ESIEE PARIS

NETWORKING

CAN & MCP2515

15.1.2012

TEACHER

BUSSIGNIES G.

TEAM

BLAHA Stepan
KUBICKA Matej

Introduction

Aim of our project is to study datasheets and connect 8-bit AVR microcontrollers to the CAN bus. We have chosen this project, because MCP2515 integrated circuit interests us for longer time - it enables low price microcontrollers to be connected to the CAN bus.

Output of our project should be an example source code showing how to setup the device, send and receive messages. We are using C language with GCC compiler with standard library, no specialized third-party libraries.

Theoretical background

CAN (Control Area Network) is bus, which was designed and standardized for communication between two or more devices without a host computer. Typically this bus is used for communication between electronic units (microcontrollers) inside a vehicle. Bus can connect for example an engine control unit and transmission unit or other units inside a vehicle like the door locks, airbags control etc. Automotive usage of CAN bus is one way. Today it’s also used as a field bus in general automation environments (in industrial automation and medical equipments). One of the reasons for this usage is because of the low cost of CAN bus components like controllers and processors.
The first version of CAN bus was developed during 1980’s as well as CAN protocol. Whole specification of this bus was specified at begin of 1990’s.

CAN is a multi-master serial bus standardized for connection of control units. Control units that are typically connected by a CAN network are for example sensors and actuators. Usually these devices are connected to the bus through a microprocessor and CAN controller. This connection allows getting a data from unit (sensors/actuators), processing them by processor and controller and send them to another unit over bus.

Transmission

What means multi-master serial bus? It means that each unit can transmit or receive a messages over the CAN bus, but it’s with some conditions. Mainly it depends on message’s ID and its priority. If bus is free and two or more units begin sending messages at same time, the message with the highest priority will overwrite other messages with lower priority. It means that only this highest priority message remains and is received to all units connected to the bus. Other messages have to wait until the bus is free again.
As it’s shown in frame description, the priority of each message is specified in part of frame called identifier. This identifier has got an 11 or 29 bits (it depends on format of frame). In identifier zero values are dominant and ones are recessive. It means that identifier with higher number of zeros (smaller value) gives higher priority to given frame message.

Frame Format

Messages between units on the bus are distributed by frames. A CAN network can be configured to operate with different type of frames. First important type of frame is standard data frame. The second important type is extended data frame. As you can see from the diagrams, the only difference between these two types is that standard frame supports 11 bits for identifier and extended frame supports 29 bits for identifier. This identifier is made of 11 bits in base and 18 bits in extension. The distraction between the two types of frames is made by using IDE bit. This bit is set as dominant (zero) in case of 11 bits frame identifier and in case of 29 bits frame identifier the bit is set as recessive (one). If CAN controller supports extended frame format of message, it’s also able to work with standard frame format. It’s back compatible.

Boundaries of each frame are created by bit Start-of-frame (SOF) at begin and bits End-of-frame (EOF) at the end. Parts EOF is group of 7 bits with value one. After starting bit (SOF) frame continue with specific identifier. This identifier is followed by IDE bit, RTR bit (remote transmission request) and reserved bit. These three bits are followed by DLC section (data length code), which consist of 4 bits. DLC section selects the number of data bytes (from 0 to 8 bytes). The most important part of frame is data field and it’s after DLC section. Data field could be from 0 to 64 bits. After this field there is a CRC section (cyclic redundancy check). It’s an error-detecting code, which allows protecting a data field. CRC section consists of 15 bits. Before EOF section (end of frame) there are bits like CRC delimiter, acknowledgement bits.

Bit Timing

All units on the given CAN bus must use the same bit rate. But all units are not required to have the same master oscillator clock frequency. The CAN protocol uses Non Return to Zero (NRZ) coding. This type of coding does not encode a clock within the data stream. Therefore, the receive clock must be recovered by the receiving data and must be synchronized to the transmitter’s clock. For this case receiver have some type of Phase Lock Loop (PLL) to synchronized receiver clock. For right synchronization of edges is used bit stuffing. For different clock frequency of each unit, the bit rate has to be adjusted by appropriately setting a baud rate prescaler and number of time quanta in each segment. Each segment (the CAN bit time) is made of 4 non-overlapping segments (synchronization, propagation and phase segments used for compensation of edge phase errors). Each of these non-overlapping segments is made by multiple of time quantum. Time quantum (TQ) is fixed to frequency of oscillator.
Synchronization segment from the CAN bit time is used for synchronization on the bus. Bit edges are expected to occur in this segment, which is fixed as one times TQ. The propagation segment is used for compensation of physical delays between nodes. This segment has variable length from 1 to 8 times TQ. Last two segments (phase 1 and 2) are used to compensate edge phase errors on the bus. They have also variable length from 1 or 2 (it depends on phase) to 8 times TQ. Synchronization is the most important part because continuous synchronization enables the receiver to read the messages properly.

Layers

The protocol of CAN could be decomposed to the specific layers when each layer describes different part of this protocol. The layers covered by CAN protocol are three (object, transfer and physical layer). Higher layers like application layer are covered by other high-level protocols. At the bottom layers structure there is a physical layer. The CAN protocol specified only abstract requirements for this layer like electrical aspects (voltage, current, number of conductors). However, other requirements like mechanical aspects (type of connector etc.) have yet to be specified. Other layer, which is above physical, is transfer layer. Most of the CAN protocol applies this layer. Transfer layer receive a messages from layer above and transmits those messages to another. Transfer layer is responsible for many processes like bit timing, synchronization, error detection, message framing etc. The highest layer included in the CAN protocol is object layer. In this part are processed message and status handling and also message filtering by given mask.

MCP2515 – CAN controller

This integrated circuit is stand-alone controller Microchip Technology’s device that implements the CAN specification in version 2.0B. It means that the controller is capable of transmitting and receiving both standard and extended data frames. The CAN specification in version 2.0B is able to communicate up to 1Mb/s (it depends on the length of bus). Controller MCP2515 is able to communicate with microprocessor via an industry standard Serial Peripheral Interface (SPI). This interface is high-speed (10MHz). It has also an extra interrupt output pin for calling interrupts of microprocessor. The MCP2515 has two acceptance masks (each of them has 29 bits) and six acceptance filters (each of them has 29 bits) that are used to filter out unwanted messages, thereby reducing the microprocessor’s overhead.
The MCP2515 controller consists of three main parts. Each part covers different functions of this CAN controller.

CAN Module

First part is CAN Module, which includes the CAN protocol engine, masks, filters, buffers for transmission and receiving.
The CAN module handles all functions for receiving and transmitting messages on the CAN bus. Messages are transmitted by first loading the appropriate message buffer and control registers. Transmission is initiated by using control register bits via the SPI interface or by using the transmit enable pins. Status and errors can be checked by reading the appropriate registers. Any message detected on the CAN bus is checked for errors and then matched against the user-defined filters to see if it should be moved into one of the two receives buffers.

Control Logic

Second part of the CAN controller is control logic, which is used to configure the device and its operation by interfacing to the other blocks in order to pass information and control.
Interrupt pins are provided to allow greater system flexibility. There is one multi-purpose interrupt pin for each of the receive registers that can be used to indicate a valid message has been received and loaded into one of the receive buffers. Use of the specific interrupts pin is optional. The general purpose interrupts pin, as well as status registers (accessible by the SPI interface), can also be used to determine when a valid message has been received.
Additionally, there are three pins available to initiate immediate transmission of a message that has been loaded into one of the three transmit registers. Use of these pins is optional, as initiating message transmissions can also be accomplished by utilizing control registers, accessed via the SPI interface.

SPI Protocol Block

Third part of controller is SPI Protocol Block. The processor interfaces to the device via the SPI interface and this block manages the communication. Writing to (and reading from) all registers is accomplished using standard SPI read and write commands, in addition to specialized SPI commands.

Testing suite

To actually test our project we have setup a solution on a prototype board with two MCU's, and a CAN network made of two nodes connected through CAN transceiver MCP2551 and CAN controller MCP2515. MCP2515 is connected through SPI bus and INT pin to AVR microcontroller. We have used microcontrollers ATmega8 and Atmega16, because they were currently available for us. They are a not the same, but belong to the same MCU family and have the same SPI peripheral. Following photograph shows the prototype board with our circuit. Please note that both SPI buses and interruption pins are not connected at this stage.

