```
/*****************************************************************************
   Midi.c

   These are the functions that provide the interface to the midi port

*****************************************************************************/

#include <io8515v.h>
#include <macros.h>
#include "smb.h"
#include "midi.h"

#pragma interrupt_handler iTxd:11
#pragma interrupt_handler iRxd:iv_UART_RX
#pragma interrupt_handler iTimer:iv_TIMER0_OVF

char GateShadow;

typedef struct {
   volatile int head;
   volatile int tail;
   volatile int nchar;
   volatile int size;
}IODESC;

volatile IODESC txdesc;
char txbuff[32];
volatile IODESC rxdesc;
char rxbuff[32];

void InitUart(int baud)
{
   int i;

   UCR |= 0x18;   /* enable uart rec and xmit   */
   UBRR = baud;
   txdesc.head = 0;  /* initialize transmit descriptor   */
   txdesc.tail = 0;
   txdesc.nchar = 0;
   txdesc.size = 32;
   rxdesc.head = 0;  /* initialize receiever descriptor  */
   rxdesc.tail = 0;
   rxdesc.size = 32;
   rxdesc.nchar = 0;
   for(i=0;i<32;++i)
   {
      txbuff[i] = 'o';
      rxbuff[i] = 'i';
   }
```

```
}

void Disable(void)      /* disable interrupts   */
{
   SREG &= ~0x80;
}

void Enable(void)    /* enable interrupts */
{
   SREG |= 0x80;
}

void EnableRxIRQ(void)
{
   /*************************
   ** Enable Receiver interrupts
   *************************/
   UCR |= 0x80;
}

void DisableRxIRQ(void)
{
   /*************************
   ** Disable Receirver interrupts
   *************************/
   UCR &= ~0x80;
}

void EnableTxIRQ(void)
{
   UCR |= 0x20;   /* enable transmit data register empty interrupt   */
}

void DisableTxIRQ(void)
{
   UCR &= ~0x20;  /* disable transmit data register empty interrupt  */
}

void iRxd(void)
{
   /*************************
   ** Interrupt handler for
   ** Receive Interrupts
   *************************/
   char c = UDR;  /* read data from receive data reg  */
   if(rxdesc.nchar < rxdesc.size)   /* is there space to put char?   */
   {
      rxbuff[rxdesc.head++] = c; /* put character into buffer  */
      if(rxdesc.head == rxdesc.size) rxdesc.head = 0; /* wrap head pointer */
```

```
      rxdesc.nchar++;   /* increment character count  */
   }  /* otherwise, just drop character   */
}

void iTxd(void)      /* interrupt handler for uart */
{
   char c;

   c = (char)txbuff[txdesc.tail++];    /* get character from buffer  */
   --txdesc.nchar;                  /* one less character to send */
   if(txdesc.nchar == 0)            /* is xmit buffer empty?   */
      DisableTxIRQ();               /* stop transmitting data  */
   if(txdesc.tail == txdesc.size) txdesc.tail = 0; /* wrap tail pointer */
   UDR = c;
}

int GetC(void)
{
   /*******************************
   ** Get a character from the UART
   ** If there is no character, wait
   *******************************/
   int c;

   while(rxdesc.nchar == 0);  /* wait for character to apear in buffer  */
   Disable();        /* disable interrupts   */
   c = (int)rxbuff[rxdesc.tail++];     /* get character from buffer  */
   if(rxdesc.tail == rxdesc.size) rxdesc.tail = 0; /* wrap tail pointer */
   rxdesc.nchar--;   /* decrement number of chars in buffer */
   Enable();
   return c;   /* return fetched character   */
}

void PutC(int i)
{
   while(txdesc.nchar == txdesc.size); /* pend on buffer full  */
   Disable();  /* Disable interrupts   */
   txbuff[txdesc.head++] = (char)i; /* put data into buffer */
   if(txdesc.head == txdesc.size) txdesc.head = 0; /* wrap head pointer */
   if(txdesc.nchar == 0)   /* first char in buff?  */
      EnableTxIRQ();    /* start transmit interrupt   */
   txdesc.nchar++;
   Enable();   /* enable interrupts */
}

//---------------------------------------------------------------------------
// Timer STuff...
