CAN & MCP2515
file:///C:/Users/KOITT/Downloads/mcp2515-avr-can-spi.pdf

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.

우리 프로젝트의 목표는 데이터 시트를 연구하고 8 비트 AVR 마이크로 컨트롤러를 CAN 버스에 연결하는 것입니다. 우리는이 프로젝트를 선택했습니다. MCP2515 집적 회로가 우리에게 더 많은 시간을 요구하기 때문에 저렴한 가격의 마이크로 컨트롤러를 CAN 버스에 연결할 수 있습니다.

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.

우리의 프로젝트의 출력 장치 설정, 메시지의 송수신 방법을 보여주는 샘플 소스 코드해야합니다. 우리는 표준 라이브러리와 함께 GCC 컴파일러와 함께 C 언어를 사용하고 있으며 특별한 타사 라이브러리는 없습니다.

Theoretical background (이론적 배경)

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.

CAN (Control Area Network)은 호스트 컴퓨터없이 2 개 이상의 장치 사이의 통신을 위해 설계된 표준화 된 버스입니다. 일반적으로이 버스는 차량 내부의 전자 장치 (마이크로 컨트롤러) 간의 통신에 사용됩니다. 버스는 예를 들어 도어록, 에어백 제어 장치 등과 같은 차량 내부의 엔진 제어 장치 및 전송 장치 또는 다른 장치를 연결할 수 있습니다. CAN 버스의자동차 사용은 한 가지 방법입니다. 오늘날에는 일반 자동화 환경 (산업 자동화 및 의료 장비)에서 필드 버스로도 사용됩니다. 이 사용법의 이유 중 하나는 컨트롤러 및 프로세서와 같은 CAN 버스 구성 요소의 비용이 낮기 때문입니다.

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 버스의 첫 번째 버전은 1980 년대와 CAN 프로토콜에서 개발되었습니다.
이 버스의 전체 사양은 1990 년대 초반에 지정되었습니다.

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. .

CAN은 제어 장치 연결을 위해 표준화 된 다중 마스터 직렬 버스입니다. 일반적으로 CAN 네트워크로 연결된 제어 장치는 센서 및 액추에이터입니다. 일반적으로 이러한 장치는 마이크로 프로세서 및 CAN 컨트롤러를 통해 버스에 연결됩니다. 이 연결을 통해 장치 (센서 / 액츄에이터)에서 데이터를 검색하고 프로세서 및 컨트롤러에서 처리하고 버스를 통해 다른 장치에 전송할 수 있습니다.

Transmission (전달)

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

멀티 마스터 직렬 버스 란 무엇입니까? 이는 각 장치가 CAN 버스를 통해 메시지를 전송하거나 수신 할 수 있음을 의미하지만 일부 조건이 있습니다. 주로 메세지의 ID와 우선 순위에 따라 다릅니다. 버스가 무료이고 두 대 이상의 장치가 동시에 메시지를 보내기 시작하면 우선 순위가 가장 높은 메시지는 우선 순위가 낮은 다른 메시지를 덮어 씁니다.이는이 최고 우선 순위 메시지 만 남고 버스에 연결된 모든 장치에 수신됨을 의미합니다. 다른 메시지는 버스가 다시 자유롭게 될 때까지 기다려야합니다.

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.

프레임 설명에 표시된 것처럼 각 메시지의 우선 순위는 식별자라는 프레임의 일부로 지정됩니다. 이 식별자에는 11 또는 29 비트가 있습니다 (프레임의 형식에 따라 다름).식별자 0에서 값이 우세하고 열성입니다. 즉, 더 높은 수의 0 (더 작은 값)을 가진 식별자는 주어진 프레임 메시지에 더 높은 우선 순위를 부여합니다.

Frame Format (프레임 형식)

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.