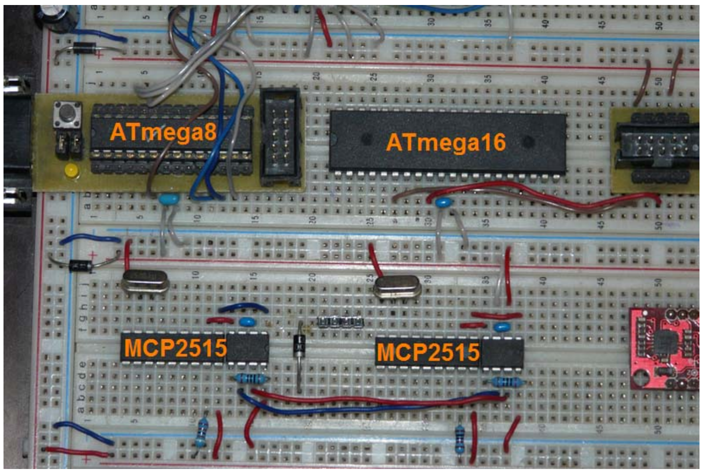

Figure 1: Prototype board solution used for testing purposes.

To connect MCP2515 to the MCU we need to connect SPI bus (MISO, MOSI, CLK, CS signals) and INT signal. INT signal is interruption generated by the MCP2515 when a new message is received.

SPI interface control

Lowest level of our solution is a little library made to communicate with the device. It consists from functions which allow access to SPI peripheral, sending and receiving octets. Additionally we have implemented all the instruction primitives supported by the MCP2515. Following instructions are supported:

- **Reset instruction** Used for software reset of the device
- **Read instruction** Provides read access to the register map
- **Read RX buffer instruction** Allows quick access to selected receive buffer
- **Write instruction** Provides write access to the register map
- **Load TX buffer instruction** Allows quick access to selected transmit buffer
- **Request-to-send (RTS)** Requests to send CAN message in selected buffer
- **Read device status instruction** Returns information about the device and its buffers.
- **Read RX status instruction** Returns information about received messages
- **Bit modify instruction** Allows changing masked bits in selected register

We have created a file MCP2515.h, which describes all the device properties and performances. For the file, see enclosed Appendix A.

Initialization of the SPI peripheral is done with function spiMasterINIT(). We have set SPI mode 0,0 (data are received and transmitted at the same moment on rising edge of the CLK) and CLK frequency to 4Mhz. Access to the SPI is done with spiMasterTRANSMIT() function which takes data to be sent as an argument and returns what was received during the sending process. SPI is full duplex bus controlled by the master device (the MCU in our case). Please note that we use this function for both transmitting and receiving data over SPI bus. Last function is spiMasterChipSelect() which is used to signalize start/end of the communication.

Following listing shows hardware access layer made of functions described above.

```
#ifdef __AVR

#define DDR_SPI   DDRB        /* Data dir. register for port with SPI */
#define PORT_SPI  PORTB       /* Port with SPI */
#define PIN_MOSI  PB3         /* MOSI pin on the PORTB_SPI */
#define PIN_MISO  PB4         /* MISO pin on the PORTB_SPI */
#define PIN_SCK   PB5         /* SCK pin on the PORTB_SPI */
#define PIN_SS    PB2         /* SS pin on the PORTB_SPI */


#include "avr/io.h"
#include "avr/interrupt.h"

/** \brief Initialization of the SPI interface on the MCU
 *
 * Initialization of the SPI hardware interface - configure this
 * device as master, set mode 0,0 and the communication speed (there
 * is limitation - 10Mhz, nominal speed should be >8Mhz, for this
 * purpose.
 *
 * \warning This is platform-dependent method!
 *
 */
void spiMasterINIT()
{
      /* Set MOSI and SCK output, all others input */
      DDR_SPI = (1<<PIN_MOSI)|(1<<PIN_SCK)|(1<<PIN_SS);
      PORT_SPI |= (1 << PIN_SS);

      /* Enable SPI, Master, set clock rate fck/4, mode 0,0 */
      SPCR = (1<<SPE) | (1<<MSTR);
      SPSR = (1<<SPI2X);

}
/** \brief Transmiting databytes via the SPI
 *
 * This function is transmitting data via the SPI interface. Input
 * parameter is uns. char array. Data are transmited from the zero
 * index
 *
 * \warning This is platform-dependent method!
 * \param data[] Source data array
 * \param length Array length
 *
 */
```

```
unsigned char spiMasterTRANSMIT(unsigned char data)
{
      /* Start transmission */
      SPDR = data;
      /* Wait for transmission complete */
      while(!(SPSR & (1<<SPIF)));
      /* SPDR must be stored as quickly
         as possible (ref. ATMegaX ds) */
      return SPDR;
}
/** \brief Settings of the CS pin
 *
 * This function is used for setting of the CS pin. CS signal
 * is inverted, so input 1 (true) means zero on the output.
 *  Otherwise is analogically the same.
 *
 * \warning This is platform-dependent method!
 * \param state Wished state
 */
void spiMasterChipSelect(unsigned char state)
{
      /* What the user wants? (remember that the CS signal is inverted) */
      if(!state) {
            /* Upper the CS pin */
            PORT_SPI |= (1<<PIN_SS);
            DDR_SPI |= (1<<PIN_SS);
      } else {
            /* Lower the CS pin */
            PORT_SPI &= ~(1<<PIN_SS);
            DDR_SPI |= (1<<PIN_SS);
      }
}
```

External interrupts are used to detect change on an INT pin which is used by MCP2515 to signal change of state (error, received message and so on). This pin is required, because SPI is master-slave bus, where MCP2515 poses as slave and do not have the right to start the communication. INT pin is used to inform MCU about occurred interruption.

```
/* Pointer to function which handle change on INT pin handler */
void (*int_handler)(void);

/** Initialization of hardware ext. interrupts
  * \param *handler pointer to a function which handle occured interrupt.
  * \return nothing
  */
void extInterruptINIT(void (*handler)(void))
{
      /* Set function pointer */
      int_handler = handler;

      /* Initialize external interrupt on pin INT0 on failing edge */
      MCUCR |= (1 << ISC01);
      GICR |= (1 << INT0);
}
/* System interrupt handler */
SIGNAL(INT0_vect)
{
      int_handler();
}
#endif
```

Hardware access layer consisting of three functions above is used for implementation of all primitive instructions. We basically need only Read register, Write register and Reset instructions, but there are actually 9 types which allow communicating with the device more effectively. Listing below shows implementation of Read, Write and Reset instructions.

```
/**
 * Read value of the register on selected address inside the
 * MCP2515. Works for every register.
 *
 * \see MCP2515 datasheet, chapter 11 - register description
 * \see MCP2515 datasheet, chapter 12 - read instruction
 * \param address Register address
 */
unsigned char readRegister(unsigned char address)
{
      /* Send read instruction, address, and receive result */
      spiMasterChipSelect(1);
      spiMasterTRANSMIT(READ_INSTRUCTION);
      spiMasterTRANSMIT(address);
      unsigned char buffer = spiMasterTRANSMIT(0);
      spiMasterChipSelect(0);

      return buffer;
}

/**
 * Change value of the register on selected address inside the
 * MCP2515. Works for every register.
 *
 * \see MCP2515 datasheet, chapter 11 - register description
 * \see MCP2515 datasheet, chapter 12 - write instruction
 * \param address Register address
 * \param value New value of the register
 */
void writeRegister(unsigned char address, unsigned char value)
{
      /* Send write instruction, address, and data */
      spiMasterChipSelect(1);
      spiMasterTRANSMIT(WRITE_INSTRUCTION);
      spiMasterTRANSMIT(address);
      spiMasterTRANSMIT(value);
      spiMasterChipSelect(0);
}

/**
 * Send reset instruction to the MCP2515. Device should
 * reinitialize yourself and go to the configuration mode
 */
void resetMCP2515()
{

      /* Send reset instruction */
      spiMasterChipSelect(1);
      spiMasterTRANSMIT(RESET_INSTRUCTION);
      spiMasterChipSelect(0);
}
```

We have precompiled functions implementing all instructions separately (thus, got object file for every each instruction) and packed it to a single *.a file (standard library of gcc is precompiled in this way). This library does not include hardware access layer and thus is completely device independent – it works on different microcontrollers from AVR8 family.
On the other hand, library functions expect the hardware access layer implemented outside and specific to used microcontroller – the code in the library is not linked. There are "unconnected" calls for functions spiMasterINIT(), spiMasterTRANSMIT() and spiMasterChipSelect(). The linker will try to link calls from library to those functions – user has to implement them.

