Control del MCP2515 (Tutorial)

Tags: AVR, Software, CAN, avr-gcc
Stand: 6. Juli 2007, 17:36
33 comentarios

Un pequeño Tutorial para empezar con la programación del MCP2515 bajo avr-gcc

Este documento es una introducción paso a paso de como se pueden transmitir mensajes de formato CAN con la ayuda de un AVR (en el presente, un ATMega8) y del MCP2515. Al contrario de la fama de complejidad publicada en los foros, es muy simple de realizar. Obviamente, este documento no puede ser una descripción exhaustiva, ya que el MCP2515 ofrece una gran cantidad de funcionalidades. Espero sin embargo que esta introducción sirva de primer paso.

Para todos los que todavía no saben claramente lo que es un bus CAN o lo que se puede hacer con él, deberían informarse primero, por ejemplo, en Wikipedia o en CAN-Wiki.

Ahora, volvemos al tema del MCP2515. Este tutorial cubre varias áreas:

1. Escribir y leer los registros
2. Inicializar el MCP2515
3. Enviar mensajes CAN
4. Recibir mensajes CAN

Los ejemplos son adaptados para el CAN-Testboard pero se pueden adaptar a otras tarjetas sin mayores cambios.

El MCP2515 es controlado por SPI. Como el AVR trae una interfaz SPI, eso facilita mucho el diseño. Para poder usar esta interfaz, se debe inicializar primero:

C:
```
void spi_init(void)
{
      // Activar el pin como interfaz SPI
      DDR_SPI  |=  (1<<P_SCK) | (1<<P_MOSI);
      PORT_SPI &= ~((1<<P_SCK)| (1<<P_MOSI) | (1<<P_MISO));

      DDR_CS   |= (1<<P_CS);
      PORT_CS  |= (1<<P_CS);

      // Activar la interfaz como SPI Master, fosc = fclk / 2
      SPCR = (1<<SPE)|(1<<MSTR);
      SPSR = (1<<SPI2X);
}
```

Esta secuencia configura la interfaz SPI en modo Master. Se puede tranquilamente elegir la velocidad máxima (fosc / 2 = 8 MHz Clk-Takt para un oscilador de 16 MHz) para la interfaz SPI, ya que el MCP2515 soporta una frecuencia de hasta 10 MHz. No obstante, si la PCB no está bien diseñada, esta frecuencia puede dar problemas y, en este caso, se debe elegir una frecuencia más baja. Sobre mi placa de prueba, el bus SPI opera con 3,7MHz sin problema (por el quartz de 7,3728MHz del AVR).

En la inicialización, se debe averiguar que los pins que corresponden son configurados como salida (SCK, MOSI, \CS) o entrada (MISO). Al contrario de muchas otras interfaces de los AVR (UART, TWI etc.), la dirección de los pins del SPI no se define automáticamente en el chip. Para efecto de portabilidad a otros AVR, definí los pins con unos *#defines*. Así, la adaptación se ejecuta en un solo lugar si un día, por ejemplo, el \CS debe cambiar de pin (los otros 3 pins son específicos a cada AVR y no pueden ser cambiados, salvo si la funcionalidad SPI se implementa en software).

C:
```
#define DDR_CS    DDRB
#define PORT_CS   PORTB
#define P_CS      2

#define DDR_SPI   DDRB
#define PORT_SPI  PORTB
#define P_MISO    4
#define P_MOSI    3
#define P_SCK     5
```

Para no estar obligado de revisar las direcciones de los registros del MCP2515 a cada rato, escribí un archivo de cabecera con los principales *#define*. Así, se pueden fácilmente inferir los nombres de los registros y de los bits en el código fuente. Este archivo se puede bajar de acá .

Ahora, se pueden enviar bytes por el bus SPI:

C:
```
uint8_t spi_putc( uint8_t data )
{
      // Enviar un Byte
      SPDR = data;

      // Esperar hasta el final de la transmisión
      while( !( SPSR & (1<<SPIF) ) )
            ;

      return SPDR;
}
```

Como el bus SPI transmite y recibe desde pins distintos, se puede operar en full-duplex. Para recibir un byte, se debe enviar un byte de contenido arbitrario y evaluar el dato del lado del receptor.

