MikroElektronika
DEVELOPMENT TOOLS | COMPILERS | BOOKS

# Muy bien.

## Lo que necesitamos ahora es un ... bus CAN

Sistemas de desarrollo EasydsPIC4A y módulos CAN-SPI

A menudo es necesario disponer de varios microcontroladores realizando diferentes operaciones, integrados en un sistema, haciendo que todos funcionen como uno solo. En este caso vamos a mostrar cómo conectar tres microcontroladores a un bus CAN y cómo usar los filtros en los nodos CAN para conseguir hacer un filtrado de mensajes.

Por Zoran Ristic
Departamento de Software – MikroElektronika

En los casos en los que varias unidades periféricas comparten el mismo bus de datos, es necesario definir cómo acceder a dicho bus. El protocolo CAN describe, de manera precisa, todos los detalles de cómo conectar varios dispositivos a una red y ésta es la forma en la que se usa ampliamente en la industria. El protocolo define principalmente la prioridad de la implementación del bus y soluciona el problema de "colisión" dentro del circuito, en el caso de que varios periféricos inicien su comunicación al mismo tiempo.

## El Circuito

En este ejemplo, un bus CAN será configurado de manera que el primer dispositivo envíe mensajes que consisten de 0x10 y 0x11, como su ID, mientras que el segundo y el tercer dispositivo envían mensajes que consisten de los IDs 0x12 y 0x13, respectivamente. También vamos a configurar los nodos CAN de manera que el segundo nodo responda a los mensajes entrantes que contengan solo el ID 0x10, mientras que el tercero sólo responderá a aquellos mensajes que contengan el ID 0x11. Por consiguiente, el primer dispositivo está configurado para recibir mensajes que contengan un ID 0x12 y 0x13 (ver Figura 2). El filtrado de mensajes se implementa fácilmente llamando a la rutina CANSPISetFilter, la cual también se encargará de manejar todas las configuraciones necesarias de los registros del microcontrolador y de la placa CAN SPI. En general, el protocolo CAN no requiere que haya un dispositivo Maestro presente en el bus. Sin embargo, para conseguir que este ejemplo sea más fácil de entender, al mismo tiempo que nos sirve de propósito general, sólo vamos a configurar el primer dispositivo para iniciar la comunicación en la red y los otros dos dispositivos para responder a las llamadas individuales.

## El Programa

Cuando se envía un mensaje, el nodo Maestro deja transcurrir suficiente tiempo para que el nodo llamado responda. En el caso en que un nodo remoto no responda dentro del tiempo requerido, el dispositivo Maestro informa de un error en el mensaje actual y procede con la llamada a otros nodos (ver Figura 3). En el caso de que un nodo CAN periférico responda al mismo tiempo que otro nodo, se producirá una "colisión" en el bus CAN. Sin embargo, la dirección prioritaria del dispositivo y el propio bus CAN establecen que, en este caso, el nodo que transmite el mensaje con la prioridad más baja lo retira del bus, lo que permite que el nodo que transmite el mensaje con la prioridad más alta proceda con su transmisión de manera inmediata. Como hemos mencionado anteriormente, usaremos un módulo SPI interno del microcontrolador para transferir datos sobre el bus CAN. Algunas de las ventajas de usar el modulo SPI interno del microcontrolador son: la posibilidad de generar una interrupción cuando se envía o se recibe datos; el módulo SPI funciona independientemente de otros periféricos y dispone de

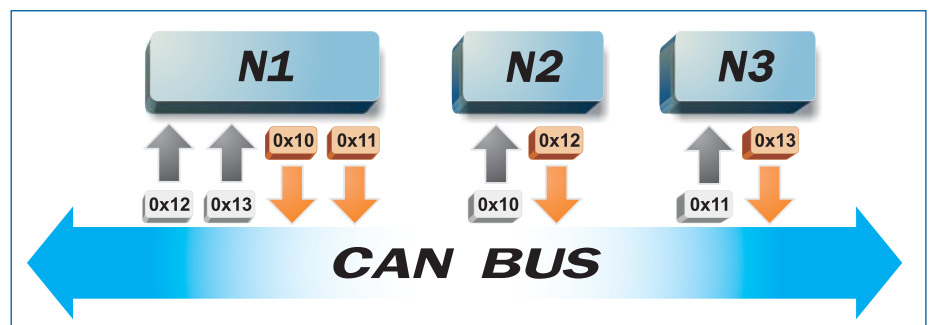
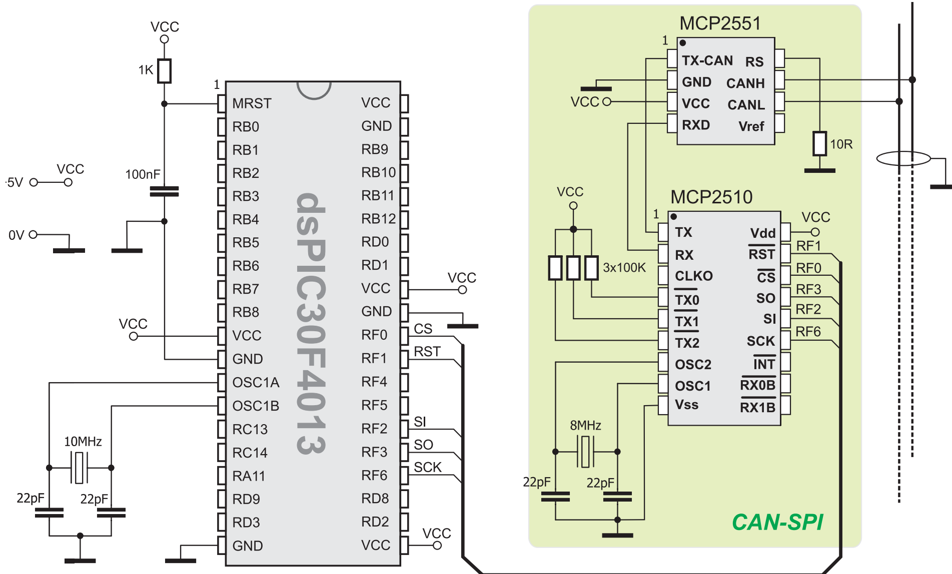



