Hochschule Esslingen

CAN Bus communication with Arduino

Using CANdiy shield

Julio César Rodríguez Mejía
01/05/2014

Index

Objective

Develop a prototype for communication between Arduino and a CAN Bus

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.

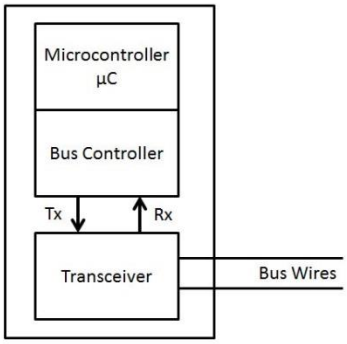


Figure 1 Network components

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 it 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 thus, whenever an ATMEGA328 microcontroller is intended to be used as a standalone microcontroller application developed with Arduino, it is necessary to build the minimum component circuit and load the bootloader in it prior to upload the sketch.

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

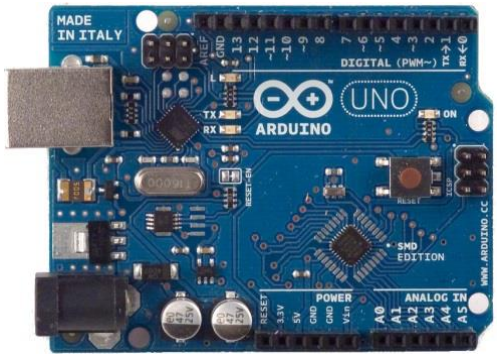


Figure 2 Arduino UNO board

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 CAN messages from a USB device, normally 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

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:

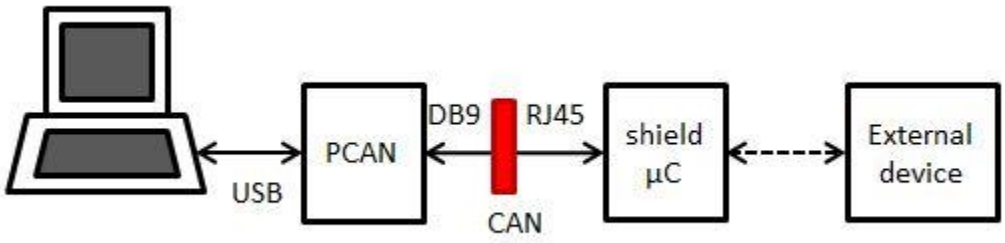
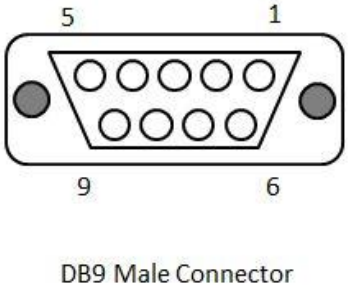


Figure 5 Block diagram, communication

- 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.

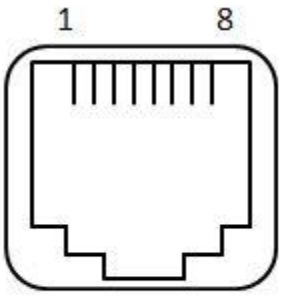
The CAN Bus is compound by two wires, the CAN_H and CAN_L, it can also be implemented by a one-wire Bus for low speed bit rates. The wires arrange in the sockets we will use (DB9 and RJ45) are shown in the following figure.



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 6 DB9 CAN pin arrangement



RJ45 Connector

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 7 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.

Compared 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

Within the inclusion of a microcontroller in our application, it is always necessary to start with the configuration mode; in this case we can “tune” the CAN controller.

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 buffers.

In order to forward the message from the MAB to the other 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 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 bits 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 thus 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.

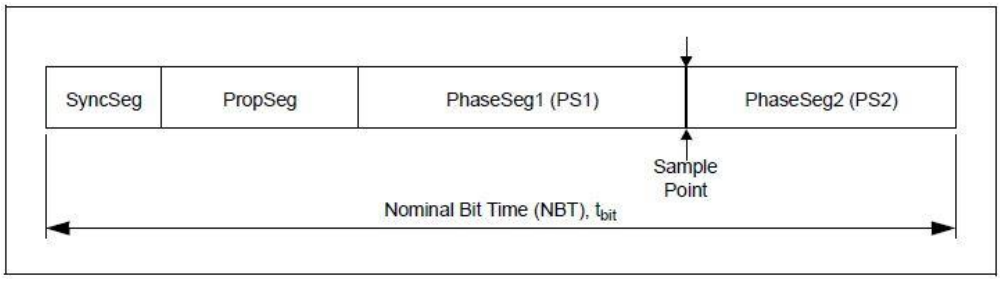


Figure 8 CAN Bit Segments

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. PS1 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 2TQ resulting in a minimum PS2 of 2TQ, 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

e < 0; edge after Sample Point; PS2 shortened

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. To perform different configuration actions and message processing it is possible to use 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-parity library helps us saving time in code writing; they include de values of the registers 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.

