TUTORIAL

Using CAN Bus Network With an ATMEGA32M1

Remy GUYONNEAU, Franck MERCIER
{name}.{SURNAME}@univ-angers.fr

Thanks to the work of Baptiste Hamard.

Preface

The ATMEGA32M1 implements CAN 2.0A/B with 6 message objects – ISO 16845 certified. This document is meant to help developping C program for the ATMEGA32M1 that uses the CAN communication.
Using CAN communication with the ATMEGA32M1 needs generally to handle three steps:

- Configuring CAN Bus
- Configuring MOB
- Handeling the Interruptions

Table of Contents

Configuring CAN Bus

Here are the lines that have to be called to initialize the CAN Bus.

Enable the MCP2562

Note that for the following the MCP2562 (connected to the ATMEGA32M1) Standby pin is assumed to be connected to the PC7 pin of the ATMEGA32M1.

```
DDRC |= 0x80;
```

This command sets the bit7 (DDC7) of the DDRC (Port C Data Direction Register) to 1 : i.e. as an ouput (it is an Input by default otherwise).

```
PORTC &= 0x7F;
```

This command sets the Standby pin (7th bit of port C configured as an output just before) to 0, wich activates the MCP2562.

Initialization of the CAN module

First, the CANGCON register needs to be initialized (CAN Global CONfiguration). That is, we reset the CAN module, just in case, and then we enable the CAN module by doing:

```
CANGCON  = (1 << SWRES); // Reset CAN module
CANGCON  = 0x02; // Enable CAN module
```

SWRES means SoftWare REset requeSt. By setting CANGCON to 0x02, the bit 1 is set to 1 and the others to 0. The bit 1 corresponds to ENA/STB (EnAble/STandBy mode) wich is enabled at 1.
The next step is to configure the CAN speed. This can be done with the CANBTi registers, i={1,2,3}. CANBT meaning CAN Bit Timing. The folowing sets the speed to 500 kb/s.

```
CANBT1 = 0x06; // Speed Config (500kb/s)
CANBT2 = 0x04;
CANBT3 = 0x13;
```

We do not want any priority between the MOB, MOB meaning Message OBject. We configure that by setting the CANHPMOB register to 0, CANHPMOB meaning CAN Highest Priority MOB.

```
CANHPMOB = 0x00; // No priority between the CAN MOB
```

Last thing to configure is to enable CAN interruptions, using the CANGIE register (CAN General Interrupt Enable).

```
CANGIE |= 0xA0;
```

By doing that we set bit 7 and bit 5 of the CANGIE register to 1 (0xA0 = 0b10100000). The 7th bit corresponds to ENIT (ENable all INterrupts) and the 5th bit corresponds to ENRX (Enable Receive Interrupt).

Example of CAN initialization function

Here is a example of CAN initialization function

```
void initCANBus(){
        // Enable of CAN transmit
        DDRC|=0x80;  // Configuration of the standby pin as an output
        PORTC &= 0x7F; // Activation of the MCP2562

        // Initialization of CAN module
        CANGCON  = (1 << SWRES); // Reset CAN module
        CANGCON  = 0x02; // Enable CAN module

        CANBT1 = 0x06; // Speed Config (500kb/s)
        CANBT2 = 0x04;
        CANBT3 = 0x13;

        CANHPMOB = 0x00; // No priority between the CAN MOB

        // Enable CAN interruptions and especially reception interruptions
        CANGIE |= 0xA0;
}
```

Configuring MOB

A MOB is a CAN Message OBject. As specified by the ATMEGA32M1 documentation :

> "The MOb is a CAN frame descriptor. It contains all information to handle a CAN frame. This means that a MOb has been outlined to allow to describe a CAN message like an object. The set of Mobs is the front end part of the "mailbox" where the messages to send and/or to receive are pre-defined as well as possible to decrease the work load of the software. The MObs are independent but priority is given to the lower one in case of multi matching."

Thus, for each interraction with the CAN network (send, receive data) a MOB has to be used. Each MOB is mapped into a page to save place. To select a MOB to modify, the CANPAGE register has to be used, CANPAGE meaning CAN PAGE MOB… The four MSB (Most Significant Bits) of the CANPAGE register allows to select the current MOB number (from 0 to 5 as 6 MOBs are available).
For instance:

```
CANPAGE = 0x00; // select MOB0
// every MOB modification will affect the MOB0 here
CANPAGE = 0x40; // select MOB4
// every MOB modification will affect the MOB4 here
```

Transmit mode MOB

Assuming that we want to use the MOBn to send data over the CAN bus network. The first thing to do is to select the MOBn with CANPAGE and set the CANIDTi registers, i={1, 2, 3, 4}, CANIDT meaning CAN IDentifier Tag.