Figura 1. Filtrado de mensajes

Artículo publicitario de MikroElektronika www.mikroe.com.
mikroPASCAL® y mikroPASCAL PRO® son marcas registradas de MikroElektronika. Reservados todos los derechos.




Esquema eléctrico 1. Conexión del módulo CAN-SPI al dsPIC30F4013

una configuración simple. La librería CAN SPI nos permite configurar el modo de funcionamiento del bus CAN y los filtros del nodo, leer datos desde el "buffer" de la placa CAN SPI, etc.

Este ejemplo también incluye diodos LED en los terminales del microcontrolador que indican que la red funciona adecuadamente. Cuando el nodo 2 responde a la llamada del nodo 1, los LED del PORTB se encenderán. El código fuente para los tres nodos de la red se proporciona con este ejemplo. Para poder crear un fichero HEX para cada uno de estos nodos, de manera individual, es necesario escribir tan sólo una directiva DEFINE en la cabecera del ejemplo.

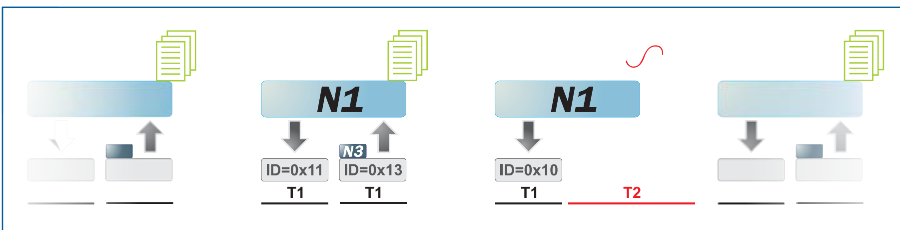
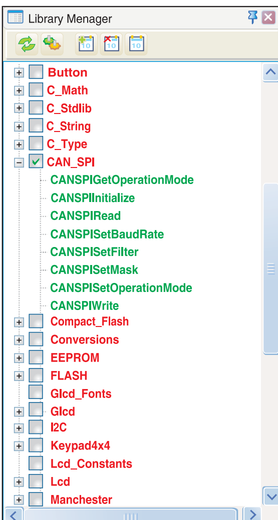



Figura 2. Ejemplo de comunicación

En resumen, hemos descrito una manera de conectar microcontroladores al bus CAN. También hemos descrito como detectar errores por medio de un protocolo de comunicación, en el caso en que un nodo remoto no responda como se esperaba, como filtrar mensajes usando los filtros CAN, además de cómo establecer comunicación, de manera general, sobre el bus CAN.

***Editor de librerías mikroPASCAL for dsPIC® con librerías, listas para ser usadas, como: CAN_SPI, GLCD, Ethernet etc.***




## Funciones usadas en el programa

| Función | Descripción |
|---|---|
| CANSPIGetOperationMode() | Modo de operación actual |
| CANSPIInitialize()* | Inicializa el módulo CANSPI |
| CANIRead()* | Lee el mensaje |
| CANSPISetBaudRate() | Establece la velocidad del CANSPI |
| CANSPISetFilter()* | Configura el filtro de mensajes |
| CANSPISetMask()* | Configuración de filtrado avanzada |
| CANSPISet OperationMode()* | Modo de operación actual |
| CANSPIWrite()* | Escribe el mensaje |

*** Funciones de la librería CANSPI usadas en el programa**

**Otras funciones mikroPASCAL for dsPICs usadas en el programa:**
**Delay_us()**
**SPI1_init()**
**SPI1_read()**

GO TO El código para este ejemplo escrito para microcontroladores dsPIC® en C, Basic y Pascal, así como los programas escritos para microcontroladores PIC®, 8051® y AVR® los pueden encontrar en nuestra página web: www.mikroe.com/en/article/.