Test

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

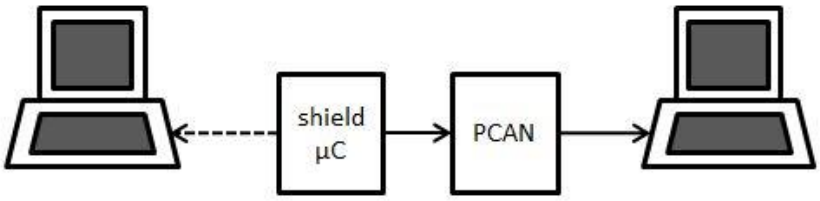


Figure 9 Connection for send test

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

The connection between the shield/µC 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 test.

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

Libraries

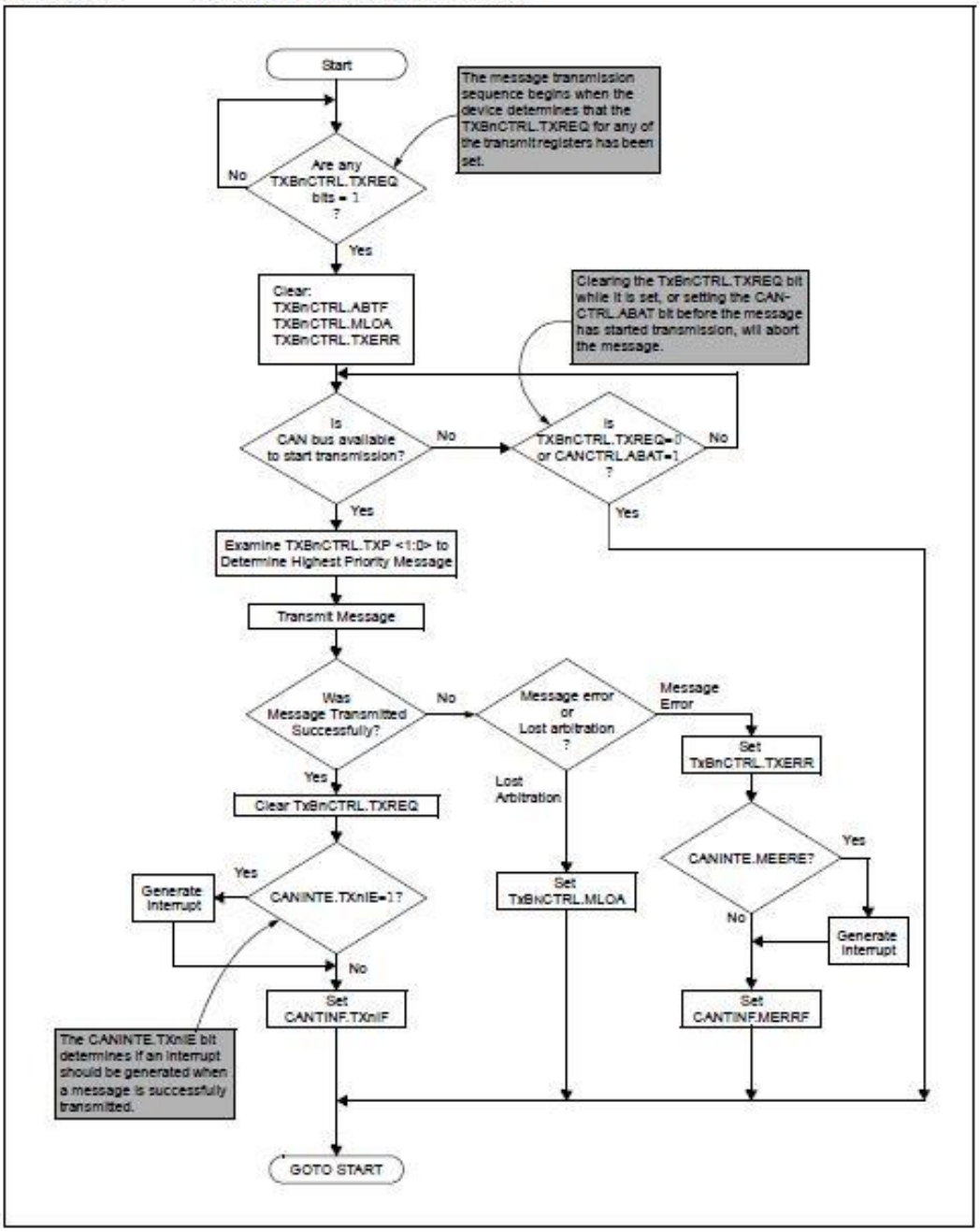
The libraries used for the code development are available at