버스상의 장치 들간의 메시지는 프레임 단위로 분배됩니다. CAN 네트워크는 다른 유형의 프레임으로 작동하도록 구성 할 수 있습니다. 가장 중요한 유형의 프레임은 표준 데이터 프레임입니다. 두 번째 중요한 유형은 확장 데이터 프레임입니다. 다이어그램에서 볼 수 있듯이이 두 유형의 유일한 차이점은 표준 프레임이 식별자 용으로 11 비트를 지원하고 확장 프레임에서 식별자 용으로 29 비트를 지원한다는 것입니다. 이 식별자는 기본 11 비트와 확장 18 비트로 구성됩니다. IDE 비트를 사용하여 두 가지 유형의 프레임 간 산만 함을 만듭니다. 이 비트는 11 비트 프레임 식별자의 경우 지배적 (0)으로 설정되고 29 비트 프레임 식별자의 경우 비트는 열성 (1)으로 설정됩니다. CAN 컨트롤러가 확장 프레임 형식의 메시지를 지원하면 표준 프레임 형식으로도 작동 할 수 있습니다. 다시 호환됩니다.

Boundaries of each frame are created by bit Start-of-frame (SOF) at begin and bits End-offrame (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.

각 프레임의 경계는 시작시 비트 Start-of-frame (SOF)과 끝의 비트 End-offrame (EOF)에 의해 생성됩니다. 파트 EOF는 값이 1 인 7 비트 그룹입니다. 시작 비트 (SOF) 프레임 이후에 특정 식별자로 계속 진행하십시오. 이 식별자 다음에 IDE 비트, RTR 비트 (원격 전송 요청) 및 예약 된 비트가옵니다. 이 세 비트 다음에는 DLC 섹션 (데이터 길이 코드)이 4 비트로 구성됩니다. DLC 섹션은 데이터 바이트 수를 선택합니다 (0에서 8 바이트까지). 프레임의 가장 중요한 부분은 데이터 필드이며 DLC 섹션 뒤에 있습니다. 데이터 필드는 0에서 64 비트가 될 수 있습니다. 이 필드 다음에는 CRC 섹션 (순환 중복 확인). 이는 데이터 필드를 보호 할 수있는 오류 감지 코드입니다. CRC 섹션은 15 비트로 구성됩니다. EOF 섹션 (프레임 끝) 전에는 CRC 구분 기호, 승인 비트와 같은 비트가 있습니다.

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.

주어진 CAN 버스의 모든 유닛은 동일한 비트 전송률을 사용해야합니다. 그러나 모든 유닛은 동일한 마스터 발진기 클럭 주파수를 가질 필요가 없습니다. CAN 프로토콜은 NRZ (Non Return to Zero) 코딩을 사용합니다. 이러한 유형의 코딩은 데이터 스트림 내에 클럭을 인코딩하지 않습니다. 따라서 수신 클록은 수신 데이터에 의해 복구되어야하며 송신기의 클럭에 동기화되어야합니다. 이 경우 수신기는 수신기 클록을 동기화하기위한 몇 가지 유형의 PLL (Phase Lock Loop)이 있습니다. 가장자리의 오른쪽 동기화에 비트 스터핑이 사용됩니다.

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.

각 장치의 다른 클럭 주파수의 경우 각 세그먼트의 전송률 프리스케일러 및 시간 퀀텀 수를 적절하게 설정하여 비트 전송률을 조정해야합니다. 각 세그먼트 (CAN 비트 시간)는 겹치지 않는 4 개의 세그먼트 (에지 위상 오차 보상에 사용되는 동기화, 전파 및 위상 세그먼트)로 구성됩니다. 이러한 중첩되지 않는 세그먼트는 각각 시간 퀀텀의 배수로 만들어집니다.
시간 퀀텀 (TQ)은 오실레이터의 주파수에 고정됩니다.

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.

CAN비트 시간의 동기화 세그먼트는 버스 동기화에 사용됩니다. 이 세그먼트에서 비트 에지가 발생할 것으로 예상되며,이 세그먼트는 TQ의 1 배로 고정됩니다. 전파 세그먼트는 노드 간의 물리적 지연을 보상하는 데 사용됩니다. 이 세그먼트는 TQ의 1-8 배의 가변 길이를가집니다. 마지막 2 개의 세그먼트 (1 상 및 2 상)는 버스의 에지 위상 오류를 보상하는 데 사용됩니다. 그것들은 또한 가변 길이가 1 또는 2 (위상에 따라 다름)에서 8 배 TQ까지 있습니다. 연속 동기화는 수신자가 메시지를 제대로 읽을 수 있기 때문에 동기화가 가장 중요한 부분입니다.