C:
```
// Enviar un dato por SPI
spi_putc(0xaa);

// Leer dato por SPI
data = spi_putc(0xff);
```

Ahora, podemos comunicar con el MCP2515. La comunicación sigue siempre el mismo esquema:

- Bajar /CS (activo bajo)
- Enviar el comando SPI (ver especificación)
- Enviar y recibir los datos relevantes
- Subir /CS (inactivo alto)

Las funciones de escritura y lectura se describen a continuación:

Escribir en registro

C:
```
void mcp2515_write_register( uint8_t adress, uint8_t data )
{
      // Activar el /CS del MCP2515 (activo bajo)
      PORT_CS &= ~(1<<P_CS);
```

```
      spi_putc(SPI_WRITE);
      spi_putc(adress);
      spi_putc(data);

      // Liberar el /CS
      PORT_CS |= (1<<P_CS);
}
```

Leer registro

C:

```
uint8_t mcp2515_read_register(uint8_t adress)
{
      uint8_t data;

      // Activar el /CS del MCP2515 (activo bajo)
      PORT_CS &= ~(1<<P_CS);

      spi_putc(SPI_READ);
      spi_putc(adress);

      data = spi_putc(0xff);

      // Liberar el /CS
      PORT_CS |= (1<<P_CS);

      return data;
}
```

Otra funcionalidad práctica del MCP2515 es la modificación de bit. Eso permite setear o resetear bits individuales. Se presenta un ejemplo de eso en el proceso de inicialización, que permite cambiar el modo de operación:

Setear/resetear Bits individuales

C:

```
void mcp2515_bit_modify(uint8_t adress, uint8_t mask, uint8_t data)
{
      // Activar el /CS del MCP2515
      PORT_CS &= ~(1<<P_CS);

      spi_putc(SPI_BIT_MODIFY);
      spi_putc(adress);
      spi_putc(mask);
      spi_putc(data);

      // Liberar el /CS
      PORT_CS |= (1<<P_CS);
}
```

Se debe notar que esta función opera solamente sobre ciertos registros (ver especificación p. 59). Se envía primero una máscara que define los bits a afectar. Después vienen los datos significativos. Si un bit está en 1 en la máscara y en 0 en el dato significativo, se resetea el bit que corresponde en el registro. Si está en 1 en el dato significativo, se setea el bit en el registro. Si el bit de la máscara está en 0, el registro no cambia.

Ahora podemos inicializar el MCP2515 para enviar y recibir mensajes. La única complejidad es la configuración de la velocidad del bus CAN.

Para poder escribir en los registros de configuración del MCP2515, se debe poner primero en modo de configuración. Sólo en este modo se pueden escribir los registros **CNF1**, **CNF2**, **CNF3**, **TXRTSCTRL** y los filtros y máscaras.

Hay todavía algunos otros modos que voy a omitir acá ya que la especificación los describe muy claramente. :-)

Configuración de los timings de bits

Como este tema es muy complejo, empiezo solamente con unos links a leer con valores típicos para varias velocidades. La velocidad se configura con los 3 registros **CNF1** a **CNF3**:

C:

```
// CAN Bitrate 125 kbps
#define R_CNF1    (1<<BRP2)|(1<<BRP1)|(1<<BRP0)
#define R_CNF2    (1<<BTLMODE)|(1<<PHSEG11)
#define R_CNF3    (1<<PHSEG21)

// CAN Bitrate 250 kbps
#define R_CNF1    (1<<BRP1)|(1<<BRP0)
#define R_CNF2    (1<<BTLMODE)|(1<<PHSEG11)
#define R_CNF3    (1<<PHSEG21)

// CAN Bitrate 500 kbps
#define R_CNF1    (1<<BRP0)
#define R_CNF2    (1<<BTLMODE)|(1<<PHSEG11)
#define R_CNF3    (1<<PHSEG21)

// CAN Bitrate 1 Mbps
#define R_CNF1    0
#define R_CNF2    (1<<BTLMODE)|(1<<PHSEG11)
#define R_CNF3    (1<<PHSEG21)
```

Para los que quieren saber como se definen estos valores, recomiendo leer la especificación (p.37 y siguientes) o los links siguientes:

http://www.intrepidcs.com/BitCindex.html

http://www.kvaser.com/index.html

En particular, el “Microchip CAN Bit Timing Calculator” es ideal para una representación gráfica y para probar varias configuraciones.

Interrupciones

Hay 8 fuentes distintas de interrupciones:

1. Message Error Interrupt
2. Wakeup Interrupt
3. Error Interrupt Enable (multiple sources in EFLG register)
4. Transmit Buffer 2 Empty Interrupt
5. Transmit Buffer 1 Empty Interrupt
6. Transmit Buffer 0 Empty Interrupt
7. Receive Buffer 1 Full Interrupt

8. Receive Buffer 0 Full Interrupt

Cuando se activan estas interrupciones, la línea de interrupción se activa y queda activa hasta que la última fuente sea procesada.