Diagrams

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

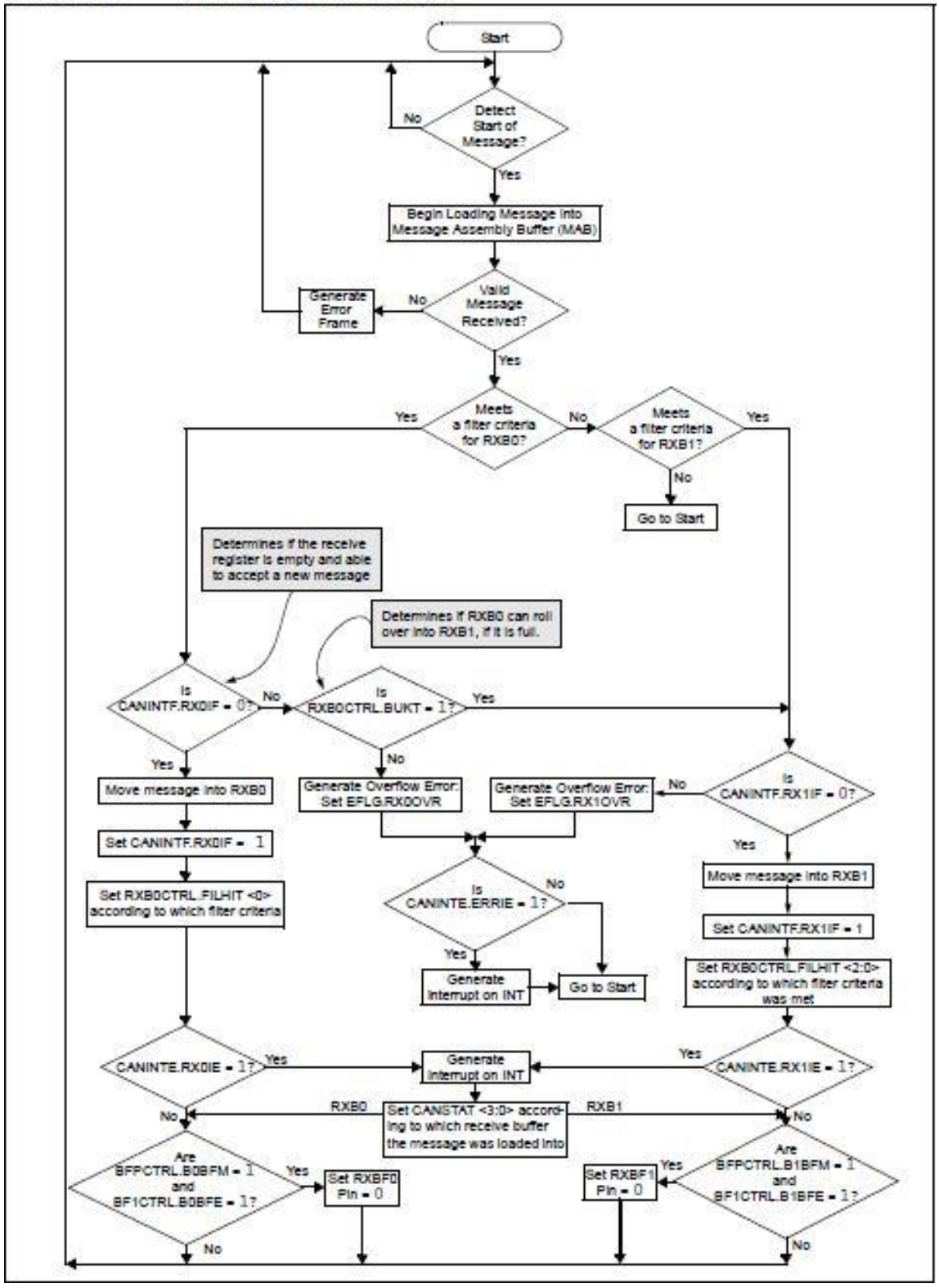
Message transmission

FIGURE 3-1: TRANSMIT MESSAGE FLOWCHART