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.

CAN의 프로토콜은 각 계층이이 프로토콜의 다른 부분을 설명 할 때 특정 계층으로 분해 될 수 있습니다. CAN 프로토콜로 덮여있는 레이어는 3 개 (객체, 전송 및 물리적 레이어)입니다. 응용 프로그램 계층과 같은 상위 계층은 다른 상위 수준의 프로토콜로도 처리됩니다. 최 하층의 레이어 구조에는 물리적인 레이어가 있습니다. CAN 프로토콜은 전기적인 측면 (전압, 전류, 도체 수)과 같은이 계층에 대한 추상 요구 사항 만 지정했습니다. 그러나 기계적 측면 (커넥터 유형 등)과 같은 다른 요구 사항은 아직 지정되지 않았습니다. 물리적 계층 위에있는 다른 계층은 전송 계층입니다. 대부분의 CAN 프로토콜은이 계층을 적용합니다. 전송 계층은 위의 계층에서 메시지를 수신하고 해당 메시지를 다른 계층으로 전송합니다. 전송 계층은 비트 타이밍, 동기화, 오류 감지, 메시지 프레이밍 등과 같은

많은 프로세스를 담당합니다. CAN 프로토콜에 포함 된 최상위 계층은 객체 계층입니다. 이 부분에서는 메시지 및 상태 처리와 주어진 마스크로 메시지 필터링을 처리합니다.

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.

이 집적 회로는 버전 2.0B에서 CAN 사양을 구현하는 독립형 컨트롤러 Microchip Technology의 장치입니다. 이는 컨트롤러가 표준 및 확장 데이터 프레임을 송수신 할 수 있음을 의미합니다. 버전 2.0B의 CAN 사양은 최대 1Mb / s (버스 길이에 따라 다름)까지 통신 할 수 있습니다. 컨트롤러 MCP2515는 산업 표준 SPI (Serial Peripheral Interface)를 통해 마이크로 프로세서와 통신 할 수 있습니다. 이 인터페이스는 고속 (10MHz)입니다.
또한 마이크로 프로세서의 인터럽트를 호출하기위한 여분의 인터럽트 출력 핀이 있습니다.
MCP2515는 불필요한 메시지를 걸러내어 마이크로 프로세서의 오버 헤드를 줄이기 위해 2 개의 수용 마스크 (각각 29 비트)와 6 개의 수용 필터 (각각 29 비트)를 가지고 있습니다.MCP2515 컨트롤러는 크게 세 부분으로 구성됩니다. 각 부분은이 CAN 컨트롤러의 다른 기능을 다루고 있습니다.

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.

첫 번째 부분은 CAN 프로토콜 엔진, 마스크, 필터, 전송 및 수신을위한 버퍼를 포함하는 CAN 모듈입니다. CAN 모듈은 CAN 버스에서 메시지를 수신하고 전송하는 모든 기능을 처리합니다.
메시지는 먼저 적절한 메시지 버퍼 및 제어 레지스터를로드하여 전송됩니다. 전송은 SPI 인터페이스를 통한 제어 레지스터 비트를 사용하거나 전송 인 에이블 핀을 사용하여 시작된다. 상태 및 오류는 해당 레지스터를 읽음으로써 확인할 수 있습니다.
CAN 버스에서 감지 된 모든 메시지는 오류를 검사 한 다음 사용자 정의 필터와 대조하여 두 개의 수신 버퍼 중 하나로 이동해야하는지 확인합니다.

Control Logic (제어 논리)

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.