Setting the Identifier

Here is an extract of the official documentation to ease the use of the CANIDTi registers. Note that the CANIDTi registers depend on the considered CAN bus frame version : extended or not (V2.0B or V2.0A)

V2.0 part A (standard messages)

Bit	15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0	
	-	-	-	-	-	RTRTAG	-	RB0TAG	CANIDT4
	-	-	-	-	-	-	-	-	CANIDT3
	IDT2	IDT1	IDT0	-	-	-	-	-	CANIDT2
	IDT10	IDT9	IDT8	IDT7	IDT6	IDT5	IDT4	IDT3	CANIDT1
Bit	31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16	

V2.0 part B (extended identifier)

Bit	15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0	
	IDT4	IDT3	IDT2	IDT1	IDT0	RTRTAG	RB1TAG	RB0TAG	CANIDT4
	IDT12	IDT11	IDT10	IDT9	IDT8	IDT7	IDT6	IDT5	CANIDT3
	IDT20	IDT19	IDT18	IDT17	IDT16	IDT15	IDT14	IDT13	CANIDT2
	IDT28	IDT27	IDT26	IDT25	IDT24	IDT23	IDT22	IDT21	CANIDT1
Bit	31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16	

With

- RTRTAG : Remote Transmission Request Tag
- IDTx : IDentifier Tag
- RBxTAG : Reserved Bit x TAG

For instance, the following code select the MOB0, set the RTR to 0 (data) and the identifier to 0xC0 (not an extended data frame):

```
#define IDMSGCAN 0xC0 // the message identifier

CANPAGE  = 0x00; //Selection of MOB 0
CANIDT4 = 0x00; // Configure the message as data (rtr = 0)
CANIDT3 = 0x00; // not used for 2.0A messages
CANIDT2 = (uint8_t)( (IDMSGCAN & 0x00F)<< 5 );  // set IDT2, IDT1 and IDT0
CANIDT1 = (uint8_t)( IDMSGCAN >> 3 ); // set the other IDTx bits
```

Setting the data

Now lets configure the data buffer of the data message we want to send. This is done by using the CANMSG (CAN data MeSsaGe) register. It is possible to send 8 bytes of data in one CAN message. To select the corresponding byte to configure using CANMSG we need to use CANPAGE register once again. More precisely the 4 LSB (Lowest Significant Bits) of the register.

For instance:

```
CANPAGE = 0x00; // select the first data byte of MOB0
CANMSG = 0x11; // set the first data byte to 0x11
CANPAGE = 0x01; // select the second data byte of MOB0
CANMSG = 0x22; // set the second data byte to 0x22
CANPAGE = 0x02; // select the third data byte of MOB0
CANMSG = 0x33; // set the third data byte to 0x33
```

Configure the data buffer as 0x112233 in the MOB0 :

Sending the message

Once the message if configured (identifier and data) the last thing to do it is to send it. This is done by using the CANCDMOB register, CANCDMOB meaning CAN MOB Control and DLC.

Here is an extract of the official documentation to ease the use of the CANCDMOB register.

Bit	7	6	5	4	3	2	1	0	
	CONMOB1	CONMOB0	RPLV	IDE	DLC3	DLC2	DLC1	DLC0	CANCDMOB

With

- CONMOB1:0 Configuration of Message Object
  - 00 disable
  - 01 enable transmission
  - 10 enable reception
  - 11 enable frame buffer reception
- RPLV, Reply valid (0 reply not ready, 1 reply ready and valid)
- IDE, Identifier Extension (0 -> 2.0A, 1 -> 2.0B)
- DLC, Data Length Code

For instance, if we want to send the message defined before, we can do:

```
CANCDMOB = 0x43; // send the message using the MOB 0
```

0x42 = 0b 0100 0010, wich means 01 for the CONMOBx registers (enable transmission), 0 for the IDE (standard data frame), and 3 for the DLC (we want to send 3 data bytes: 0x 11 22 33).

