Assignment notes 5

**Info**

**SPI and CAN controller**

* This exercise will make the basis for enabling CAN network communication in our system.
* It will be realized using a stand-alone **CAN controller with an SPI interface to the MCU** and a **can-transceiver for driving the bus lines**
* The can exercise is quite demanding, and students often need extra time to complete a working solution. It is therefore very important that you read the documentation thoroughly and set aside enough time to do each part properly.
* Firs, you will **implement the spi bus** to enable communication between the mcu and the can controller.
* Then, you will make a **driver** that lets you access to the **can controller’s control register**
* The can controller will be tested using it in a loopback mode (sending to itself)
* In the next exercise, you will enable communication between Node 1 and Node 2, as described in chapter 1.1.

**CAN bus**

* Can (controller are network) is a fault tolerant field bus with excellent real-time characteristics.
* Being a multi-master bus, **ever node connected to the bus can initiate sending of can message and all nodes will receive all message at the same time.**
* The can controller have **ID**s that the can controller can use to decide whether a message should be accepted or ignored.
* Message ID also serves as a priority mechanism. If several node try to send a message simultaneously the **message wit the lowest ID will win the arbitration**
* Arbitration in can is more thoroughly described in the can 2.0B specification.
* **Reading the application notes “AN713; Can Basics”** and “A CAN Physical Layer Discussion;” is also recommended.

**MCP2515**

* This project uses the CAN controller MCP2515 which implements the data link layer and some of the physical layer of the can protocol. Each node will be physically connected to the bus via the transceiver MCP2551
* The MCP2515 is a stand alone can controller with an spi interface
* Is requires the same minimal configuration as an AVR microcontroller: **a clock signal, a reset circuit and a stable power supply.** The reset signal can be taken from the existing reset circuit.
* The controller has several register and a **header file with register names and address provided on the manufacture’s homepage**
* There are several sources of interrupt in the can controller, and MCP2515 **has a common interrupt output pin that should be connected to the atmega162**
* After the interrupt, atmegae should find out which interrupt triggered by **reading the MCP2515 register CANINT**

**SPI**

* SPI is very popular serial, synchronous, full duplex master/slave bus for inter-IC communication. The bus itself consist of three shared between all units connected to the bus, MISO, MOSI, SCK. In addition, each unit has an SS signal and only the slave with this signal active will participate in the transmission.

An SPI transmission using the AVR microcontroller goes on like this:

1. MCU select one of the salves by setting its corresponding SS signal to low.

2. The MCU starts the clock signal when the program writes to the SPI data register (SPDR), given that SPI module is enabled. For each clock period one bit of the SPDR will be shiftet from the master to the slave, and one bit from the slave to the master (MISO)

3. when transmission completes after eight clock periods, SS will be pulled high to indicate that the operation has completed and release the slave.

**Modularization**

* This exercise is complex and relies on the correct functioning of several modules. You are strongly advised to partition your drivers into several layers. This will make your code easier to understand and maintain.
* Both nodes will be using the same CAN controller and reusability will save you a lot of time. An example of how the CAN driver could be organized is shown below

|  |  |
| --- | --- |
| **SPI** | **Content** |
| CAN communication | High level sending and receiving can messages |
| MCP2515 manipulation | Low level driver for setting up the CAN controller and accessing its control and status registers |
| SPI communication | SPI communication driver |

**Exercise**

* make the connection for the MCP2515; clock signal, reset circuit and power supply
* Connect the MCP2515 to the atmega162’s SPI bus and one of its interrupt pins

**Can message** transmissions

**Transmit buffers**

The MCP has three transmit buffers. Each occupies 14 bytes of SRAM.

The first byte

* TXB0CTRL is a control register associated with the message buffer
* Information in this register determines the conditions under which the message will be transmitted and indicates the status of the message transmission
* Five bytes are used to hold the standard or extended identifiers
* The last eight bytes are for the eight possible data bytes of the message to be transmission

**At minimum the TXB0SIDH, TXB0SIDL and TXB0DLC** registers must be loaded. If data bytes are present in the message, the TCB0DM register must also be loaded.

Prior to sending the message the MCU must initialize the CANTE.TWIne bit to enable or disable the generation of interrupt when the message is sent.

**Transmit priority**

Transmit priority is a prioritizing within the MCP of the pending transmittable messages.

Prior to sending the SOF, the priority of all buffers that are queued for transmission is compare . Thr transmit buffer with the highest priority will be sent first.

**We are only using one buffer so this is irrelevant.**

**Initiating transmission**

In order to initiate message transmission., the TXB0CTRL.TXREQ bit must be set for each buffer to be transmitted.

Once the transmission is done, the CANINTF.TX0IF will be set and an **interrupt will be generated if the CANINTE.TX0IE bit is set**

**Note:**

* **When the message is received the CANINTF.TX0IF bit will be set**
* **An interrupt will be generated if the CANINTE.TX0IE bit is set**

**Message reception**

**Receive message buffering**

Includes to full receive buffers with multiple acceptance filters for each. There is also a separate message assepbly buffer that acts as a third buffer.

**Message assembly buffer**

Of the three receive buffers, the MAB is always committed to receiving the next message from the bus. The MAB assembles all messages received. These messages will be transferred to the RXBn buffers

**Rceive flags/interrupts**

When a message is moved into either of the receive buffers, the appropriate CANINTF.RX0IF is set. This must be cleared by the MCU.

**If the CANINTE.RX0IE is set,** an interrupt will be generated on the INTT pin to indicate that a valid message has been received.

**Interrups**

The mcu has eight sources of interrupts.

* The CANINTE register contain the individual interrupts enable bits for each interrupt source.
* The CANINTF register contain the corresponding interrupt flag bit for each interrupt source.

**When an interrupt occurs, the INT pin is driven low by the MCP and will remain low until the interrup is cleared by the MCU.**

**Note**

* **The receive interrupt is set when the CANINTE.RX0IE Is set.**
* **The CANINTF.RX0IF is**