CAN 컨트롤러의 두 번째 부분은 정보를 전달하고 제어하기 위해 다른 블록과 인터페이싱하여 장치와 그 동작을 구성하는 데 사용되는 제어 로직입니다. 인터럽트 핀은 시스템 유연성을 높이기 위해 제공됩니다. 유효한 메시지가 수신 버퍼 중 하나로로드되었음을 나타내는 데 사용할 수있는 수신 레지스터 각각에 대해 하나의 다목적 인터럽트 핀이 있습니다. 특정 인터럽트 핀의 사용은 선택 사항입니다. 범용 인터럽트 핀과 상태 레지스터 (SPI 인터페이스로 액세스 가능)는 유효한 메시지가 수신 된시기를 결정하는 데에도 사용할 수 있습니다.
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.

또한 세 개의 전송 레지스터 중 하나에로드 된 메시지의 즉각적인 전송을 시작할 수있는 세 개의 핀이 있습니다.SPI 인터페이스를 통해 액세스되는 제어 레지스터를 사용하여 메시지 전송 시작을 수행 할 수 있으므로이 핀의 사용은 선택 사항입니다.

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.

컨트롤러의 세 번째 부분은 SPI 프로토콜 블록입니다.
프로세서는 SPI 인터페이스를 통해 장치와 인터페이스하며이 블록은 통신을 관리합니다.
모든 레지스터에 쓰기 (및 읽기)는 특수 SPI 명령 외에도 표준 SPI 읽기 및 쓰기 명령을 사용하여 수행됩니다.

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.

우리 프로젝트를 실제로 테스트하기 위해 두 개의 MCU가있는 프로토 타입 보드와 CAN 트랜시버 MCP2551 및 CAN 컨트롤러 MCP2515를 통해 연결된 두 개의 노드로 구성된 CAN 네트워크에 대한 솔루션을 설정했습니다. MCP2515는 SPI 버스 및 INT 핀을 통해 AVR 마이크로 컨트롤러에 연결됩니다. 마이크로 컨트롤러 ATmega8 및 Atmega16은 현재 사용 가능하므로 사용하고 있습니다. 이들은 동일하지는 않지만 동일한 MCU 제품군에 속하며 동일한 SPI 주변 장치를 갖추고 있습니다. 다음 사진은 회로가있는 프로토 타입 보드를 보여줍니다. 이 단계에서는 SPI 버스와 인터럽트 핀이 모두 연결되어 있지 않습니다.

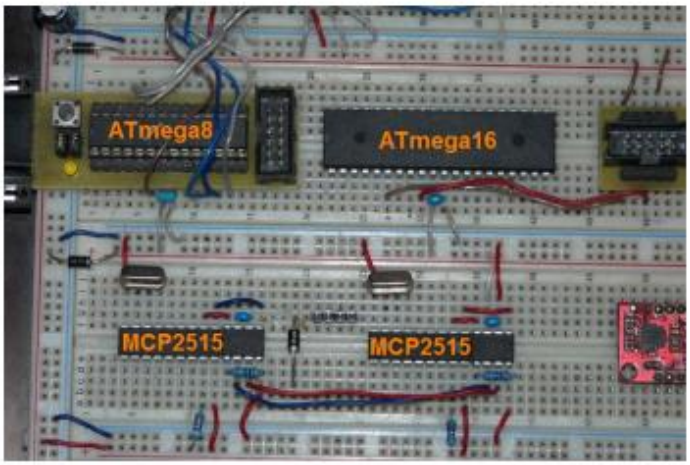

Figure 1: Prototype board solution used for testing purposes.

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.

MCP2515를 MCU에 연결하려면 SPI 버스 (MISO, MOSI, CLK, CS 신호)와 INT 신호를 연결해야합니다. INT 신호는 새로운 메시지가 수신 될 때 MCP2515에 의해 생성 된 인터럽트입니다.

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:

가장 낮은 수준의 솔루션은 장치와 통신하기 위해 만들어진 작은 라이브러리입니다.
이것은 SPI 주변 장치에 대한 액세스, 옥텟 송수신을 허용하는 기능들로 구성됩니다.
또한 MCP2515가 지원하는 모든 명령어 프리미티브를 구현했습니다.
다음 지침이 지원됩니다.