### Programa para demostrar el funcionamiento de una red CAN

```
 program CanSpi;
{ Description:    This program demonstrates how to make a CAN
network using mikroElektronika
                  CANSPI boards and mikroPascal compiler for
dsPIC.
  Target device:  Microchip dsPIC 30F4013
  Oscillator:     10MHz crystal }

var Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags : byte;
                    // can flags
    Rx_Data_Len : byte;
                    // received data length in bytes
    RxTx_Data   : array[8] of byte;
                    // can rx/tx data buffer
    Msg_Rcvd : byte;
                    // reception flag
    Tx_ID, Rx_ID : longint;
                    // can rx and tx ID
    ErrorCount: byte;

begin
  ADPCFG := 0xFFFF;
                    // Configure analog pins as digital I/O
  PORTB  := 0; TRISB  := 0;
                    // Initialize ports
  PORTD  := 0; TRISD  := 0;
  TRISF  := 0;
  ErrorCount      := 0;
                    // Error flag
  Can_Init_Flags := 0; Can_Send_Flags := 0; Can_Rcv_Flags := 0;
                    // clear flags

  Can_Send_Flags := CANSPI_TX_PRIORITY_0 and
                    // form value to be used
                        CANSPI_TX_XTD_FRAME and
                    // with CANSPIWrite
                        CANSPI_TX_NO_RTR_FRAME;

  Can_Init_Flags := CANSPI_CONFIG_SAMPLE_THRICE and
                    // form value to be used
                        CANSPI_CONFIG_PHSEG2_PRG_ON and
                    // with CANSPIInit
                        CANSPI_CONFIG_XTD_MSG and
                        CANSPI_CONFIG_DBL_BUFFER_ON and
                        CANSPI_CONFIG_VALID_XTD_MSG;

  SPI1_Init();

  CANSPI1Init(1,3,3,3,1,Can_Init_Flags, PORTF, 1, PORTF, 0);
                    // Initialize external CANSPI module

  CANSPI1SetOperationMode(CANSPI_MODE_CONFIG, TRUE);
                    // set CONFIGURATION mode
  CANSPI1SetMask(CANSPI_MASK_B1, -1, CANSPI_CONFIG_XTD_MSG);
                    // set all mask1 bits to ones
  CANSPI1SetMask(CANSPI_MASK_B2, -1, CANSPI_CONFIG_XTD_MSG);
                    // set all mask2 bits to ones

  CANSPI1SetFilter(CANSPI_FILTER_B2_F4, 0x12, CANSPI_CONFIG_
XTD_MSG);         // Node1 accepts messages with ID 0x12
  CANSPI1SetFilter(CANSPI_FILTER_B1_F1, 0x13, CANSPI_CONFIG_
XTD_MSG);         // Node1 accepts messages with ID 0x13

  CANSPI1SetOperationMode(CANSPI_MODE_NORMAL,0xFF);
                    // set NORMAL mode
  RxTx_Data[0] := 0x40;
                    // set initial data to be sent

  Tx_ID := 0x10;
                    // set transmit ID for CAN message

  CANSPI1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags);
                    // Node1 sends initial message

  while (TRUE) do
                    // endless loop
    begin
      Msg_Rcvd := CANSPI1Read(Rx_ID, RxTx_Data, Rx_Data_Len,
Can_Rcv_Flags); // attempt receive message
      if (Msg_Rcvd) then begin
                    // if message is received then check id

        if Rx_ID = 0x12 then
                    // check ID
            PORTB := RxTx_Data[0]
                    // output data at PORTB
        else
            PORTD := RxTx_Data[0];
                    // output data at PORTD
        delay_ms(50);
                    // wait for a while between messages
        CANSPI1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags);
                    // send one byte of data
        inc(Tx_ID);
                    // switch to next message
        if Tx_ID > 0x11 then Tx_ID := 0x10;
                    // check overflow

       end
      else begin
                    // an error occured, wait for a while

        inc(ErrorCount);
                    // increment error indicator
        Delay_ms(10);
                    // wait for 10ms
        if (ErrorCount > 10) then begin
                    // timeout expired - process errors
          ErrorCount := 0;
                    // reset error counter
          inc(Tx_ID);
                    // switch to another message
          if Tx_ID > 0x11 then Tx_ID := 0x10;
                    // check overflow
          CANSPI1Write(Tx_ID, RxTx_Data, 1, Can_Send_
Flags);             // send new message
        end;

       end;
    end;
end.
```

Microchip®, el logotipo y combinaciones de los mismos, dsPIC® y otros, son marcas registradas o marcas de Microchip Corporation o sus subsidiarias. Otros términos y nombres de productos pueden ser marcas de otras compañías.