En mi programa, usé solamente las interrupciones de “buffer lleno” para indicar que un nuevo mensaje llegó. Obviamente, hay muchas otras posibilidades de uso de las interrupciones – más detalles p.49 de la especificación técnica.

Para activar la interrupción, se debe setear el bit correspondiente en CANINTE:

C:
```
// Aktivieren der Rx Buffer Interrupts
mcp2515_write_register(CANINTE, (1<<RX1IE)|(1<<RX0IE));
```

Funciones de los Pins

El MCP2515 ofrece también 5 pines que se pueden usar de varias formas. Los pines **RX0BF** y **RX1BF** pueden servir de marcador de interrupción o de salida digital de uso genérico (GPIO). Así, se pueden conectar, por ejemplo, 2 LEDS sin requerir I/Os del AVR.

El uso de estos pines se controla mediante **BFPCTRL**. Como este registro soporta la función de modificación de bit, estos pines son muy fáciles de usar.

C:
```
// Usar RX0BF como GPOut
mcp2515_bit_modify( BFPCTRL, (1<<B0BFE)|(1<<B0BFM), (1<<B0BFE));

// Setear el pin
mcp2515_bit_modify( BFPCTRL, (1<<B0BFE), (1<<B0BFE));
```

El pin **RX1BF** se controla de la misma manera pero, obviamente, se puede usar como salida de interrupción:

C:
```
// Usar RX0BF y RX1BF como Interrupciones
mcp2515_write_register( BFPCTRL, (1<<B1BFE)|(1<<B0BFE)|(1<<B1BFM)|(1<<B0BFM));
```

De la misma manera, se pueden usar los pins **TxnRTS** como GPIO o para gatillar el envío del mensaje siguiente. La selección se hace con **TXRTSXTRL** (ver p.19 de la especificación técnica). El control se hace de la misma manera que los **RxnBF**.

CLKOUT Pin

El MCP2515 ofrece la posibilidad de proporcionar una señal de clock a otros componentes. El pin **CLKOUT** del MCP2515 genera la señal, de tal forma que se puede conectar a otro microcontrolador. Además, contiene un divisor por 2, 4 o 8, que permite generar frecuencias adicionales. Con un clock de 16MHz, se puede contar con una frecuencia de 16, 8, 4 o 2MHz.

El pin **CLKOUT** se controla con los 3 últimos bits del registro **CANCTRL**. Como el pin está activo por defecto al alimentar el MCP2515, se puede contar con esta señal para arrancar el AVR. (Nota del traductor: verificar que entendí bien...)

C:
```
// Resetear CLKOUT
// => fuerza la frecuencia de clock del MCP2515 en CLKOUT
mcp2515_bit_modify( CANCTRL, 0x07, (1<<CLKEN));
```

No obstante, se debe tener cuidado con el manejo del RESET: se debe evitar a toda costa de

conectar directamente el RESET del AVR con él del MCP2515, ya que eso puede impedir la programación del AVR. Durante la fase de programación del AVR, se debe activar el RESET del microcontrolador, lo que resetearía también el MCP2515 y deactivaría CLKOUT. En este caso, el AVR no recibiría su clock. La única solución es separar las líneas de reset y poner un pull-up sobre el MCP2515. Eso requiere que el microcontrolador efectúe un reset en software durante la fase de inicialización para asegurar que el MCP2515 parte en un estado conocido.

Filtro de aceptación y máscara

Como lo mencionamos desde el principio, una fortaleza del bus CAN (o mejor dicho del controlador CAN) es la habilidad de filtrar mensajes. El MCP2515 ofrece 2 filtros para buffer 0 y 4 para buffer 1. Así, se pueden ignorar desde la capa MAC los mensajes que no se dirigen a un nodo en particular, sin que el AVR tenga que tomar la decisión.

Acerca de los filtros, se encuentra en la especificación la tabla a continuación:

Mask Bit	Filter Bit	Message Identifier bit	
0	x	x	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

Los bits del identificador se toman en cuenta solamente si el bit de máscara está en 1. En caso de que todos los mensajes se deban interpretar (por ejemplo en fase de prueba), basta con forzar el bit de máscara en 0. Recuerde que los bits de filtro y de máscara se pueden modificar sólo en modo de configuración.