Example of sending CAN data function

The following function uses the MOB0 to send a standard data frame, with an identifier on 8 bits.

```
void sendCANdata(uint8_t id, uint8_t dlc, uint8_t* data){
	CANPAGE  = 0x00; //Selection of MOB 0

	CANIDT4 = 0x00; // Config as data (rtr = 0)
	CANIDT3 = 0x00;
	CANIDT2 = (uint8_t)( (id & 0x00F)<< 5 ); // set the identifier
	CANIDT1 = (uint8_t)( id >> 3 );

	for(uint8_t i=0; i< dlc; i++){ // set the data
		CANMSG = data[i];
		CANPAGE = 0x00 + 1 + i;
	}
	CANCDMOB = 0x40 + dlc;// send the message using the MOB 0
	while ( !(CANSTMOB & (1 << TXOK))); // wait for the data to be sent
}
```

Note that in this example we wait for the data to be sent by checking the CANSTMOB register. CANSTMOB meaning CAN Status MOB, and TXOK meaning Transmit OK.

Receive mode MOB

Filtering or not

Configuring a MOB in a reception mode means that the MOB will raise an interruption if a corresponding message occurs on the CAN network. Two options:

- The MOB raises a interruption at each message on the CAN busdisplay:none
- The MOB raises an interruption only if specific messages occur (mask filtering)

Two registers have to be configured to handle and configure those cases: the CANIDT and the CANIDM (CAN Identifier Mask). The CANIDT has already been detailed above.
To simplilfy: the CANIDM defines if the corresponding CANIDT should be filtered or not… That’s all.
A value 0 in the CANIDM means that the corresponding CANIDT value must match to raise an interruption over the received can message, a value 1 means we do not mind the corresponding CANIDT value.

For instance, lets consider a MOB configured with:

```
CANIDT4 = 0x04;  // => RTR = 1
CANIDT3 = 0x00;
CANIDT2 = 0x02;  // => IDT0 = 1
CANIDT1 = 0x10;  // => IDT7 = 1
```

If we have:

```
CANIDM4 = 0x00;
CANIDM3 = 0x00;
CANIDM2 = 0x00;
CANIDM1 = 0x00;
```

The MOB will raise an interruption for every received CAN message (in other words the CANIDTx registers configuration is useless)..

On the other hand if we have:

```
CANIDM4 = 0x04;
CANIDM3 = 0x00;
CANIDM2 = 0x00;
CANIDM1 = 0x00;
```

The MOB will raise an interruption only for a request message (CAN message with RTR = 1).

Finally if we have:

```
CANIDM4 = 0x04;
CANIDM3 = 0xFF;
CANIDM2 = 0xFF;
CANIDM1 = 0xFF;
```

The MOB will raise an interruption only for a request message (RTR=1) with the identifier 0x81 (0b 1000 0001, bit 7 and bit 0 equal to 1, the same as for CANIDTx).

Reception mode

Then, using CANCDMOB, configure the MOB as a reception MOB by doing:

```
CANCDMOB = 0x80;
```

CANCDMOB is detailed above, 0x80 = 0b1000 000, that is CONMOBi = 10 (reception mode).

Enable the interruptions

Do not forget to enable the interruption over the configured MOB using CANIE register, CANIE2 meaning CAN Enable Interrupt MOB. Here is an extract of the document about CANIE2:

Bit	7	6	5	4	3	2	1	0	
	-	-	IEMOB5	IEMOB4	IEMOB3	IEMOB2	IEMOB1	IEMOB0	CANIE2

With:

- IEMOBx : Interrupt Enable MOBx (0 → interrupt disable, 1 → interrupt enable)

It is also needed to enable the general interruptions of the ATMEGA32M1, by using the

```
sei();
```

function (set enable interruption).

Warning : Once an interruption has been raised by a MOB, it is needed to reset the CANSTMOB, reconfigure the CANSTMOB (status), CANCDMOB (Control and DLC) and the CANIE2 (interrupt enable) registers.

For instance if the MOB1 raised an interruption, to catch the next one it is needed to do:

```
CANPAGE = 0x01; // selection of MOB1
CANSTMOB = 0x00; // reset the MOB status
CANCDMOB = 0x80; //re configure the MOB in reception mode
CANIE2 |= 0x02;  // re enable interruption over the MOB1
```

Handeling CAN Interruptions

To handle the CAN bus interruptions, you need to implement the ISR(CAN_INT_vect) function.

```
ISR(CAN_INT_vect){
        //this code is executed each time a CAN interruption occurs
}
```