Device settings

The device gets after startup automatically to the configuration mode. In this mode there are configuration registers available for writing. We have to setup bit timing, message receiving policy (receive masks and filters) and wanted mode (listen only mode, loopback mode, standard mode). To get and set actual device mode, we have prepared macros getMode() and setMode().

```
#define getMode       (readRegister(CANSTAT) >> 5);
#define setMode(mode) { changeBits(CANCTRL, (7 << REQOP0), \
      (mode << REQOP0)); while(getMode != mode); }
```

Bit timing configuration

As was described in the introduction part we have to setup bit timing. To do that, we have to set the three bit timing registers – CNF1, CNF2 and CNF3 (MCP2515 datasheet, page 42). For this purposes we have prepared function setBitTiming().

```
unsigned char setBitTiming(unsigned char rCNF1,
                           unsigned char rCNF2,
                           unsigned char rCNF3)
{
      if(getMode == CONFIGURATION_MODE) {
            writeRegister(CNF1, rCNF1);
            writeRegister(CNF2, rCNF2);
            changeBits(CNF3, 0x07, rCNF3);
            return 1;
      }
      return 0;
}
```

Wanted communication speed is 250 kb/s, we are using 25 MHz crystal as input clock. Also, without prescaler the MCP2515 divides input clock for bit timing by 2, so real input clock is 12.5 MHz.

To get transmission speed of 250kb/s, we need exactly 50 time quanta (50 * 500 kHz = 12.5 MHz). 50 time quanta as nominal bit time is too large number, so we set input clock prescaler to divide by 5 and get 2.5 MHz input clock, where 1 time quanta takes 400ns and nominal bit time is 10 time quanta. It means that we have to set sizes of all segments in a way that their total size equals to ten. There are 4 segments – synchronization segment, propagation segment, phase segment 1 and phase segment 2. Synchronization segment takes exactly 1 time quanta, which leaves for us 9 time quanta to set up. The measurement is done between phase segment 1 and phase segment 2 and it should be done around 2/3 of nominal bit time, hence after six or seven time quanta from the beginning.

We decided to setup our bit timing in a way that 3 time quanta are taken in propagation segment, 3 time quanta are taken by phase segment 1, then the measurement is done and last 3 time quanta are taken by phase segment 2 to finish nominal bit time. Additionally, MCP2515 allows setting some additional features – it can do three samples instead of one, which improves noise resistance. We also enabled wakeup filter, which works in a sleep mode and filters input data - the device is woken up when activity on the bus occurs, the filter increases noise resistance for unwanted device wakeup. According register values are CNF1=0x04, CNF2=0xD2, CNF3=0x42.

Receive masks and filters configuration

The device has two receive buffers, six 29-bit filters and two 29-bit masks. In general, masks are used to mask selected bits of the identifier which are compared with the filters. Filters are setting criteria for accepting/rejecting messages which effectively lowers device overhead by receiving only chosen messages. Normally filters decide where is the message stored according to acceptance criteria, but this device have special feature - a “rollover” mode, in which received message is always stored to the receive buffer 0, and in case it is full, to receive buffer 1. We have used this mode and that we want to receive all messages. To set rollover we defined following macro:

```
#define setRollover(v) changeBits(RXB0CTRL, 1 << BUKT, v << BUKT);
```

To setup n-th receive filter we have to set 4 registers – RXFnSIDH, RXFnSIDL, RXFnEID8 and RXFnEID0. RXFnSIDH, RXFnSIDL are used for standard identifiers, all four registers together are used for extended identifiers. To setup n-th receive mask, we have to set registers in the same manner as when configuring filters, only register names are bit different - RXMnSIDH, RXMnSIDL, RXMnEID8 and RXMnEID0.

In order to receive any message, standard or extended identifier, we have to setup receive filters and mask. For that purpose we created function setAcceptanceCriteria(). This function uses directly SPI hardware access layer and takes advantage of MCP2515 write register instruction properties – we can directly write all 4 registers inside one instruction. To simplify usage of this function we define two macros setMask() and setFilter() which need only mask/filter index instead of register address, which is required by setAcceptanceCriteria().

```
#define setMask(n, c, e) setAcceptanceCriteria(RXMnSIDH(n), c, e)
#define setFilter(n, c, e) setAcceptanceCriteria(RXFnSIDH(n), c, e)

/** Set up acceptance filters/masks
 * \param address starting address of 4 registers to setup. It can be mask
 * or filter, doesn't matter
 * \param criterion message identifier criterion to be set
 * \param is_ext 1 if message is extended, otherwise 0 */
void setAcceptanceCriteria (unsigned char address,
                            unsigned long criterion,
                            unsigned char is_ext)
{
      /* Initialize reading of the receive buffer */
      spiMasterChipSelect(1);
      /* Send header and address */
      spiMasterTRANSMIT(WRITE_INSTRUCTION);
      spiMasterTRANSMIT(addr);

      /* Setup standard or extended identifier */
      if(is_ext) {
        spiMasterTRANSMIT((unsigned char)(criterion>>3));
        spiMasterTRANSMIT((unsigned char)(criterion<<5)
             |(1<<EXIDE)|((unsigned char)(criterion>>27));
        spiMasterTRANSMIT((unsigned char)(criterion>>19));
        spiMasterTRANSMIT((unsigned char)(criterion>>11));
      } else {
        spiMasterTRANSMIT((unsigned char)(criterion >> 3));
        spiMasterTRANSMIT((unsigned char)(criterion << 5));
      }

      /* Release the bus */
      spiMasterChipSelect(0);
}
```

In order to pass any received message, we need to setup only masks and mask all bits from the filters. Then any received message passes acceptance criteria. To do that we need to set both mask to zeros (0x00000000).

Putting it all together – initialization routine

Full initialization routine is listed below. First of all we initialize hardware resources on the microcontroller to control the MCP2515, and then we commit reset of the device to make sure that is in startup state. Then we set the device to configuration mode, which allows us to write to configuration registers. Now, we can set bit timing, message acceptance policy and rollover mode. In the end we change device state to normal mode, which starts normal function.

```
/* Configuration routine */
void initMCP2515(void)
{
      /* Initialize SPI as a master device, on frequency < 10Mhz */
      spiMasterINIT();
      /* Initialize external interrupt service on this device */
      extInterruptINIT(interruptMCP2515);

      /* Send reset instruction */
      resetMCP2515();
      /* Set configuration mode */
      setMode(CONFIGURATION_MODE);

      /* Set bit timing , masks and rollover mode*/
      setBitTiming(0x04, 0xD2, 0x42);
      setMask(RXM0, 0x00000000, 1);
      setMask(RXM1, 0x00000000, 1);
      setRollover(1);

      /* Get into normal mode and setup communication */
      setMode(NORMAL_MODE)
}
```

Sending messages

There are three available transmit buffers. In simple means, to send a message we have to choose a buffer, then fill it with data and then make a request for the message to be sent. The data to be set consists from the message itself (up to 8 bytes), identifier (standard or extended) and configuration data (length of the message, its priority).

To set message priority and request message to be sent we can use TXBnCTRL register. Then we have to set message identifier. It consists from 4 register with same organisation as when setting acceptance criteria (TXBnSIDH, TXBnSIDL, TXBnEID8, TXBnEID0). After that there is a TXBnDLC register used to setup message length (0 to 8 bits) and to specify whether this message is a remote request (in that case message length is 0 bytes and the RTR bit in the message frame is set). After this there are 8 registers (TXBnD0 to TXBnD7) for the message content.