Ahora, un ejemplo de inicialización:

C:

```
void mcp2515_init(void)
{
      // Inicializar la interfaz SPI del AVR
      spi_init();

      // Resetear el MCP2515 en Software, lo que deja
      // el MCP2515 en modo de configuración
      PORTB &= ~(1<<SPI_CS);
      spi_putc( SPI_RESET );
      PORTB |= (1<<SPI_CS);

      /*
      *     Configurar los Timings de bits (velocidad)
      *
      *     Fosc = 16MHz
      *     BRP = 7            (dividir por 8)
      *     TQ = 2 * (BRP + 1) / Fosc (=> 1 uS)
      *
      *     Sync Seg = 1TQ
      *     Prop Seg = (PRSEG + 1) * TQ = 1 TQ
      *     Phase Seg1 = (PHSEG1 + 1) * TQ = 3 TQ
      *     Phase Seg2 = (PHSEG2 + 1) * TQ = 3 TQ
      *
```

```
	*     Bus speed   = 1 / (Total # of TQ) * TQ
	*                 = 1 / 8 * TQ = 125 kHz
	*/

	// BRP = 7
	mcp2515_write_register( CNF1, (1<<BRP0)|(1<<BRP1)|(1<<BRP2) );

	// Configurar Prop Seg y Phase Seg1
	mcp2515_write_register( CNF2, (1<<BTLMODE)|(1<<PHSEG11) );

	// Desactivar Wake-up Filter, configurar Phase Seg2
	mcp2515_write_register( CNF3, (1<<PHSEG21) );

	// Activar Rx Buffer Interrupts
	mcp2515_write_register( CANINTE, (1<<RX1IE)|(1<<RX0IE) );

	/*
	 * Configuración del filtro
	 */

	// Buffer 0 : Recibir todos los mensajes
	mcp2515_write_register( RXB0CTRL, (1<<RXM1)|(1<<RXM0) );

	// Buffer 1 : Recibir todos los mensajes
	mcp2515_write_register( RXB1CTRL, (1<<RXM1)|(1<<RXM0) );

	// Borrar todos los bits de la máscara de recepción
	// para recibir todos los mensajes
	mcp2515_write_register( RXM0SIDH, 0 );
	mcp2515_write_register( RXM0SIDL, 0 );
	mcp2515_write_register( RXM0EID8, 0 );
	mcp2515_write_register( RXM0EID0, 0 );

	mcp2515_write_register( RXM1SIDH, 0 );
	mcp2515_write_register( RXM1SIDL, 0 );
	mcp2515_write_register( RXM1EID8, 0 );
	mcp2515_write_register( RXM1EID0, 0 );

	/*
	 * Configurar los pines de funciones
	 */

	// Desactivar los pines RXnBF (High Impedance State)
	mcp2515_write_register( BFPCTRL, 0 );

	// Configurar los bits TXnRTS como entradas
	mcp2515_write_register( TXRTSCTRL, 0 );

	// Pasar el chip en modo normal
	mcp2515_bit_modify( CANCTRL, 0xE0, 0);
}
```

La configuración de los registros se puede efectuar de una manera más elegante todavía cargando los valores en la Flash del AVR y de allí transmitirlos (ver último capítulo).

Ahora, estamos casi listos para el primer envío de mensaje en el bus CAN. En este capítulo, presentaré solamente el envío de ID estándares (11 bits de largo). Los que lo quieran pueden

fácilmente extender el código para soportar los de 29 bits.

El MCP2515 tiene 3 buffers de emisión independientes con prioridad configurable. Para enviar un mensaje, se deben cargar los registros de ID, el campo de datos y gatillar el envío.

Para gatillar el envío, hay 3 posibilidades:

- Escribir los bits correspondientes en los registros **TXBnCTRL**
- Enviar el comando **RTS** por SPI
- Forzar a 0 el pin **TXnRTS** del buffer que corresponde.

Si el microcontrolador tiene pocos pines libres, se recomienda manejar todo por SPI. Eso ahorra entre 1 y 3 I/Os según la cantidad de buffer en uso en la aplicación sin afectar demasiado la latencia.

El envío se puede hacer como sigue:

C:
```
/* Enviar un mensaje sobre buffer 0
 * 2 bytes de datos (0x04, 0xf3)
 * Standard ID: 0x0123
 */
uint16_t id = 0x0123;

/* Configurar el buffer de mensaje con la prioridad más alta
 * (no es necesario si se usa solamente 1 buffer, ver texto)
 */
mcp2515_bit_modify( TXB0CTRL, (1<<TXP1)|(1<<TXP0), (1<<TXP1)|(1<<TXP0) );

// Configurar ID
mcp2515_write_register(TXB0SIDH, (uint8_t) (id>>3));
mcp2515_write_register(TXB0SIDL, (uint8_t) (id<<5));

// Configurar el largo del dato + RTR
mcp2515_write_register(TXB0DLC, 2);

// Datos
mcp2515_write_register(TXB0D0, 0x04);
mcp2515_write_register(TXB0D1, 0xf3);

// Enviar mensaje CAN
PORT_CS &= ~(1<<P_CS);
spi_putc(SPI_RTS | 0x01);
PORT_CS |= (1<<P_CS);
```

