], HEX);
    Serial.print(" ");
  }
}
```

```
void sendSerialCmd(unsigned char cmd[]){
  for(int i = 0; i<10; i++){
    Serial.write(cmd[i]);
  }
}

//Main execution program
void loop()
{
  //checkReceive fcn, return 3 when message is availalbe
  //returns 4 when no message arrived (check library code)
  if(CAN.checkReceive()==3){
    //Read data: len = data length, buf = data byte(s)
    CAN.readMsgBuf(&len, rxBuf);
    //Get message ID
    rxId = CAN.getCanId();
    //Serial print to computer for debugging
    //printCANmsg_serial();


    /*
    //Control with ID check
    if(rxId == 0x123){
      sendSerialCmd(start_cmd);
    }
    else if (rxId = 111){
      sendSerialCmd(stop cmd);
    }
    */

    //Control with first bit of Data field check
    if(rxBuf[0] == 0x01){
      sendSerialCmd(start_cmd);
    }
    else if(rxBuf[0] == 0x00){
      sendSerialCmd(stop_cmd);
    }

    //echo the received message to the BT device
    // send data:  id = rxID, standrad frame, data len = 8, stmp: data buffer (received
data)
    CAN.sendMsgBuf(rxId, 0, 8, rxBuf);
  }

}
```

Code 9 Final Arduino Sketch with user declared functions

Common learnt lessons

After working in this project is important to remark some situations that are common during the assimilation of the technology and the given solutions.

SPI Communication

The first situation regarding assimilation is the communication between the Arduino and the shield to perform the configuration of the CAN controller. Without the correct communication between the µC (Arduino) and the CAN controller nothing will be possible. Regarding this physical connection between the CAN shield (and generally any shield for Arduino), the shield design is done such that the pins are directly connected to the Arduino board. This is done in order to make the shields compatible with different Arduino Boards.

The user can assume that the SPI pins (used for the communication between Arduino and the CAN shield) are directly mapped from the headers to the corresponding pins of the integrated circuit; the SPI pins are indicated in the Arduino reference to be pins number 10 to 13. However the correct communication is done, due to the use of third-party libraries, through the alternative SPI connection pins, located in the ICSP pins of the Arduino board. In the case, the user decides to modify the library, taking into account the corresponding property rights, the communication can be establish through the desired pins, either ICSP or 10 to 13 of the Arduino UNO.

The decision of which pins to be used for SPI communication depends on the application to be developed, for instance, the situation where the pins 10 to 13 are used for SPI in an Arduino UNO will demand four usable pins. In the case where the six available Arduino UNO pwm outputs are needed, the use of the ICSP pins will be better in order to leave the pwm pins available.

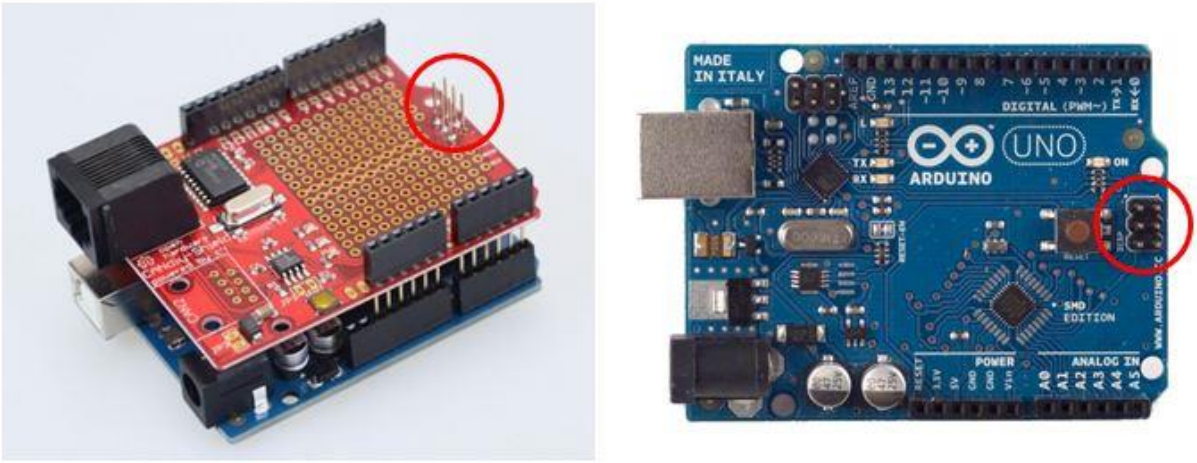


Figure 20 ICSP pins in Arduino Board and CAN Shield

Third-party library

In an Arduino-based system is, as it is Open-source hardware and software, it is possible to find different user developed solutions for the same application; most of them are available on-line and under the GNU license in order to be modified and contribute for further development.