We can setup whole transmit buffer in single write instruction. Message is transmitted as soon as the bus is available after setting up RTS bit in TXBnCTRL register. Setting up RTS can be done by using write instruction for second time, or by special instruction “set RTS” which is faster because it removes overhead by 1 byte over write instruction. For sending messages we have designed function sendCANmsg(). This function is written in a way to support all properties of the transmit buffer. All the special properties can be set in “prop” argument. In case we don't want to use them, “prop” argument can be used directly as a message size.

```c
/** Send a CAN message
 * \param bi transmit buffer index
 * \param id message identifier
 * \param data pointer to data to be stored
 * \param prop message properties, the octet has following structure:
 *          - bits 7:6 - message priority (higher the better)
 *          - bit  5   - if set, message is remote request (RTR)
 *          - bit  4   - if set, message is considered to have ext. id.
 *          - bits 3:0 - message length (0 to 8 bytes) */
void sendCANmsg(unsigned char bi,
                unsigned long id,
                unsigned char * data,
                unsigned char prop)
{
      /* Initialize reading of the receive buffer */
      spiMasterChipSelect(1);

      /* Send header and address */
      spiMasterTRANSMIT(WRITE_INSTRUCTION);
      spiMasterTRANSMIT(TXBnCTRL(bi));

      /* Setup message priority */
      spiMasterTRANSMIT(prop >> 6);

      /* Setup standard or extended identifier */
      if(prop & 0x10) {
        spiMasterTRANSMIT((unsigned char)(id>>3));
        spiMasterTRANSMIT((unsigned char)(id<<5)
             |(1<<EXIDE)|((unsigned char)(id>>27));
        spiMasterTRANSMIT((unsigned char)(id>>19));
        spiMasterTRANSMIT((unsigned char)(id>>11));
      } else {
        spiMasterTRANSMIT((unsigned char)(id>>3));
        spiMasterTRANSMIT((unsigned char)(id<<5));
      }
      /* Setup message length and RTR bit */
      spiMasterTRANSMIT((prop & 0x0F) | ((prop & 0x20) ? (1 << RTR) : 0));


      /* Store the message into the buffer */
      for(unsigned char i = 0; i < (prop & 0x0F); i++)
            spiMasterTRANSMIT(data[i]);

      /* Release the bus */
      spiMasterChipSelect(0);

      /* Send request to send */
      sendRTS(bi);
}
```

Receiving messages

Message reception is opposite operation to message transmission – register map for received messages is organized in the same way as for transmitted messages. Instead of writing we have to load all the data now. In order to make it as fast as possible we copy whole message registers from MCP2515 to MCU and release the buffer without any analysis.

When the message is received, a falling edge occurs at the INT pin which generates an external interrupt request in the microcontroller. In the device initialization part, we have set the interrupt handler to a function interruptMCP2515() which should handle the situation. First what we have to do is to find out what actually happened. For that we can use instruction “Read RX status” which returns actual receive buffer state. It tells what buffer contains a new message.

```
unsigned char * msgReceived = 0;
unsigned char rbuffer[2][14]; /* 2 RX buffers, each have 14B */

void interruptMCP2515(void)
{
   /* get receive buffer index (we don't consider that both buffer contain
      message, this situation in our environment cannot happen – message is
      directly copied from the buffer and released in this very IRQ ) */
   unsigned char bi = getRXState() >> 6;

   /* Copy the message from the device and release buffer */
   spiMasterTRANSMIT(READ_INSTRUCTION);
   spiMasterTRANSMIT(RXBnCTRL(bi));

   /* Make the local copy */
   for(unsigned char i; i < 14; i++)
      rbuffer[bi][i] = spiMasterTRANSMIT(0);

   msgReceived = &rbuffer[bi];
}
```

In the interrupt handler, we are trying to finish as soon as possible, so the system is not blocked by the IRQ. It only makes local copy of the message and sets the flag which says where the data are stored. If the message is not processed before next one comes, the pointer will be overwritten. This is very simple approach to handle messages and it is not suited for real-life applications, because it may cause many sorts of problems. For example, if system is interrupted while processing the message, the data in the buffer may change. To handle message content, we use a set of macros which dig out the data inside the local copy of the message.

```
#define getData(n) msgReceived[6+i];
#define getId (unsigned short)((msgReceived[1]<<3)|(msgReceived[2]>>5));
#define getLength msgReceived[5] >> 4;
```

Testing solution

We have designed two applications to test MCP2515 – feeder and listener. Feeder sends periodically a CAN message, listener receives the message and sends it via RS-232 to the computer, where we can see it in the terminal. Following source code is for the feeder application.

```
#include "MCP2515.h"
#include "avr/io.h"
#include "avr/interrupt.h"

/* Configuration routine */
void initMCP2515(void)
{
      /* Initialize SPI as a master device, on frequency < 10Mhz */
      spiMasterINIT();
      /* Send reset instruction */
      resetMCP2515();
      /* Set configuration mode */
      setMode(CONFIGURATION_MODE);

      /* Set bit timing only - this is feeder, no listening required */
      setBitTiming(0x04, 0xD2, 0x42);

      /* Get into normal mode and setup communication */
      setMode(NORMAL_MODE)
}

/** Send a CAN message
 * \param bi transmit buffer index
 * \param id message identifier
 * \param data pointer to data to be stored
 * \param prop message properties, the octet has following structure:
 *         - bits 7:6 - message priority (higher the better)
 *         - bit  5   - if set, message is remote request (RTR)
 *         - bit  4   - if set, message is considered to have ext. id.
 *         - bits 3:0 - message length (0 to 8 bytes) */
void sendCANmsg(unsigned char bi,
                unsigned long id,
                unsigned char * data,
                unsigned char prop)
{
      /* Initialize reading of the receive buffer */
      spiMasterChipSelect(1);

      /* Send header and address */
      spiMasterTRANSMIT(WRITE_INSTRUCTION);
      spiMasterTRANSMIT(TXBnCTRL(bi));

      /* Setup message priority */
      spiMasterTRANSMIT(prop >> 6);
      /* Setup standard or extended identifier */
      if(prop & 0x10) {
        spiMasterTRANSMIT((unsigned char)(id>>3));
        spiMasterTRANSMIT((unsigned char)(id<<5)
             |(1<<EXIDE)|((unsigned char)(id>>27));
        spiMasterTRANSMIT((unsigned char)(id>>19));
        spiMasterTRANSMIT((unsigned char)(id>>11));
      } else {
        spiMasterTRANSMIT((unsigned char)(id>>3));
        spiMasterTRANSMIT((unsigned char)(id<<5));
      }

      /* Setup message length and RTR bit */
      spiMasterTRANSMIT((prop & 0x0F) | ((prop & 0x20) ? (1 << RTR) : 0));


      /* Store the message into the buffer */
      for(unsigned char i = 0; i < (prop & 0x0F); i++)
            spiMasterTRANSMIT(data[i]);

      /* Release the bus */
      spiMasterChipSelect(0);

      /* Send request to send */
      sendRTS(bi);
}

/* Make some delay */
void doDelay()
{
      unsigned char i,j,k;
      for(i=0;i<255;i++)
            for(j=0;j<255;j++)
                  for(k=0;k<5;k++);
}

/* Main routine */
int main(void)
{
      unsigned char data[8] = "Hello w!"

      /* initialize CAN */
      initMCP2515();
      /* Enable global interrupt flag */
      sei();
      /* Send the messages periodically */
      while(1) {
            sendCANmsg(0, 0x7E1, data, 8);
            doDelay();
      }

      return 0;
}
```