Transmitir con un buffer

No es muy práctico tener que cargar los registros cada vez. Necesitamos una función que lo haga sola. Para facilitar el intercambio de mensajes entre funciones, se recomienda juntar los campos en una estructura, que se puede pasar fácilmente en argumento:

C:
```
typedef struct
{
      uint16_t    id;
      uint8_t     rtr;
```

```
      uint8_t     length;
      uint8_t     data[8];
} CANMessage;

// Crear un nuevo mensaje
CANMessage message;

// Cargar datos
message.id = 0x0123;
message.rtr = 0;
message.length = 2;
message.data[0] = 0x04;
message.data[1] = 0xf3;

// Enviar mensaje
can_send_message(&message);
```

La función de emisión que corresponde se puede estructurar como sigue:

C:

```
void can_send_message(CANMessage *p_message)
{
      uint8_t length = p_message->length;

      // Configurar ID
      mcp2515_write_register(TXB0SIDH, (uint8_t) (p_message->id>>3));
      mcp2515_write_register(TXB0SIDL, (uint8_t) (p_message->id<<5));

      // Es el mensaje un "Remote Transmit Request" ?
      if (p_message->rtr)
      {
            /* Un mensaje RTR tiene un largo pero no tiene datos */

            // Configurar largo de mensaje + RTR
            mcp2515_write_register(TXB0DLC, (1<<RTR) | length);
      }


      else
      {
            // Configurar largo de mensaje
            mcp2515_write_register(TXB0DLC, length);

            // Datos
            for (uint8_t i=0;i<length;i++) {
                  mcp2515_write_register(TXB0D0 + i, p_message->data[i]);
            }
      }

      // Enviar mensaje CAN
      PORT_CS &= ~(1<<P_CS);
      spi_putc(SPI_RTS | 0x01);
      PORT_CS |= (1<<P_CS);
}
```

Obviamente, esta función se puede extender para usar siempre el primer buffer libre evaluando el bit TXREQ del registro TXBnCTRL y elegir el buffer en cuanto a eso. Eso se facilita con un comando

SPI particular (Read Status Instruction):

Enviar con los 3 buffers

Al contrario del primer ejemplo, no queremos usar funciones preparadas esta vez para cargar los registros sino acelerar un poco la transmisión pasando los datos directamente sobre el bus SPI, que el MCP2515 enviará en secuencia. Se programa de la forma siguiente:

C:

```
uint8_t can_send_message(CANMessage *p_message)
{
      uint8_t status, address;

      // Leer el estado del MCP2515
      PORT_CS &= ~(1<<P_CS);
      spi_putc(SPI_READ_STATUS);
      status = spi_putc(0xff);
      spi_putc(0xff);
      PORT_CS |= (1<<P_CS);

      /* Byte de estado:
       *
       * Bit Función
       * 2   TXB0CNTRL.TXREQ
       * 4   TXB1CNTRL.TXREQ
       * 6   TXB2CNTRL.TXREQ
       */

      if (bit_is_clear(status, 2)) {
            address = 0x00;
      }
      else if (bit_is_clear(status, 4)) {
            address = 0x02;
      }
      else if (bit_is_clear(status, 6)) {
            address = 0x04;
      }
      else {
            /*Todos los buffers estan ocupados, no se puede transmitir el mensaje*/
            return 0;
            }

      PORT_CS &= ~(1<<P_CS);        // CS Low
      spi_putc(SPI_WRITE_TX | address);

      // Configurar Standard ID
      spi_putc((uint8_t) (p_message->id>>3));
      spi_putc((uint8_t) (p_message->id<<5));

      // Extended ID
      spi_putc(0x00);
      spi_putc(0x00);

      uint8_t length = p_message->length;
```

```
      if (length > 8) {
            length = 8;
      }

      // Es el mensaje un "Remote Transmit Request" ?
      if (p_message->rtr)
      {
            /* Un RTR tiene un largo pero no tiene datos */

            // Configurar largo de mensaje + RTR
            spi_putc((1<<RTR) | length);
      }
      else
      {
            // Configurar largo de mensaje
            spi_putc(length);

            // Datos
            for (uint8_t i=0;i<length;i++) {
                  spi_putc(p_message->data[i]);
            }
      }
      PORT_CS |= (1<<P_CS);         // CS = High

      asm volatile ("nop");

      /* Enviar el mensaje CAN los ultimos 3 bits del comando RTS indican
       * a que buffer se deben enviar los datos.
       */
      PORT_CS &= ~(1<<P_CS);        // CS = Low
      if (address == 0x00)
      {
            spi_putc(SPI_RTS | 0x01);
      }
      else {
            spi_putc(SPI_RTS | address);
      }
      PORT_CS |= (1<<P_CS);         // CS = High

      return 1;
}
```

En teoría, puede suceder que un mensaje se quede trancado si se envían muchos mensajes uno trás el otro, ya que el primer buffer tiene una prioridad más alta y que se transmite primero. Para evitar eso, se debería configurar la prioridad del buffer actual con la menor prioridad e incrementar la prioridad de los otros buffers. Eso aseguraría que los mensajes serían transmitidos en la secuencia correcta.