- **Reset instruction** Used for software reset of the device
- 리셋 명령 디바이스의 소프트웨어 리셋에 사용됩니다.
- **Read instruction** Provides read access to the register map
- 읽기 명령어 레지스터 맵에 대한 읽기 액세스를 제공합니다.
- **Read RX buffer instruction** Allows quick access to selected receive buffer
- RX 버퍼 읽기 명령어 선택된 수신 버퍼에 빠르게 액세스 할 수 있습니다.
- **Write instruction** Provides write access to the register map
- 쓰기 명령어 레지스터 맵에 대한 쓰기 액세스를 제공합니다.
- **Load TX buffer instruction** Allows quick access to selected transmit buffer
- 로드 버퍼 명령로드 선택한 전송 버퍼에 빠르게 액세스 할 수 있습니다.

- **Request-to-send (RTS)** Requests to send CAN message in selected buffer
- Request-to-send (RTS) 요청 된 버퍼에서 CAN 메시지를 전송하라는 요청
- **Read device status instruction** Returns information about the device and its buffers.
- 장치 상태 읽기 명령 장치와 해당 버퍼에 대한 정보를 반환합니다.
- **Read RX status instruction** Returns information about received messages

- 수신 RX 상태 명령 수신 메시지에 대한 정보를 반환합니다.
- **Bit modify instruction** Allows changing masked bits in selected register
- 비트 수정 명령 선택된 레지스터의 마스킹 된 비트 변경 가능

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

우리는 MCP2515.h 파일을 만들었습니다.이 파일은 모든 장치 속성과 성능을 설명합니다. 파일은 부록 A를 참조하십시오.

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.

SPI 주변 장치의 초기화는 spiMasterINIT () 함수를 사용하여 수행됩니다. 우리는 SPI 모드를 0,0으로 설정했다 (데이터는 CLK의 상승 에지에서 동시에 수신 및 전송 됨) 및 CLK 주파수는 4Mhz로 설정되었다. SPI에 대한 액세스는 spiMasterTRANSMIT () 함수를 통해 수행됩니다.이 함수는 인수로 전송할 데이터를 취해 전송 프로세스 중에 수신 된 데이터를 반환합니다. SPI는 마스터 디바이스 (이 경우에는 MCU)에 의해 제어되는 전이중 버스입니다. SPI 버스를 통해 데이터를 송수신하는 데이 기능을 사용합니다. 마지막 함수는 통신의 시작 / 끝 신호를 보내는 데 사용되는 spiMasterChipSelect ()입니다. 다음 목록은 위에서 설명한 기능으로 만들어진 하드웨어 액세스 계층을 보여줍니다.

```
#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.

외부 인터럽트는 MCP2515가 상태 변화 (에러, 수신 된 메시지 등)를 알리는 데 사용되는 INT 핀의 변화를 감지하는 데 사용됩니다. 이 핀은 SPI가 마스터 - 슬레이브 버스이기 때문에 필요합니다. MCP2515는 슬레이브로 위치하며 통신을 시작할 수있는 권한이 없습니다. INT 핀은 발생한 중단에 대해 MCU에 알리는 데 사용됩니다.

```
/* 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.

위의 세 가지 기능으로 구성된 하드웨어 액세스 계층은 모든 기본 명령어의 구현에 사용됩니다. 우리는 기본적으로 읽기 레지스터, 쓰기 레지스터 및 리셋 명령 만 필요하지만 실제로 장치와 더 효과적으로 통신 할 수있는 9 가지 유형이 있습니다. 아래 목록은 읽기, 쓰기 및 재설정 명령의 구현을 보여줍니다.

```
/**
* 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.

우리는 모든 명령어를 개별적으로 구현하는 함수를 미리 컴파일하여 (각 명령어마다 오브젝트 파일을 가져옴) 단일 * .a 파일 (gcc의 표준 라이브러리는 이런 방식으로 사전 컴파일 됨)에 압축합니다. 반면에

라이브러리 함수는 외부에서 구현 된 하드웨어 액세스 레이어와 사용 된 마이크로 컨트롤러를 필요로합니다. 라이브러리의 코드는 연결되지 않습니다.
spiMasterINIT (), spiMasterTRANSMIT () 및 spiMasterChipSelect () 함수에 대한 "연결되지 않은"호출이 있습니다.
링커는 라이브러리의 호출을 이러한 함수로 연결하려고 시도합니다. 사용자는이를 호출해야합니다.

이 라이브러리는 하드웨어 액세스 레이어를 포함하지 않으므로 완전히 독립적 인 장치입니다. AVR8 제품군의 다른 마이크로 컨트롤러에서 작동합니다.

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().

장치는 시작 후 자동으로 구성 모드로 전환됩니다. 이 모드에는 쓰기 위해 사용할 수있는 구성 레지스터가 있습니다. 비트 타이밍, 메시지 수신 정책 (마스크 및 필터 수신) 및 원하는 모드 (수신 전용 모드, 루프백 모드, 표준 모드)를 설정해야합니다. 실제 장치 모드를 가져오고 설정하기 위해 매크로 getMode () 및 setMode ()를 준비했습니다.