The feeder just sends the same message periodically; listener is a bit more complicated, because it contains also a UART configuration and communication. For the code see the listing below.

```c
/**
 * Example code - simple CAN listener.
 * This program is listening on the CAN bus and transmits all
 * received messages via the UART (EIA-232) to the computer.
 *
 *
 * Target MCU: ATMega8 (ATMEL AVR)
 * MCU frequency: 16MHz
 *
 * UART Communication speed: 19200Bd
 * UART behaviour: 8-N-1
 *
 * MCP2515 frequency: 25MHz
 * MCP2515 bittiming configuration 0x04, 0xD2, 0x42 (CNF1, CNF2, CNF3) */
#include "MCP2515-interface.h"
#include "MCP2515.h"
#include "avr/io.h"
#include "avr/interrupt.h"

#define getData(n) msgReceived[6+i];
#define getId (unsigned short)((msgReceived[1]<<3)|(msgReceived[2]>>5));
#define getLength msgReceived[5] >> 4;
#define setRollover(v) changeBits(RXB0CTRL, 1 << BUKT, v << BUKT);
#define getMode (readRegister(CANSTAT) >> 5);
#define setMode(mode) { changeBits(CANCTRL, (7 << REQOP0), \
      (mode << REQOP0)); while(getMode != mode); }

unsigned char * msgReceived = (void *)0;
unsigned char rbuffer[2][14]; /* 2 RX buffers, each have 14B */


/* Send a byte via the UART
 * MACHINE DEPENDENT CODE!!! */
void sendByte(unsigned char code)
{
      /* Wait for empty transmit buffer */
      while ( !( UCSRA & (1<<UDRE)) )
      ;
      /* Put data into buffer, sends the data */
      UDR = code;
}

/* Send a string via the UART */
void sendBytes(char data[])
{
      unsigned char i = 0;
      while(data[i] != 0)
            sendByte(data[i++]);
}

/* Initialize the UART
 * MACHINE DEPENDENT CODE!!! */
void initUART(void)
{
      UBRRH=0;
      UBRRL=51;

      UCSRB=(1<<RXCIE)|(1<<RXEN)|(1<<TXEN);
      UCSRC=(1<<URSEL)|(3<<UCSZ0);
}
```

```c
/* Send the number in the hexadecimal format */
void writeInHex(unsigned char number)
{
      sendByte(((number/16)<10)?(number/16)+'0':(number/16)-10+'A');
      sendByte(((number % 16)<10)?(number%16)+'0':(number%16)-10+'A');
}

/* Send the number in the decimal format */
void writeInDec(unsigned char code)
{
      sendByte((code / 100) + '0');
      sendByte(((code % 100) / 10) + '0');
      sendByte(((code % 100) % 10) + '0');
}

/* Interrupt handling routine */
void interruptMCP2515(void)
{
      /* get receive buffer index (we don't consider that both buffer
      contain message, this situation in our environment cannot happen –
      message is directly copied from the buffer and released in IRQ)*/
      unsigned char bi = getRXState() >> 6;

      /* Copy the message from the device and release buffer */
      spiMasterTRANSMIT(READ_INSTRUCTION);
      spiMasterTRANSMIT(RXBnCTRL(bi));

      /* Make the local copy */
      for(unsigned char i; i < 14; i++)
            rbuffer[bi][i] = spiMasterTRANSMIT(0);

      msgReceived = &rbuffer[bi];
}

/* Configuration routine */
void initMCP2515(void)
{
      /* Initialize SPI as a master device, on frequency < 10Mhz */
      spiMasterINIT();
      /* Initialize external interrupt service on this device */
      extInterruptINIT(interruptMCP2515);

      /* Send reset instruction */
      resetMCP2515();
      /* Set configuration mode */
      setMode(CONFIGURATION_MODE);

      /* Set bit timing , masks and rollover mode*/
      setBitTiming(0x04, 0xD2, 0x42);
      setMask(RXM0, 0x00000000, 1);
      setMask(RXM1, 0x00000000, 1);
      setRollover(1);

      /* Get into normal mode and setup communication */
      setMode(NORMAL_MODE)
}
```

```c
/* Main routine */
int main(void)
{
	unsigned char i = 0;

	/* initialize UART peripherals */
	initUART();
	initMCP2515();

	/* set global interrupt flag (MACHINE DEPENDENT) */
	sei();

	/* Say hello to the user */
	sendBytes("\r\n=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-");
	sendBytes("\r\nMCP2515-driver: logger & tracer");
	sendBytes("\r\n=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\r\n");

	while(1) {
		if(msgReceived) {
			/* Send some nice header */
			sendBytes("\r\n-----------------------------------");
			sendBytes("\r\nPrinting received CAN message #");
			sendBytes("\r\n-----------------------------------");

			/* Send standard identifier */
			sendBytes("\r\nSID: 0x");
			writeInHex((unsigned char)getId<<8);
			writeInHex((unsigned char)getId);

			/* Send information about message length (in bytes) */
			sendBytes("\r\nLENGTH: ");
			writeInHex(getLength);

			/* Show received message */
			sendBytes("\r\nMESSAGE: \"");
			for(i=0; i<getLength; i++)
				sendBytes(getData(i));
		}
	}

	return 0;
}
```