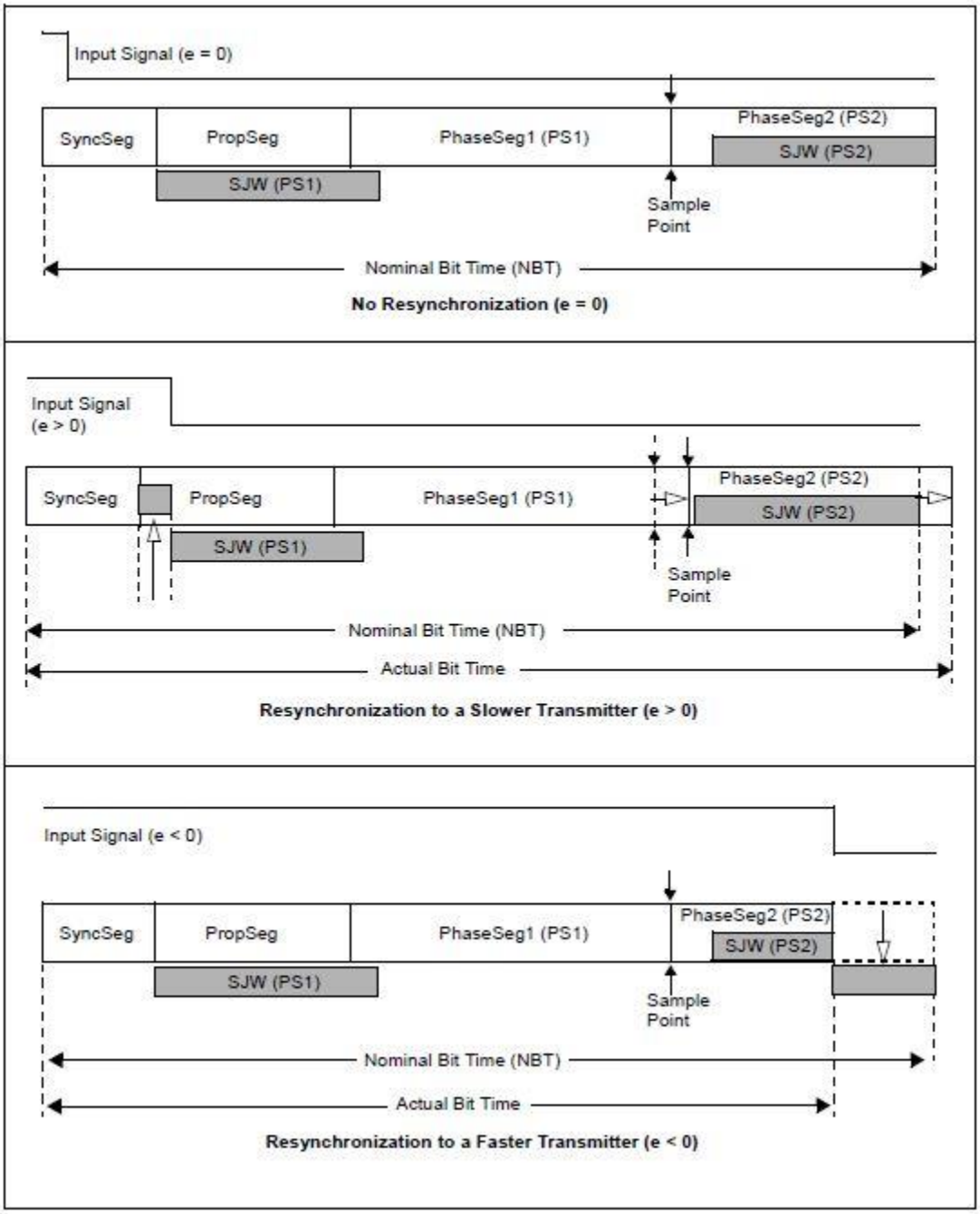
Message reception

FIGURE 4-3: RECEIVE FLOW FLOWCHART



Synchronization

Synchronization cases for Bit-Synchronization in CAN



Taken from the MSP 2515 Datasheet.