```
#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().

소개 부분에서 설명한 것처럼 비트 타이밍을 설정해야합니다. 이를 위해 3 비트 타이밍 레지스터 (CNF1, CNF2 및 CNF3)를 설정해야합니다 (MCP2515 데이터 시트, 42 페이지). 이를 위해 setBitTiming () 함수를 준비했습니다.

```
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.

원하는 통신 속도는 250kb / s입니다. 입력 클럭으로 25MHz 크리스털을 사용하고 있습니다. 또한 프리스케일러없이 MCP2515는 비트 타이밍을 위해 입력 클럭을 2로 나눕니다. 따라서 실제 입력 클럭은 12.5 MHz입니다.

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 2.5MHz 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.

250kb / s의 전송 속도를 얻으려면 정확히 50 시간 퀀텀 (50 * 500kHz = 12.5MHz)이 필요합니다. 50 시간 퀀텀은 공칭 비트 시간이 너무 많기 때문에 입력 클럭 프리 스케일러를 5로 나누고 2.5MHz 입력 클록을 얻고, 여기서 1 시간 퀀텀은 400ns이고 공칭 비트 시간은 10 시간 퀀텀입니다
즉, 모든 세그먼트의 크기를 전체 크기가 10이되도록 설정해야한다는 의미입니다. 동기화 세그먼트, 전파 세그먼트, 위상 세그먼트 1 및 위상 세그먼트 2의 4 개의 세그먼트가 있습니다. 동기화 세그먼트는 정확하게 1 시간의 퀀텀을 필요로하며, 이는 우리에게 9 시간의 퀀텀을 설정하게합니다.
측정은 위상 세그먼트 1과 위상 세그먼트 2 사이에서 수행되며, 따라서 공칭 비트 시간의 2/3 전후로 수행되어야하며, 따라서 처음부터 6 또는 7 시간 퀀텀 후에 수행해야합니다.

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.

전파 세그먼트에서 3 개의 시간 퀀텀을 취하는 방식으로 비트 타이밍을 설정하기로 결정했습니다. 3 시간 퀀텀은 위상 세그먼트 1에 의해 취해지며, 측정이 완료되고 마지막 3 시간 퀀텀이 위상 세그먼트 2에 의해 취해 져 명목상 비트 시간이 종료됩니다. 또한 MCP2515는 몇 가지 추가 기능을 설정할 수 있습니다. 하나 대신 3 개의 샘플을 수행 할 수 있으므로 노이즈 내성이 향상됩니다. 우리는 또한 웨이크 업 필터 (wakeup filter) 휴면 모드에서 작동하며 입력 데이터를 필터링합니다. 버스에서 활동이 발생할 때 장치가 깨어나고, 필터는 원치 않는 장치 웨이크 업을 위해 노이즈 저항을 증가시킵니다. 레지스터 값에 따라 CNF1 = 0x04, CNF2 = 0xD2, CNF3 = 0x42입니다.

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:

이 디바이스는 2 개의 수신 버퍼, 6 개의 29 비트 필터 및 2 개의 29 비트 마스크를 가지고있다. 일반적으로 마스크는 필터와 비교되는 식별자의 선택된 비트를 마스크하는 데 사용됩니다. 필터는 선택한 메시지 만 수신하여 장치 오버 헤드를 효과적으로 줄이는 메시지 수용 / 거부 기준을 설정합니다. 일반적으로 필터는 수용 기준에 따라 저장된 메시지의 위치를 결정하지만이 장치는 "롤오버"모드, 수신 메시지는 항상 수신 버퍼 0에 저장되고, 수신 메시지가 가득 차면 버퍼 1을 수신합니다. 우리는이 모드를 사용했으며 모든 메시지를 받고 싶습니다. 롤오버를 설정하려면 다음 매크로를 정의했습니다.