Conclusions

We have successfully managed to use MCP2515 to extend CAN capability to AVR family microcontrollers. Implemented solution is quite simple, it doesn't really use all MCP2515 facilities, it ignores many real applications problems, do not solve bus errors, sleep modes and so on, but it is able to communicate over CAN and it is a good start for writing real CAN applications.

```
#ifndef MCP2515_DEFS
#define MCP2515_DEFS

//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
//    MCP2515 pins - PDIP/SOIC Package
//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
#define TXCAN               1      //       Transmit CAN signal (To CAN transciever)
#define RXCAN               2      //       Receive CAN signal (From CAN transciever)
#define CLKOUT_SOF          3      //       Clock out, can be used as clock source for MCU's / SOF signal
#define TX0RTS              4      // (NEG) Request to send input for TXB0 / digital input (with 100k pullup)
#define TX1RTS              5      // (NEG) Request to send input for TXB1 / digital input (with 100k pullup)
#define TX2RTS              6      // (NEG) Request to send input for TXB2 / digital input (with 100k pullup)
#define OSC2                7      //       Oscillator output
#define OSC1                8      //       Oscillator input
#define VSS                 9      //       Ground
#define RX1BF               10     // (NEG) RXB1 full interrupt signal / digital output
#define RX0BF               11     // (NEG) RXB0 full interrupt signal / digital output
#define INT                 12     // (NEG) nterrupt pin (for MCU, reduces SPI overhead)
#define SCK                 13     //       SPI clock
#define SI                  14     //       SPI slave input
#define SO                  15     //       SPI slave output
#define CS                  16     // (NEG) SPI chip select
#define RESET               17     // (NEG) Reset pin (should be connected via RC with tau > 128 osc. ticks)
#define VDD                 18     //       Power supply

//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
//    MCP2515 pins - TSSOP Package
//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
#define TSSOP_TXCAN         1      //       Transmit CAN signal (To CAN transciever)
#define TSSOP_RXCAN         2      //       Receive CAN signal (From CAN transciever)
#define TSSOP_CLKOUT_SOF    3      //       Clock out, can be used as clock source for MCU's / SOF signal
#define TSSOP_TX0RTS        4      // (NEG) Request to send input for TXB0 / digital input (with 100k pullup)
#define TSSOP_TX1RTS        5      // (NEG) Request to send input for TXB1 / digital input (with 100k pullup)
#define TSSOP_TX2RTS        7      // (NEG) Request to send input for TXB2 / digital input (with 100k pullup)
#define TSSOP_OSC2          8      //       Oscillator output
#define TSSOP_OSC1          9      //       Oscillator input
#define TSSOP_VSS           10     //       Ground
#define TSSOP_RX1BF         11     // (NEG) RXB1 full interrupt signal / digital output
#define TSSOP_RX0BF         12     // (NEG) RXB0 full interrupt signal / digital output
```

```
#define TSSOP_INT         13     // (NEG) nterrupt pin (for MCU, reduces SPI overhead)
#define TSSOP_SCK         14     //       SPI clock
#define TSSOP_SI          16     //       SPI slave input
#define TSSOP_SO          17     //       SPI slave output
#define TSSOP_CS          18     // (NEG) SPI chip select
#define TSSOP_RESET       19     // (NEG) Reset pin (should be connected via RC with tau > 128 osc. ticks)
#define TSSOP_VDD         20     //       Power supply

//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
//    MCP2515 instructions & modes & interrupt codes & buffer indexes
//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

#define RESET_INSTRUCTION       0xC0    // Instruction for immediate reset
#define READ_INSTRUCTION        0x03    // Read register
#define WRITE_INSTRUCTION       0x02    // Write register
#define READRX_INSTRUCTION      0x90    // 10010mn0 - m,n is address of the RX buffer
#define LOADTX_INSTRUCTION      0x40    // 01000abc - a,b,c is address of the TX buffer
#define RTS_INSTRUCTION         0x80    // 10000abc - a,b,c is address of the TX buffer
#define READSTAT_INSTRUCTION    0xA0    // Read device status instructio
#define RXSTAT_INSTRUCTION      0xB0    // Read receive buffer status instruction
#define BITMODIFY_INSTRUCTION   0x05    // For specific bit modifications

// Flags in the result of READ STATE instruction
// (Can be used as bit masks for each flag)
#define STATE_RX0_FULL          (0x01 << 0)    // 00000001
#define STATE_RX1_FULL          (0x01 << 1)    // 00000010
#define STATE_TX0_TXREQ         (0x01 << 2)    // 00000100
#define STATE_TX0_EMPTY         (0x01 << 3)    // 00001000
#define STATE_TX1_TXREQ         (0x01 << 4)    // 00010000
#define STATE_TX1_EMPTY         (0x01 << 5)    // 00100000
#define STATE_TX2_TXREQ         (0x01 << 6)    // 01000000
#define STATE_TX2_EMPTY         (0x01 << 7)    // 10000000

//RX status instruction - mask bits in result state
#define RM_POS                  (0x03 << 6)    // Received message position
#define EXT_MSG                 (0x01 << 4)    // Is message extended?
#define REMOTE_MSG              (0x01 << 3)    // Is message remote?
#define AFM                     (0x07 << 0)    // Acceptance filter match
```

```
// Masks for informations contained in the result of RX state instruction
#define RXSTATE_MSG_POSITION    0xC0           // xx000000
#define RXSTATE_MSG_TYPE        0x18           // 000xx000
#define RXSTATE_ACCEPT_FILTER   0x07           // 00000xxx

//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
//    MCP2515 buffers
//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

// Indexes of RX and TX buffers
#define NO_FILTERS              6
#define NO_MASKS                2
#define NO_TX_BUFFERS           3
#define NO_RX_BUFFERS           2
#define TXB0                    0             // Transmit buffer 0
#define TXB1                    1             // Transmit buffer 1
#define TXB2                    2             // Transmit buffer 2
#define RXB0                    0             // Receive buffer 0
#define RXB1                    1             // Receive buffer 1

//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
//    MCP2515 buffers
//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

#define RXF0                   0        // RXB0 filter 0 (or RXB1 filter 0, when rollover is active)
#define RXF1                   1        // RXB0 filter 1 (or RXB1 filter 1, when rollover is active)
#define RXF2                   2        // RXB1 filter 2
#define RXF3                   3        // RXB1 filter 3
#define RXF4                   4        // RXB1 filter 4
#define RXF5                   5        // RXB1 filter 5

//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
//    MCP2515 registers
//=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

// Transmit & receive error counters
#define TEC                     0x1C // Transmit error counter
#define REC                     0x1D // Receive error counter
```

```
// Bit definitions for register BFPCTRL
// (RXn buffer pin control and status)
#define BFPCTRL                 0x0C
#define B1BFS                   5    // RW-0, RX1BF pin state bit
#define B0BFS                   4    // RW-0, RX0BF pin state bit
#define B1BFE                   3    // RW-0, RX1BF pin function enable bit
#define B0BFE                   2    // RW-0, RX0BF pin function enable bit
#define B1BFM                   1    // RW-0, RX1BF pin operation mode bit
#define B0BFM                   0    // RW-0, RX0BF pin operation mode bit

// Bit definitions for register CANCTRL
// (CAN control register)
#define CANCTRL                 0x0F
#define REQOP2                  7    // RW-1, Request operation mode bit 2
#define REQOP1                  6    // RW-0, Request operation mode bit 1
#define REQOP0                  5    // RW-0, Request operation mode bit 0
#define ABAT                    4    // RW-0, Abort all pending transmissions bit
#define OSM                     3    // RW-0, One shot mode
#define CLKEN                   2    // RW-1, CLKOUT pin enable
#define CLKPRE1                 1    // RW-1, CLKOUT pin prescaler 1
#define CLKPRE0                 0    // RW-1, CLKOUT pin prescaler 0

// Bit definitions for register CANSTAT
// (CAN status register)
#define CANSTAT                 0x0E
#define OPMOD2                  7    // R-1, Operation mode bit 2
#define OPMOD1                  6    // R-0, Operation mode bit 1
#define OPMOD0                  5    // R-0, Operation mode bit 0
#define ICOD2                   3    // R-0, Interrupt flag code bit 2
#define ICOD1                   2    // R-0, Interrupt flag code bit 1
#define ICOD0                   1    // R-0, Interrupt flag code bit 0

// Bit definitions for register CNF3
// (Configuration 3 register)
#define CNF3                    0x28
#define SOF                     7    // RW-0, Start of frame signal bit
#define WAKFIL                  6    // RW-0, Wake-up filter bit
#define PHSEG22                 2    // RW-0, PS2 bit 2
#define PHSEG21                 1    // RW-0, PS2 bit 1
```

```
#define PHSEG20                 0    // RW-0, PS2 bit 0