Para verificar que algo está sucediendo, se necesita una capa de recepción, lo que vamos a ver ahora.

Recibir mensajes

Para saber si llegó un mensaje, hay varias posibilidades, según la configuración:

- a través de los pines **RXnBF**
- por el pin de interrupción
- por comando SPI

Obviamente, estas posibilidades no son exclusivas y se pueden combinar. Una posibilidad recomendable es, por ejemplo, conectar la señal de interrupción al AVR para detectar la llegada de un mensaje y preguntar por SPI en qué buffer está. Como la placa de prueba de CAN tiene esta configuración, le voy a presentar esta solución en más detalle:

Para poder usar el pin de interrupción como modo de detección de llegada de mensaje, se debe configurar como corresponde. Como ya lo hemos hecho en la inicialización (ver sección Interrupciones), podemos sencillamente esperar hasta que el pin de interrupción baje y que nos muestre así, que tenemos un mensaje a disposición.

Para detectar en qué buffer está el mensaje o de qué tipo es (ID estándar, ID extendido, Remote Transfer Request, etc.), el MCP2515 soporta un comando dedicado (sino, se puede también, leer **CANINTF**):

C:
```
uint8_t mcp2515_read_rx_status(void)
{
      uint8_t data;

      // Bajar /CS sobre el MCP2515
      PORT_CS &= ~(1<<P_CS);

      spi_putc(SPI_RX_STATUS);
      data = spi_putc(0xff);

      // Los datos se van a repetir, así que se requiere leer solo uno de los 2
      spi_putc(0xff);

      // Liberar /CS
      PORT_CS |= (1<<P_CS);


      return data;
}
```

Verificamos primero si el pin de interrupción está bajo. Si es el caso, un mensaje está disponible.

C:
```
uint8_t can_get_message(CANMessage *p_message)
{
      // Leer estado
      uint8_t status = mcp2515_read_rx_status();



      if (bit_is_set(status,6))
      {
            // Mensaje en buffer 0
            PORT_CS &= ~(1<<P_CS);        // CS Low
            spi_putc(SPI_READ_RX);
```

```
      }


      else if (bit_is_set(status,7))
      {
            // Mensaje en buffer 1
            PORT_CS &= ~(1<<P_CS);        // CS Low
            spi_putc(SPI_READ_RX | 0x04);
      }
      else {
            /* Error: no hay mensaje disponible */
            return 0xff;
      }

      // Leer Standard ID
      p_message->id  = (uint16_t) spi_putc(0xff) << 3;
      p_message->id |= (uint16_t) spi_putc(0xff) >> 5;

      spi_putc(0xff);
      spi_putc(0xff);

      // Leer el largo
      uint8_t length = spi_putc(0xff) & 0x0f;
      p_message->length = length;

      // Leer datos
      for (uint8_t i=0;i<length;i++) {
      p_message->data[i] = spi_putc(0xff);
      }

      PORT_CS |= (1<<P_CS);

      if (bit_is_set(status,3))
      {
            p_message->rtr = 1;
      }
      else {
            p_message->rtr = 0;
      }

      // Borrar la bandera de interrupción
      if (bit_is_set(status,6))
      {
            mcp2515_bit_modify(CANINTF, (1<<RX0IF), 0);
      }
      else {
            mcp2515_bit_modify(CANINTF, (1<<RX1IF), 0);
      }

      return (status & 0x07);
}
```

La función de recepción recupera, con los comandos SPI previamente descritos, la información sobre

el tipo de mensaje y el buffer en él que está el mensaje. Después, se deben leer los datos en el registro que corresponde. Aquí, las tramas con ID extendido son descartadas. (Nota del traductor: verificar si es el caso...).

La función retorna el número del filtro que permitió en ingreso del mensaje. Si no hay nuevo mensaje, la función retorna el error 0xff.