That is the case with the CAN library used in this project. During the assimilation phase, it was possible to find and test three different third-party libraries.

The first proposed solution was the library suggested by the manufacturer of the used shield. The library, under the folder MCP2515, can be found in:

http://github.com/watterott/Arduino-Libs

The second tested library was developed by Cory J Fowler, the main change in this library is the possible inclusion of more than one CAN shields, object oriented in order to perform communication between them. Available at:

http://github.com/coryjfowler/MCP2515_lib

The third, and selected library due to the inclusion of useful functions for this project, is the library from Seed Studio. Mostly is the same as the suggested by the manufacturer, it includes functionality to perform the configuration of the CAN controller registers and functions to receive and transmit messages. One useful function is the possibility to read the ID of the CAN message directly with a function. The library, and some documentation, can be found in:

http://github.com/Seeed-Studio/CAN_BUS_Shield

Data over serial UART

The option to include some other communication due to the use of a micro controller represents scalability; however it is important to understand how data is transferred. The serial communication after the CAN message processing can be implemented using the Arduino-built serial library. In this library the user will find two different options to send the message, the functions with print (print and println) and the function with write.

The functions print send the ASCII value of the parameter, the function writes sends the series of bytes. In our application, it is necessary to send bytes so we use the function write instead of the commonly use print.

Further development work

As in any project, there is always the possibility to enhance the current functionality. The suggested improvements are software related, this first approach was under the constraint of technology assimilation so the used libraries remain as they were found, however improvements in the library with user defined functions is possible.

Another improvement could be the interrupt routine to process the received message. At the moment the Arduino sketch is checking the corresponding register every loop (polling); in the case there is something, the code proceeds to the following sections.

The first situation regarding the interruption is, there should exist a physical connection between the interrupt pin of the CAN controller and the pin where the interruption will be detected in the Arduino. After the physical connection is established, the corresponding configuration should be done, via the registers in the CAN controller and via the attatchInterrupt() function.

Depending on the scenario or place where the system is used, there is also the possibility to focus on energy consumption, normally the main objective when the application is battery powered. For this, there are sleep modes available, either for the CAN controller and for the Arduino. This also requires the review part of interruptions and configuration, the use of interruptions is needed due to the operation mode change in both devices, from sleep (low power) to normal and vice versa.

Apendix

Datasheet

ATMEGA328 Datasheet http://www.atmel.com/Images/doc8161.pdf

MCP2515 (CAN Controller) Datasheet

http://ww1.microchip.com/downloads/en/DeviceDoc/21801G.pdf

MCP2551 (CAN Transceiver) Datasheet

http://ww1.microchip.com/downloads/en/DeviceDoc/21667f.pdf

CAN Blue II manual

http://www.ixxat.com/download/manuals/canblue-ii_manual.pdf

Libraries

The library used for the code was made by Seed Studio. They are responsible for the development of another CAN-Shield for Arduino (not the one used here); however the CAN controller is the same (MCP 2515) and the functions are fully compatible.