// Bit definitions for register CNF2
// (Configuration 2 register)
#define CNF2                    0x29
#define BTLMODE                 7    // RW-0, PS bit time length bit
#define SAM                     6    // RW-0, Sample point configuration bit
#define PHSEG12                 5    // RW-0, PS1 length bit 2
#define PHSEG11                 4    // RW-0, PS1 length bit 1
#define PHSEG10                 3    // RW-0, PS1 length bit 0
#define PRSEG2                  2    // RW-0, Propagation segment length bit 2
#define PRSEG1                  1    // RW-0, Propagation segment length bit 1
#define PRSEG0                  0    // RW-0, Propagation segment length bit 0

// Bit definitions for register CNF1
// (Configuration 1 register)
#define CNF1                    0x2A
#define SJW1                    7    // RW-0, Synchronization jump width length bit 1
#define SJW0                    6    // RW-0, Synchronization jump width length bit 0
#define BRP5                    5    // RW-0, Baud rate prescaler bit 5
#define BRP4                    4    // RW-0, Baud rate prescaler bit 4
#define BRP3                    3    // RW-0, Baud rate prescaler bit 3
#define BRP2                    2    // RW-0, Baud rate prescaler bit 2
#define BRP1                    1    // RW-0, Baud rate prescaler bit 1
#define BRP0                    0    // RW-0, Baud rate prescaler bit 0

// Bit definitions for register CANINTE
// (Interrupt enable register)
#define CANINTE                 0x2B
#define MERRE                   7    // RW-0, Message error interrupt enable bit
#define WAKIE                   6    // RW-0, Wakeup interrupt enable bit
#define ERRIE                   5    // RW-0, Error interrupt enable bit
#define TX2IE                   4    // RW-0, Transmit buffer 2 empty interrupt enable bit
#define TX1IE                   3    // RW-0, Transmit buffer 1 empty interrupt enable bit
#define TX0IE                   2    // RW-0, Transmit buffer 0 empty interrupt enable bit
#define RX1IE                   1    // RW-0, Receive buffer 1 full interrupt enable bit
#define RX0IE                   0    // RW-0, Receive buffer 0 full interrupt enable bit

// Bit definitions for register CANINTF
// (Interrupt flag register)
#define CANINTF                         0x2C
#define MERRF                           7    // RW-0, Message error interrupt flag bit
#define WAKIF                           6    // RW-0, Wakeup interrupt flag bit
#define ERRIF                           5    // RW-0, Error interrupt flag bit
#define TX2IF                           4    // RW-0, Transmit buffer 2 empty interrupt flag bit
#define TX1IF                           3    // RW-0, Transmit buffer 1 empty interrupt flag bit
#define TX0IF                           2    // RW-0, Transmit buffer 0 empty interrupt flag bit
#define RX1IF                           1    // RW-0, Receive buffer 1 full interrupt flag bit
#define RX0IF                           0    // RW-0, Receive buffer 0 full interrupt flag bit

// Bit definitions for register EFLG
// (Error flag register)
#define EFLG                            0x2D
#define RX1OVR                          7    // RW-0, Receive buffer 1 overflow flag bit
#define RX0OVR                          6    // RW-0, Receive buffer 0 overflow flag bit
#define TXBO                            5    // R-0, Bus-off error flag bit
#define TXEP                            4    // R-0, Transmit error - passive flag bit
#define RXEP                            3    // R-0, Receive error - passive flag bit
#define TXWAR                           2    // R-0, Transmit error warning flag bit
#define RXWAR                           1    // R-0, Receive error warning flag bit
#define EWARN                           0    // R-0, Error warning flag bit

// Bit definitions for registers TXBnCTRL
// (Transmit buffer n control register)
#define TXBnCTRL(n)                     0x30+(n*0x10)
#define TXB0CTRL                        TXBnCTRL(0)
#define TXB1CTRL                        TXBnCTRL(1)
#define TXB2CTRL                        TXBnCTRL(2)
#define ABTF                            6     // R-0, Message aborted flag bit
#define MLOA                            5     // R-0, Message lost arbitration bit
#define TXERR                           4     // R-0, Transmit error detected bit
#define TXREQ                           3     // RW-0, Message transmit request bit
#define TXP1                            1     // RW-0, Transmit buffer priority bit 1
#define TXP0                            0     // RW-0, Transmit buffer priority bit 0

// Bit definitions for registers TXBnDLC
// (Transmit buffer data length control)
#define TXBnDLC(n)                      0x35+(n*0x10)
```

```
#define TXB0DLC                TXBnDLC(0)
#define TXB1DLC                TXBnDLC(1)
#define TXB2DLC                TXBnDLC(2)
#define RTR                    6     // RW-x, Remote transfer request
#define DLC3                   3     // RW-x, Data length code bit 3
#define DLC2                   2     // RW-x, Data length code bit 2
#define DLC1                   1     // RW-x, Data length code bit 1
#define DLC0                   0     // RW-x, Data length code bit 0

// Bit definitions for register TXBnSIDH
// (Transmit buffer n - standard identifier high)
#define TXBnSIDH(n)            0x31+(n*0x10)
#define TXB0SIDH               TXBnSIDH(0)
#define TXB1SIDH               TXBnSIDH(1)
#define TXB2SIDH               TXBnSIDH(2)
#define SID10                  7     // RW-x, Standard identifier, bit 10
#define SID9                   6     // RW-x, Standard identifier, bit 9
#define SID8                   5     // RW-x, Standard identifier, bit 8
#define SID7                   4     // RW-x, Standard identifier, bit 7
#define SID6                   3     // RW-x, Standard identifier, bit 6
#define SID5                   2     // RW-x, Standard identifier, bit 5
#define SID4                   1     // RW-x, Standard identifier, bit 4
#define SID3                   0     // RW-x, Standard identifier, bit 3

// Bit definitions for register TXBnSIDL
// (Transmit buffer n - standard identifier low)
#define TXBnSIDL(n)            0x32+(n*0x10)
#define TXB0SIDL               TXBnSIDL(0)
#define TXB1SIDL               TXBnSIDL(1)
#define TXB2SIDL               TXBnSIDL(2)
#define SID2                   7     // RW-x, Standard identifier, bit 2
#define SID1                   6     // RW-x, Standard identifier, bit 1
#define SID0                   5     // RW-x, Standard identifier, bit 0
#define EXIDE                  3     // RW-x, Extended identifier enable bit
#define EID17                  1     // RW-x, Extended identifier, bit 17
#define EID16                  0     // RW-x, Extended identifier, bit 16

// Bit definitions for register TXBnEID8
// (Transmit buffer n - extended identifier high)
```

```
#define TXBnEID8(n)              0x33+(n*0x10)
#define TXB0EID8                 TXBnEID8(0)
#define TXB1EID8                 TXBnEID8(1)
#define TXB2EID8                 TXBnEID8(2)
#define EID15                    7     // RW-x, Extended identifier, bit 15
#define EID14                    6     // RW-x, Extended identifier, bit 14
#define EID13                    5     // RW-x, Extended identifier, bit 13
#define EID12                    4     // RW-x, Extended identifier, bit 12
#define EID11                    3     // RW-x, Extended identifier, bit 11
#define EID10                    2     // RW-x, Extended identifier, bit 10
#define EID9                     1     // RW-x, Extended identifier, bit 9
#define EID8                     0     // RW-x, Extended identifier, bit 8

// Bit definitions for register TXBnEID0
// (Transmit buffer n - extended identifier high)
#define TXBnEID0(n)              0x34+(n*0x10)
#define TXB0EID0                 TXBnEID0(0)
#define TXB1EID0                 TXBnEID0(1)
#define TXB2EID0                 TXBnEID0(2)
#define EID7                     7     // RW-x, Extended identifier, bit 7
#define EID6                     6     // RW-x, Extended identifier, bit 6
#define EID5                     5     // RW-x, Extended identifier, bit 5
#define EID4                     4     // RW-x, Extended identifier, bit 4
#define EID3                     3     // RW-x, Extended identifier, bit 3
#define EID2                     2     // RW-x, Extended identifier, bit 2
#define EID1                     1     // RW-x, Extended identifier, bit 1
#define EID0                     0     // RW-x, Extended identifier, bit 0

// Bit definitions for registers TXBnDm
// (Transmit buffer N data byte M)
#define TXBnDm(n,m)              0x36+(n*0x10)+m
#define TXBnDm7                  7     // RW-x, Transmit buffer N, data byte M, bit 7
#define TXBnDm6                  6     // RW-x, Transmit buffer N, data byte M, bit 6
#define TXBnDm5                  5     // RW-x, Transmit buffer N, data byte M, bit 5
#define TXBnDm4                  4     // RW-x, Transmit buffer N, data byte M, bit 4
#define TXBnDm3                  3     // RW-x, Transmit buffer N, data byte M, bit 3
#define TXBnDm2                  2     // RW-x, Transmit buffer N, data byte M, bit 2
#define TXBnDm1                  1     // RW-x, Transmit buffer N, data byte M, bit 1