Con eso, podemos finalmente enviar y recibir mensajes y empezar a escribir el software relevante para solucionar el problema :-)

UPDATE: En la sección Downloads, hay un pequeño programa para la placa de prueba de CAN. Si el diseño usa una frecuencia distinta para el quartz, se debe ajustar en el Makefile. Si /CS y /INT se conectan a pines distintos, se debe configurar en defaults.h . Los valores para P_MOSI, P_MISO y P_SCK se deben ajustar a la implementación hardware del SPI. Este programa se debe compilar sin problema con otros AVR.

Inicializa el MCP2515, lo pasa al modo Loopback, envía un mensaje (internamente), recibe el mensaje, pasa a modo normal, reenvía el mensaje al bus CAN y espera hasta que llegue una respuesta.

Biblioteca

Se acaba de crear una biblioteca CAN que se basa en la aplicación descrita acá.

Downloads:

mcp2515_demo.zip [17.49 kB]

Comentarios

Manfred Feitzinger - 15. November 2006, 12:46:

Hola, lindo y bueno. No conozco bien GCC. Lo tienes también para Bascom, o me podrías decir como puedo escribir en los registros?

Gracias Manfred

Fabian Greif - 15. November 2006, 21:52 :

En principio, no debería ser difícil adaptar el código para Bascom. Tienes que reemplazar `spi_init()` y `spi_putc()` por su equivalente, adaptar el manejo de CS y, sino, deberías poder reusar el código sin muchos cambios. Como no uso Bascom, desgraciadamente, no puedo probar.

Aquí encontrarás unos ejemplos que otro escribió.

Frank - 2. Dezember 2006, 14:42:

Hola. Hay algún programa completo basado en este tutorial que muestre el manejo del MCP2515? O será que no vi el enlace? A propósito, buen trabajo! Saludos, Franck

Ralf Handrich - 11. Dezember 2006, 16:37:

Hola,

yo también te felicito. El circuito y el software son bárbaros!

Armé 2 módulos más. El bus CAN funciona sin problema. Sin embargo, tengo un pequeño problema con el reset del módulo. Cada vez que activo el reset (o que enciendo la placa), se envían 2 o 3 mensajes sin que yo envíe algo explícitamente. Me fijé que es una repetición del último mensaje enviado antes del reset. Los IDs parecen ser arbitrarios. Dónde está el problema? Qué puedo hacer para evitarlo?

Saludos Ralf

Maxx - 30. Dezember 2006, 01:21:

Hola todos! Primero, muchas gracias por este súper tutorial.

Mientras estaba compilando, me encontré con unas cositas. Dónde se definen SPI_RESET o SPI_CS? Me gustaría compilar mi proyecto con GCC. Tiene alguien un ejemplo funcional con Makefile que me podría proporcionar?

Gracias!

Saludos Maxx

Volker Bosch am 4. Januar 2007, 13:06 :

Hola Fabian,

muchas gracias por el tutorial sobre el control del MCP2515 – me facilitó mucho la vida para empezar con mi proyecto.

Tengo 3 comentarios sobre tu texto:

1. Escribes que el MCP2515 enviaría básicamente el byte de estado 2 veces. Como interpreto yo la especificación, envía el byte de estado una cantidad arbitraria de veces, es decir que podrías omitir la segunda lectura del byte de estado.
2. No entiendo el último ejemplo, es la misma función de transmisión de mensaje por el primero de los 3 buffers libre?
3. Según la especificación, el AVR Mega16 es muy sensible a cambios de frecuencia de clock. Atmel recomienda activar el RESET durante el cambio. No tuve problema pero, ahora, activo 1 reset por el watchdog después de cambiar la frecuencia.

Saludos, Volker.

Björn - 5. Januar 2007, 20:52:

Hola,

tu enlace hacia el #defines no funciona con la nueva página!

Saludos

Björn

Fabian Greif - 9. Januar 2007, 00:08 :

Lo corregí.

Manfred Feitzinger - 23. Februar 2007, 09:57:

Hola Fabian. Traté de portar tu programa a Bascom. Creo que la comunicación con el MCP2515 funciona pero si, por ejemplo, leo los registros después del reset, recibo valores distintos a los que escribí. Por ejemplo, escribo 0x02 en el registro DCF1 y leo 0x06. De dónde puede venir?

Fabian Greif - 23. Februar 2007, 17:50 :

Acerca de un problema de valores erróneos, verificaría primero si no viene de un exceso de velocidad en el bus SPI, ya que puede venir de un dato incorrectamente transmitido. Cómo se ve la transmisión SPI?

A propósito, para qué sirve el registro DCF1?

Manfred Feitzinger - 26. Februar 2007, 08:53:

Hola Fabian. Me equivoqué. Quería decir CNF1. Probé con una frecuencia de 50kbytes/s. El MCP2515 tiene un quartz de 16MHz, el ATMega16 corre con una frecuencia interna de 1MHz.