Compared to the library offered by the manufacturer of the used shield (Watterott), with the Seed Studio library we can use more pre-coded functions

The library repository is available at:

http://github.com/Seeed-Studio/CAN_BUS_Shield

Diagrams

The flow diagrams (taken from the CAN controller datasheet) corresponding to send and receive tasks are shown

Diagram 1 Message transmission diagram

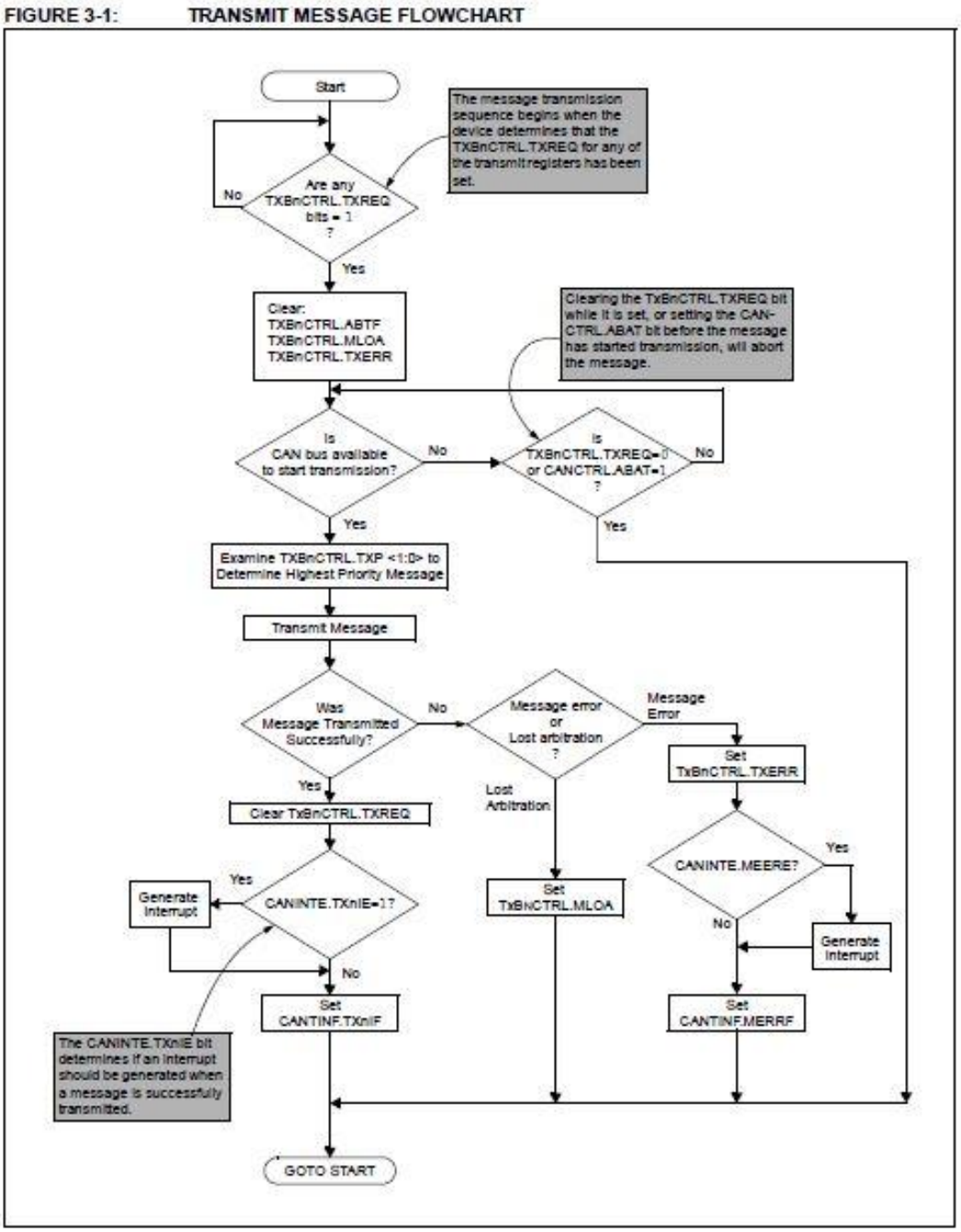
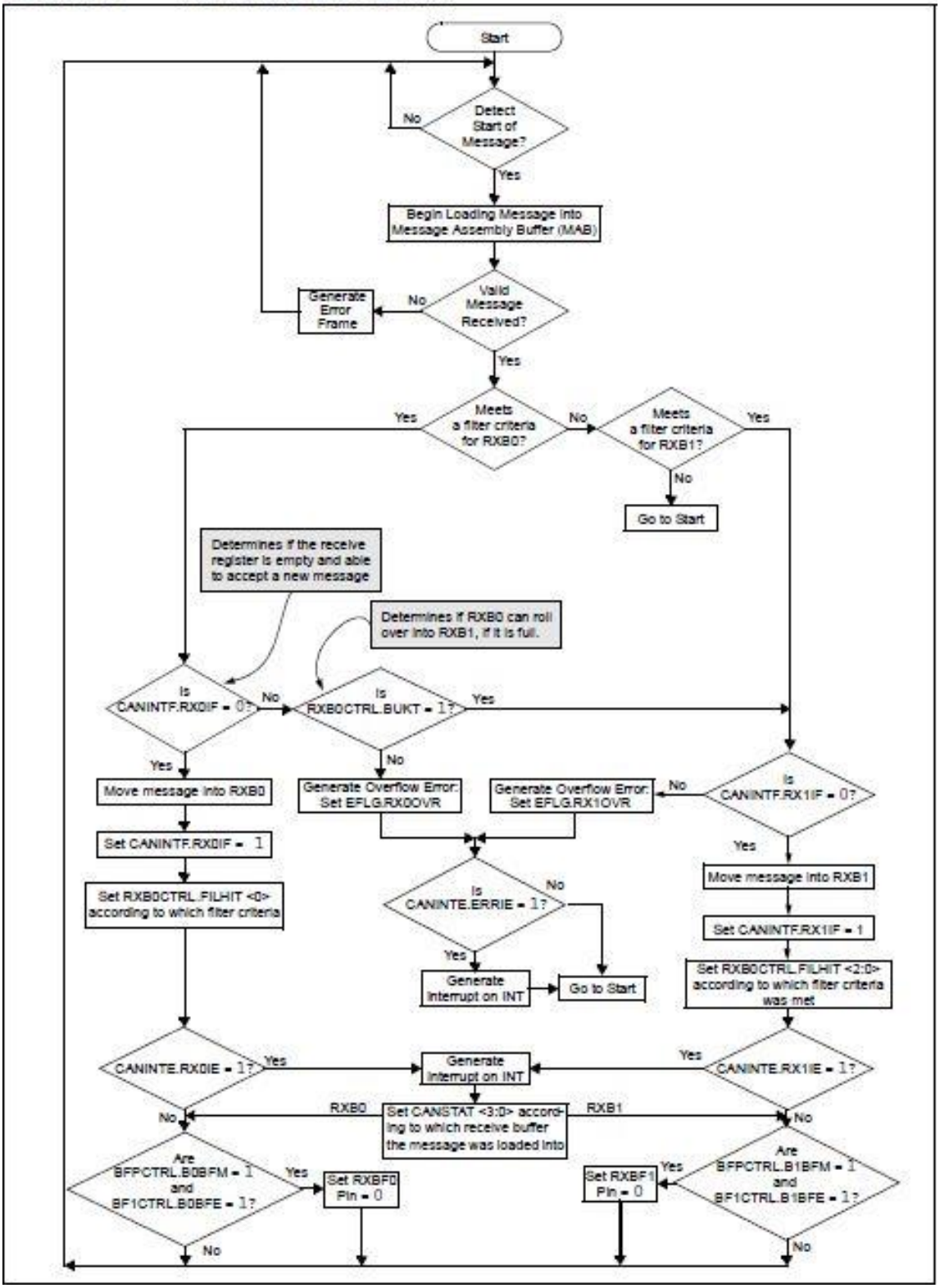


Diagram 2 Message reception diagram

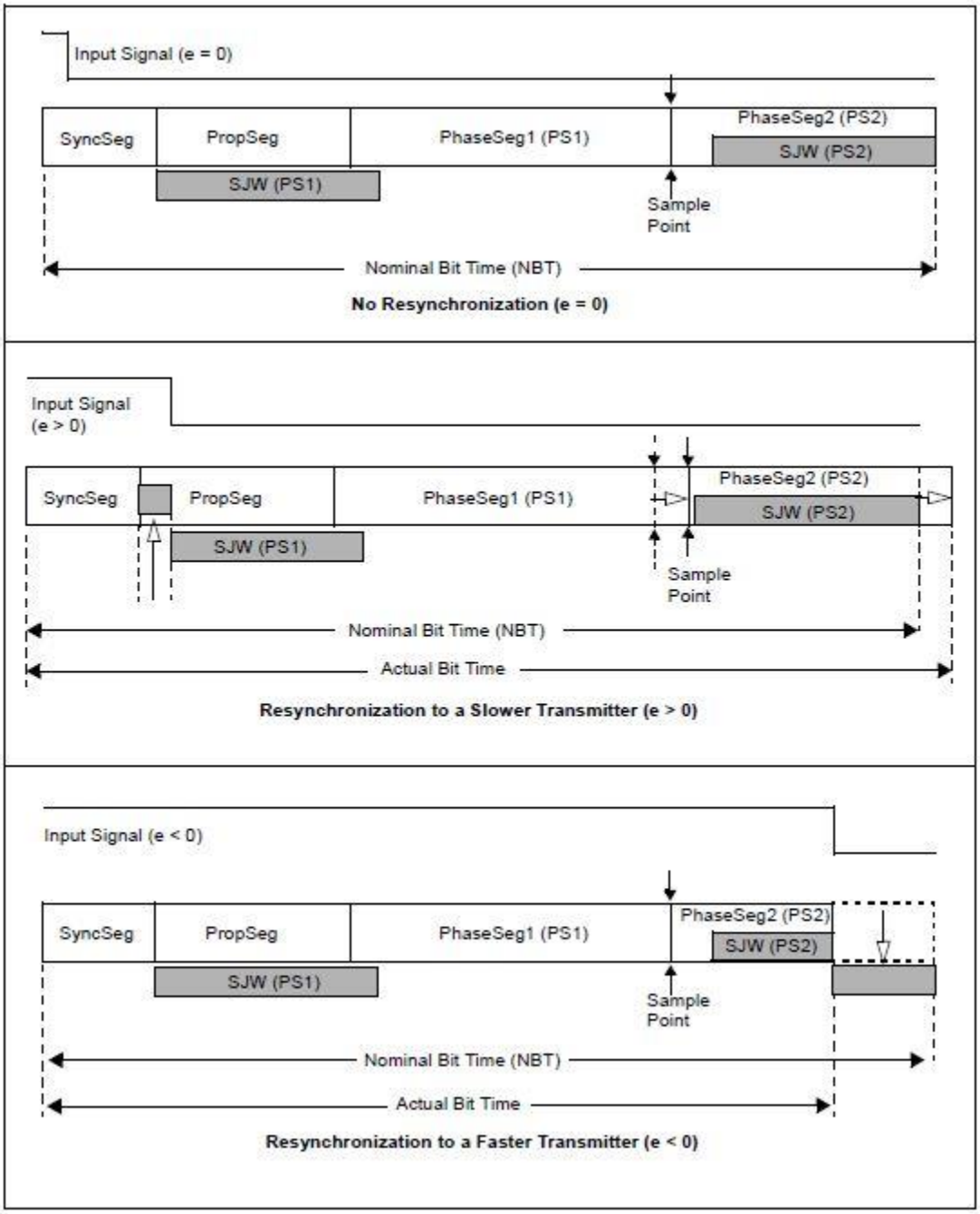
FIGURE 4-3: RECEIVE FLOW FLOWCHART



Synchronization

Synchronization cases for the CAN Controller

Diagram 3 Synchronization cases diagram



Code snippets

Figure Index

Diagram Index

CAN Bus communication with Arduino

Hochschule Esslingen Summer Semester 2014

M.Eng. Automotive Systems. Software-Based Automotive Systems

Julio César Rodríguez Mejía

Proyect supervisor: Vikas Agrawal