#define TXBnDm0                  0     // RW-x, Transmit buffer N, data byte M, bit 0
```

```
// Bit definitions for register TXRTSCTRL
// (TXn buffer pin control and status)
#define TXRTSCTRL               0x0D
#define B2RTS                   5     // R, TX2RTS pin state bit
#define B1RTS                   4     // R, TX1RTS pin state bit
#define B0RTS                   3     // R, TX0RTS pin function enable bit
#define B2RTSM                  2     // RW-0, TX2RTS pin mode bit
#define B1RTSM                  1     // RW-0, TX1RTS pin mode bit
#define B0RTSM                  0     // RW-0, TX0RTS pin mode bit

// Bit definitions for registers RXBnCTRL
// (Receive buffer n control register)
#define RXBnCTRL(n)             0x60+(n*0x10)
#define RXB0CTRL                RXBnCTRL(0)
#define RXB1CTRL                RXBnCTRL(1)
#define RXM1                    6     // RW-0, Receive buffer operating mode bit 1
#define RXM0                    5     // RW-0, Receive buffer operating mode bit 0
#define RXRTR                   3     // R-0, Receive remote transfer request bit
#define BUKT                    2     // RW-0, Rollover enable bit (used only by RXB0CTRL)
#define FILHIT2                 2     // R-0, Filter hit bit 2 (used only by RXB1CTRL)
#define FILHIT1                 1     // R-0, Filter hit bit 1 (used only by RXB1CTRL)
#define FILHIT0                 0     // R-0, Filter hit bit 0

// Bit definitions for registers RXBnSIDH
// (Receive buffer n standard identifier high)
#define RXBnSIDH(n)             0x61+(n*0x10)
#define RXB0SIDH                RXBnSIDH(0)
#define RXB1SIDH                RXBnSIDH(1)
#define SID10                   7     // RW-x, Standard identifier, bit 10
#define SID9                    6     // RW-x, Standard identifier, bit 9
#define SID8                    5     // RW-x, Standard identifier, bit 8
#define SID7                    4     // RW-x, Standard identifier, bit 7
#define SID6                    3     // RW-x, Standard identifier, bit 6
#define SID5                    2     // RW-x, Standard identifier, bit 5
#define SID4                    1     // RW-x, Standard identifier, bit 4
#define SID3                    0     // RW-x, Standard identifier, bit 3

// Bit definitions for register RXBnSIDL
// (Receive buffer n - standard identifier low)
#define RXBnSIDL(n)                 0x62+(n*0x10)
#define RXB0SIDL                    RXBnSIDL(0)
#define RXB1SIDL                    RXBnSIDL(1)
#define SID2                        7     // RW-x, Standard identifier, bit 2
#define SID1                        6     // RW-x, Standard identifier, bit 1
#define SID0                        5     // RW-x, Standard identifier, bit 0
#define SRR                         4     // RW-x, Standard frame remote transmit request bit, bit 1
#define IDE                         3     // RW-x, Extanded identifier flag bit, bit 0
#define EID17                       1     // RW-x, Extended identifier, bit 17
#define EID16                       0     // RW-x, Extended identifier, bit 16


// Bit definitions for register RXBnEID8
// (Revceive buffer n - extended identifier high)
#define RXBnEID8(n)                 0x63+(n*0x10)
#define RXB0EID8                    RXBnEID8(0)
#define RXB1EID8                    RXBnEID8(1)
#define EID15                       7     // RW-x, Extended identifier, bit 15
#define EID14                       6     // RW-x, Extended identifier, bit 14
#define EID13                       5     // RW-x, Extended identifier, bit 13
#define EID12                       4     // RW-x, Extended identifier, bit 12
#define EID11                       3     // RW-x, Extended identifier, bit 11
#define EID10                       2     // RW-x, Extended identifier, bit 10
#define EID9                        1     // RW-x, Extended identifier, bit 9
#define EID8                        0     // RW-x, Extended identifier, bit 8

// Bit definitions for register RXBnEID0
// (Receive buffer n - extended identifier high)
#define RXBnEID0(n)                 0x64+(n*0x10)
#define RXB0EID0                    RXBnEID0(0)
#define RXB1EID0                    RXBnEID0(1)
#define EID7                        7     // RW-x, Extended identifier, bit 7
#define EID6                        6     // RW-x, Extended identifier, bit 6
#define EID5                        5     // RW-x, Extended identifier, bit 5
#define EID4                        4     // RW-x, Extended identifier, bit 4
#define EID3                        3     // RW-x, Extended identifier, bit 3
#define EID2                        2     // RW-x, Extended identifier, bit 2
#define EID1                        1     // RW-x, Extended identifier, bit 1
```

```
#define EID0                    0     // RW-x, Extended identifier, bit 0

// Bit definitions for registers RXBnDLC
// (Receive buffer data length control)
#define RXBnDLC(n)              0x65+(n*0x10)
#define RXB0DLC                 RXBnDLC(0)
#define RXB1DLC                 RXBnDLC(1)
#define RTR                     6     // RW-x, Remote transfer request
#define DLC3                    3     // RW-x, Data length code bit 3
#define DLC2                    2     // RW-x, Data length code bit 2
#define DLC1                    1     // RW-x, Data length code bit 1
#define DLC0                    0     // RW-x, Data length code bit 0

// Bit definitions for registers RXBnDm
// (Receive buffer N data byte M)
#define RXBnDm(n,m)             0x66 + (n*0x10) + m
#define RXBnDm7                 7     // RW-x, Receive buffer N, data byte M, bit 7
#define RXBnDm6                 6     // RW-x, Receive buffer N, data byte M, bit 6
#define RXBnDm5                 5     // RW-x, Receive buffer N, data byte M, bit 5
#define RXBnDm4                 4     // RW-x, Receive buffer N, data byte M, bit 4
#define RXBnDm3                 3     // RW-x, Receive buffer N, data byte M, bit 3
#define RXBnDm2                 2     // RW-x, Receive buffer N, data byte M, bit 2
#define RXBnDm1                 1     // RW-x, Receive buffer N, data byte M, bit 1
#define RXBnDm0                 0     // RW-x, Receive buffer N, data byte M, bit 0

// Bit definitions for registers RXFnSIDH
// (Filter n standard identifier high)
#define RXFnSIDH(n)             0+(n*4)+((n>2)?4:0)
#define RXF0SIDH                RXFnSIDH(0)
#define RXF1SIDH                RXFnSIDH(1)
#define RXF2SIDH                RXFnSIDH(2)
#define RXF3SIDH                RXFnSIDH(3)
#define RXF4SIDH                RXFnSIDH(4)
#define RXF5SIDH                RXFnSIDH(5)
// Bits are same as in TXBnSIDH

// Bit definitions for register RXFnSIDL
// (Filter n - standard identifier low)
#define RXFnSIDL(n)             1+(n*4)+((n>2)?4:0)
```

```
#define RXF0SIDL                RXFnSIDL(0)
#define RXF1SIDL                RXFnSIDL(1)
#define RXF2SIDL                RXFnSIDL(2)
#define RXF3SIDL                RXFnSIDL(3)
#define RXF4SIDL                RXFnSIDL(4)
#define RXF5SIDL                RXFnSIDL(5)
// Bits are same as in TXBnSIDL

// Bit definitions for register RXFnEID8
// (Filter n - extended identifier high)
#define RXFnEID8(n)             2+(n*4)+((n>2)?4:0)
#define RXF0EID8                RXFnEID8(0)
#define RXF1EID8                RXFnEID8(1)
#define RXF2EID8                RXFnEID8(2)
#define RXF3EID8                RXFnEID8(3)
#define RXF4EID8                RXFnEID8(4)
#define RXF5EID8                RXFnEID8(5)
// Bits EID15-EID8 are same as in TXBnEID8

// Bit definitions for register RXFnEID0
// (Filter buffer n - extended identifier low)
#define RXFnEID0(n)             3+(n*4)+((n>2)?4:0)
#define RXF0EID0                RXFnEID0(0)
#define RXF1EID0                RXFnEID0(1)
#define RXF2EID0                RXFnEID0(2)
#define RXF3EID0                RXFnEID0(3)
#define RXF4EID0                RXFnEID0(4)
#define RXF5EID0                RXFnEID0(5)
// Bits EID7-EID0 are same as in TXBnEID0

// Bit definitions for registers RXMnSIDH
// (Mask n standard identifier high)
#define RXMnSIDH(n)             0x20+(n*4)
#define RXM0SIDH                RXMnSIDH(0)
#define RXM1SIDH                RXMnSIDH(1)
// Bits are same as in TXBnSIDH

// Bit definitions for register RXMnSIDL
// (Mask n - standard identifier low)
```

```
#define RXMnSIDL(n)             0x21+(n*4)
#define RXM0SIDL                RXMnSIDL(0)
#define RXM1SIDL                RXMnSIDL(1)
// Bits are same as in TXBnSIDL

// Bit definitions for register RXMnEID8
// (Filter n - extended identifier high)
#define RXMnEID8(n)             0x22+(n*4)
#define RXM0EID8                RXMnEID8(0)
#define RXM1EID8                RXMnEID8(1)
// Bits EID15-EID8 are same as in TXBnEID8

// Bit definitions for register RXMnEID0
// (Filter buffer n - extended identifier high)
#define RXMnEID0(n)             0x23+(n*4)
#define RXM0EID0                RXMnEID0(0)
#define RXM1EID0                RXMnEID0(1)
// Bits EID7-EID0 are same as in TXBnEID0

/** List of registers which are accessible from BIT MODIFY instruction. You can use following code
 * to declare array with all registers which are supported by "Bit modify" instruction:
 * \code
 * unsigned char test[] = {BIT_MODIFY_SUPPORTED_REGISTERS};
 * \endcode
 */
#define BIT_MODIFY_SUPPORTED_REGISTERS BFPCTRL,TXRTSCTRL,CANCTRL,CNF3,CNF2,CNF1,CANINTE,CANINTF,EFLG,TXB0CTRL,TXB1CTRL,TXB2CTRL,RXB0CTRL,RXB1CTRL

#endif
```