Stephan - 6. März 2007, 14:05 :

Hola, me gustaría manejar el MCP2515 mediante un AVR butterfly. Como no tengo mucho conocimiento en programación, quería saber si tu programa funciona también para el Butterfly.

Saludos Stephan

Fabian Greif - 6. März 2007, 14:24 :

No conozco el Butterfly pero mientras usa un núcleo AVR normal (idealmente con SPI en hardware) y tiene un mínimo de 1 a 2KB de Flash, no debería haber ningún problema para portar el código.

Frank Lorenzen - 24. Mai 2007, 22:34 :

Hola, tengo una pregunta acerca de la función mcp2515_init(). Al final, reinicializas el valor de CANCTRL con: `mcp2515_bit_modify( CANCTRL, 0xE0, 0);`

Los valores no me suenan. Según la especificación, el valor por defecto de CANCTRL es 11100000. Los parámetros no deberían ser “0xE0, 0xE0” o “0xE0, 0xFF”? Creo que sabes lo que quiero decir. Me podrías explicar?

Saludos Frank

Fabian Greif - 25. Mai 2007, 11:42:

Quiero dejar el MCP2515 en modo normal. Para eso, se deben borrar todos los bits REQOPn. Es precisamente lo que alcanzo con este comando ;-)

Es verdad que el valor por defecto de REQOP2..0 es 111. Sin embargo, la especificación dice que este estado es inválido y no se debe usar en una aplicación (ver especificación p.56)

Maximilian - 2. Juli 2007, 13:09 :

Quizás podrías dar una pequeña descripción de cómo se conectan, por ejemplo, 3 nodos y cómo se filtra el tráfico?

Fabian Greif - 6. Juli 2007, 17:39 :

...
Cada placa representa un nodo. Lo único que tienes que hacer es conectar respectivamente CANH, CANL y GND de cada nodo y terminar el bus en cada punta con una resistencia de 120 ohms.

Un ejemplo de control multinodo se encuentra aquí.

Fabian Greif - 10. Juli 2007, 15:41 :

> Entonces, por lo que veo, sólo se puede emitir o recibir.

No, se puede emitir y recibir (nota del traductor: pero en half-duplex).

Medalist - 11. August 2007, 02:43 :

Hola,

tu página es súper buena, armé 2 placas, un controlador envía y el otro recibe y funciona súper bien (125kbit/s).

Ahora mi problema:

Quisiera manejar el controlador con 100kbit/s para conectarlo al “komfortbus” de VW/Audi/Skoda/Seat. Sin embargo, tengo problemas para configurar el registro de timings. Me podrías enviar los valores a programar?

Fabian Greif - 11. August 2007, 17:49 :

El calculador en línea de www.kvaser.com indica que para 100 kBit/s con un punto de muestra (“sample“???) de 75%, necesitas lo siguiente:

```
CNF1 = 0x09
CNF2 = 0x91
CNF3 = 0x01
```

Confírmame si funciona. (Nota del traductor: OK)

J. Harms - 11. November 2007, 19:32 :

Tu artículo me ha ayudado harto. Muchas gracias.

Mientras probaba, me encontré con un error: al principio de la función MCP2515SendMessage(), lees el registro de estado para ver qué buffer está libre:

x_status = MCP2515ReadStatus (SPI_RX_STATUS) ;

Es incorrecto, debería ser:

x_status = MCP2515ReadStatus (SPI_READ_STATUS) ;

(SPI_RX_STATUS, dirección 0xB0, entrega el estado del “Receive Buffer”, y no él del “Transmit Buffer” - para eso, se debe usar SPI_READ_STATUS, dirección 0xA0).

En el ejemplo simple tipo “Ping-pong” de la aplicación de demo, no afecta pero si se arma una bucle y se envía una secuencia de mensajes (de tal forma que, a veces, Buffer #1 und #2 se llegan a usar), se tranca la aplicación hasta que el error desaparezca. Me molestó cuando empecé a trabajar con interrupciones en tiempo real.

Pregunta: alguien tiene experiencia con manejo de errores? CAN ayuda mucho en eso, por ejemplo en corrección con suma de control. Hasta ahora, no hice más – solamente una acción rápida para reconocer y botar mensajes que no se dirigen a un receptor válido – sino bloquean el buffer de transmisión. La documentación acerca de este tema es insuficiente.

Martin - 21. November 2007, 20:34 :

Pequeño error?

Me di cuenta de que en mcp2515_defs.h, la definición del registro EFLG TXB0 debería ser TXBO. Sino, te felicito por este artículo!

Traducido por : Olivier Gautherot

Ricardo Albarracin B. ralbab@gmail.com