```
#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.

n 번째 수신 필터를 설정하려면 4 개의 레지스터를 설정해야합니다. RXFnSIDH, RXFnSIDL, RXFnEID8 및 RXFnEID0. RXFnSIDH, RXFnSIDL은 표준 식별자 용으로 사용되며 4 개의 레지스터 모두 확장 식별자 용으로 사용됩니다. n 번째 수신 마스크를 설정하려면 필터를 구성 할 때와 같은 방식으로 레지스터를 설정해야합니다. RXMnSIDH, RXMnSIDL, RXMnEID8 및 RXMnEID0 만 레지스터 이름 만 다릅니다.

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().

메시지, 표준 또는 확장 식별자를 수신하려면 수신 필터 및 마스크를 설정해야합니다. 이를 위해 setAcceptanceCriteria () 함수를 생성했습니다. 이 함수는 SPI 하드웨어 액세스 레이어를 직접 사용하며 MCP2515 쓰기 레지스터 명령어 속성을 이용하여 하나의 명령어 안에 4 개의 레지스터를 모두 직접 쓸 수 있습니다. 이 함수의 사용을 단순화하기 위해 setAcceptanceCriteria ()에 필요한 레지스터 주소 대신 마스크 / 필터 인덱스 만 필요한 두 개의 매크로 setMask ()와 setFilter ()를 정의합니다.

```
#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<>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).

수신 된 메시지를 전달하려면 마스크 만 설정하고 필터의 모든 비트를 마스크해야합니다. 그런 다음 수신 된 메시지는 수락 기준을 통과합니다. 이를 위해 마스크를 0으로 설정해야합니다 (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.

전체 초기화 루틴은 다음과 같습니다.
우선 우리는 MCP2515를 제어하기 위해 마이크로 컨트롤러에서 하드웨어 리소스를 초기화 한 다음, 장치가 리셋되어 커밋이 시작 상태인지 확인합니다. 그런 다음 장치를 구성 모드로 설정하면 구성 레지스터에 쓸 수 있습니다. 이제 비트 타이밍, 메시지 허용 정책 및 롤오버 모드를 설정할 수 있습니다. 결국 우리는 정상적인 기능을 시작하는 정상 모드로 장치 상태를 변경합니다.

```
/* 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).

전송 버퍼는 세 가지가 있습니다. 간단한 방법으로 메시지를 보내려면 버퍼를 선택한 다음 데이터로 채운 다음 메시지를 보내도록 요청해야합니다. 설정할 데이터는 메시지 자체 (최대 8 바이트), 식별자 (표준 또는 확장) 및 구성 데이터 (메시지 길이, 우선 순위)로 구성됩니다.

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.

메시지 우선 순위를 설정하고 요청 메시지를 보내려면 TXBnCTRL 레지스터를 사용할 수 있습니다. 그런 다음 메시지 식별자를 설정해야합니다. 허용 기준 (TXBnSIDH, TXBnSIDL, TXBnEID8, TXBnEID0)을 설정할 때와 동일한 구성을 가진 4 개의 레지스터로 구성됩니다. 그 후 메시지 길이 (0 ~ 8 비트)를 설정하고이 메시지가 원격 요청인지 여부를 지정하는 데 사용되는 TXBnDLC 레지스터가 있습니다 (이 경우 메시지 길이는 0 바이트이고 메시지 프레임의 RTR 비트가 설정됩니다).이 후에 메시지 내용에 대해 8 개의 레지스터 (TXBnD0 ~ TXBnD7)가 있습니다.

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.