The first thing to do in this function is to disable the other interruptions, in order not to be disturtb while processing the interruption. This is done by calling the function cli() which disable the ATMEGA interruptions. In other words your ISR function should be like:

```
ISR(CAN_INT_vect){
        cli(); // clear the interruptions
        // write your code here to process the interruption
        sei(); //enable the interruptions
}
```

Test wich MOB raises the Interruption

Assuming that several MOBs are configured to raise an interruption, it can be handy to identify wich MOB actually raised the current one. To do so, you can check the CANSIT2 register (CAN Status InTerruption 2). From the official documentation we have:
CANIE2:

Bit	7	6	5	4	3	2	1	0	
	-	-	SIT5	SIT4	SIT3	SIT2	SIT1	SIT0	CANSIT2

With:
SITx : Status of Interrupt by MOBx (0: no interrupt, 1: MOB interrupt)
For instance if CANSIT2 = 0b0010 0001, it means that MOB0 and MOB5 raised an interruption.
Warning: you have to reset this register if you want to test it again! The flag does not come down to 0 by itself.

TODO
Reset the MOB that raised the interruption

Simple example

Here is a simple example that waits for a remote request message and send a data message as response. MOB0 is used to send the data and MOB1 is used to receive the request.

```
#include <avr/io.h>
#include <avr/interrupt.h>


#define ID 0xC0 // the identifier CAN2.0A for the CAN message (request and data)


// function to initialize the CAN Bus
void initCANBus(){
        // Enable of CAN transmit
        DDRC|=0x80; // Configuration of the standby pin as an output
        PORTC &= 0x7F; // Activation of the MCP2562 (STANDBY to 0)

        // Initialization of CAN module
        CANGCON  = (1 << SWRES); // Reset CAN module
        CANGCON  = 0x02; // Enable CAN module

        CANBT1 = 0x06; // Speed Config (500kb/s)
        CANBT2 = 0x04;
        CANBT3 = 0x13;

        CANHPMOB = 0x00; // No priority between the CAN MOB

        // Enable CAN interruptions and especially reception interruptions
        CANGIE |= 0xA0;
}
```

```
//function to initialize the MOB1 in a receive mode with a filter
void initCANMOB(){
	CANPAGE = 0x10; // selection of MOB1 (MOB1 receive request)

	CANIDT4 = 0x04; // Config as reception remote (rtr = 1)
	CANIDT3 = 0x00;
	CANIDT2 = (uint8_t)( (ID & 0x00F)<< 5 ); // configure the expected ID
	CANIDT1 = (uint8_t)( ID >> 3 );

	CANIDM4 = 0x04; // mask over the rtr value
	CANIDM3 = 0xFF; // mask over the identifier
	CANIDM2 = 0xFF; // mask over the identifier
	CANIDM1 = 0xFF; // mask over the identifier

	CANCDMOB = 0x80; // Config MOB as reception

	CANIE2 = 0x02; // enable the interruptions over the MOB 1, disable the
interruptions over the others MOB
}
```

```
// the main function
int main(void) {
	initCANBus(); // intialization of the CAN Bus
	initCANMOB(); // intialization of the CAN MOB
	sei(); // enable the ATMEGA interruptions

	while(1); // wait for CAN interruption, nothing else to do
}
```

```
// the interruption function
ISR(CAN_INT_vect){
	cli(); // disable the interruption (no to be disturbed when dealing with one)

	// reset the MOB1 configuration for next CAN message
	CANPAGE = 0x10; // Selection of MOB 1
	CANSTMOB = 0x00; // Reset the status of the MOB1
	CANCDMOB = 0x80; // Config as reception MOB1
	CANIE2 = 0x02; // Enable the interuption over MOB 1 (for the next one)

	// send a data message as response to the request using MOB0
	CANPAGE  = 0x00; //Selection of MOB 0

	CANIDT4 = 0x00; // Config as data (rtr = 0)
	CANIDT3 = 0x00;
	CANIDT2 = (uint8_t)( (ID & 0x00F)<< 5 ); // same ID as the request
	CANIDT1 = (uint8_t)( ID >> 3 );

	uint16_t resdata = 0x1234; // the data to be send as the response

	CANMSG = (uint8_t)( ( resdata & 0xFF00) >> 8 ); // update the first byte of data
	CANPAGE = 0x01; // move to next index for CANMSG (canpage : MOB|INDEX)
	CANMSG = (uint8_t)( resdata & 0x00FF ); // update the second byte of data

	CANCDMOB = 0x42; // send the message using the MOB 0, DLC = 2

	sei(); // enable the interruptions
}
```