단일 쓰기 명령으로 전체 송신 버퍼를 설정할 수 있습니다. TXBnCTRL 레지스터에 RTS 비트를 설정한 후 버스가 사용 가능 해지면 메시지가 전송됩니다.RTS 설정은 쓰기 명령을 두 번째로 사용하거나

특수 명령 인 "set RTS"를 사용하여 수행 할 수 있습니다.이 명령은 쓰기 명령보다 1 바이트만큼 오버헤드를 제거하므로 더 빠릅니다.
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.

메시지를 보내기 위해 sendCANmsg () 함수를 설계했습니다. 이 함수는 송신 버퍼의 모든 속성을 지원하는 방식으로 작성됩니다. 모든 특수 속성은 "prop"인수로 설정할 수 있습니다. 우리가 그들을 사용하고 싶지 않은 경우, "prop"인수는 메시지 크기로 직접 사용될 수 있습니다.

```
/** 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<>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.

메시지 수신은 메시지 전송과 반대되는 동작입니다 - 수신 메시지에 대한 레지스터 맵은 전송 된 메시지와 동일한 방식으로 구성됩니다.작성하는 대신 지금 모든 데이터를로드해야합니다. MCP2515에서 MCU로 전체 메시지 레지스터를 가능한 빨리 복사하기 위해 분석없이 버퍼를 해제합니다.

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.

메시지가 수신되면, 마이크로 컨트롤러에서 외부 인터럽트 요청을 생성하는 INT 핀에서 하강 에지가 발생한다. 장치 초기화 부분에서 인터럽트 처리기를 상황을 처리해야하는 MCP2515 () 함수 인터럽트로 설정했습니다.우선 우리가해야 할 일은 실제로 일어난 일을 찾아내는 것입니다. 이를 위해 실제 수신 버퍼 상태를 반환하는 "Read RX status"명령을 사용할 수 있습니다. 어떤 버퍼에 새 메시지가 들어 있는지 알려줍니다.

```
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.

인터럽트 처리기에서 가능한 한 빨리 끝내려고하므로 시스템이 IRQ에 의해 차단되지 않습니다. 메시지의 로컬 복사본 만 만들고 데이터가 저장된 위치를 나타내는 플래그를 설정합니다. 다음 메시지가 오기 전에 메시지가 처리되지 않으면 포인터를 덮어 씁니다. 이것은 메시지를 처리하는 매우 간단한 방법이며 많은 종류의 문제를 일으킬 수 있기 때문에 실제 응용 프로그램에는 적합하지 않습니다. 예를 들어 메시지 처리 중에 시스템이 중단되면 버퍼의 데이터가 변경 될 수 있습니다. 메시지 내용을 처리하기 위해 우리는 메시지의 로컬 사본 안의 데이터를 찾아내는 일련의 매크로를 사용합니다.

```
#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.

우리는 MCP2515 - 피더 및 수신기를 테스트하기 위해 두 가지 애플리케이션을 설계했습니다.
피더는 주기적으로 CAN 메시지를 보내고, 청취자는 메시지를 수신하고 RS-232를 통해 컴퓨터로 보내어 터미널에서 볼 수 있습니다. 다음 소스 코드는 피더 애플리케이션 용입니다.

```
#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<>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 thelisting below.

피더는 주기적으로 동일한 메시지를 보냅니다. 리스너는 UART 구성 및 통신을 포함하기 때문에 조금 더 복잡합니다. 코드는 아래 목록을 참조하십시오.

```
/**
 * 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);
}

/* 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) }

/* 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.

우리는 성공적으로 MCP2515를 사용하여 CAN 기능을 AVR 제품군 마이크로 컨트롤러로 확장했습니다. 구현 된 솔루션은 매우 간단하며 모든 MCP2515 설비를 실제로 사용하지는 않습니다. 실제 응용 프로그램의 많은 문제를 무시합니다. 버스 오류, 절전 모드 등을 해결하지 마십시오. CAN을 통해 통신 할 수 있기 때문에 실제 CAN 애플리케이션을 작성하는 데는 좋은 출발점입니다.

아래에서부터 논문 참조.
file:///C:/Users/KOITT/Downloads/mcp2515-avr-can-spi.pdf