//---------------------------------------------------------------------------
```

```
unsigned char Counter;
unsigned char CRV;

void iTimer(void)
{
   //---------------------------------
   //timer interrupt handler
   //---------------------------------
   TCNT0 = ~31;
   if(--Counter == 0)
   {
      GateShadow ^= 0x02;
      LED_PORT = ~GateShadow;
      GATE_PORT = GateShadow;
      Counter = CRV;
   }
}

void EnableTimerIrq(void)
{
   TIMSK |= 0x02; //enable timer interrupt
}

void DisableTimerIrq(void)
{
   TIMSK &= ~0x02;   //disable timer interrupt
}

void InitTimer(void)
{
   TCCR0 = 5;  //system clock divided by 1024
   TCNT0 = ~31;
   Counter = 31;
   CRV = 31;
}


void Delay()
{
   unsigned char a, b;

   for (a = 1; a; a++)
      for (b = 1; b; b++);
}

enum {IDLE,NOTE0,NOTE1};

#define SYSEX        0xf0
#define SYSCOMUNDEF     0xf1
```

```
#define SYSCOMSONGPOS   0xf2
#define SYSCOMSONGSEL   0xf3
#define SYSCOMUNDEF1 0xf4
#define SYSCOMUNDEF2 0xf5
#define SYSCOMTUNEREQ   0xf6
#define SYSEXEND     0xf7
#define SYSRTCLOCK      0xf8
#define SYSRTUNDEF      0xf9
#define SYSRTSTART      0xfa
#define SYSRTCONTINUE   0xfb
#define SYSRTSTOP    0xfc
#define SYSRTUNDEF1     0xfd
#define SYSRTACTIVESEN  0xfe
#define SYSRTRESET      0xff

#define NOTEON 0x90
#define NOTEOFF   0x80
#define NOTEPRES  0xa0     //note pressure
#define CHANPRES  0xb0     //channel pressure
#define CONTROL   0xd0        //control change
#define WHEEL  0xe0        //pitch wheel change
#define PATCH  0xc0        //patch change


#define MIDI_CLOCK   0x80
#define MIDI_START   0x40
#define MIDI_RESET   0x20
#define MIDI_GATE 0x01

main()
{
   int currentnote,nextnote;
   char c;
   char cmd;
   char chan;
   char state;
   int v;

   InitSPI();     /* initialize the SPI port */
   InitTimer();   //initialize real clock timer
   InitUart(BAUD_MIDI);
   EnableRxIRQ(); /* enable receive interrupt   */
   EnableTimerIrq();
   GateShadow = 0;
   GATE_PORT = GateShadow; /* set all outputs to zero */
   LED_PORT = ~GateShadow;
   Enable();      /* enable global interrupts   */

   while(1)
```

```
{
   c = GetC();
   if(c & 0x080)  //is it a command?
   {
      if(c < 0x0f0)
      {
         //process this stuff, dispose of everything else
         cmd = (char)(c & 0xf0);
         chan = (char)(c & 0x0f);
         if(cmd == NOTEON)
            state = NOTE0;
      }
      else
      {
         switch(c)
         {
            case SYSEX:
               break;
            case SYSCOMUNDEF:
               break;
            case SYSCOMSONGPOS:
               break;
            case SYSCOMSONGSEL:
               break;
            case SYSCOMUNDEF1:
               break;
            case SYSCOMUNDEF2:
               break;
            case SYSCOMTUNEREQ:
               break;
            case SYSEXEND:
               break;
            case SYSRTCLOCK:
               Disable();
               GateShadow ^= MIDI_CLOCK;  /* toggle these bits */
               GATE_PORT = GateShadow;
               LED_PORT = ~GateShadow;
               Enable();
               break;
            case SYSRTUNDEF:
               break;
            case SYSRTSTART:
               Disable();
               GateShadow |= MIDI_START;
               GateShadow &= ~MIDI_RESET;
               GATE_PORT = GateShadow;
               LED_PORT = ~GateShadow;
               Enable();
               break;
```

```
            case SYSRTCONTINUE:
               break;
            case SYSRTSTOP:
               Disable();
               GateShadow &= ~MIDI_START;
               GATE_PORT = GateShadow;
               LED_PORT = ~GateShadow;
               Enable();
               break;
            case SYSRTUNDEF1:
               break;
            case SYSRTACTIVESEN:
               break;
            case SYSRTRESET:
               Disable();
               GateShadow &= ~MIDI_START;
               GateShadow |= MIDI_RESET;
               GATE_PORT = GateShadow;
               LED_PORT = ~GateShadow;
               Enable();
               break;
         }  //end of swtich(c)
      }  //end of if(c < 0xf0)

   }
   else  // if(c & 0x080)
   {
      switch(cmd)
      {
         case NOTEOFF:
            switch(state)
            {
               case NOTE0:
                  nextnote = (int)c << 5;
                  state = NOTE1;
                  break;
               case NOTE1:
                  if(nextnote == currentnote)
                  {
                     Disable();
                     GateShadow &= ~MIDI_GATE;
                     GATE_PORT = GateShadow;
                     LED_PORT = ~GateShadow;
                     Enable();
                  }
                  state = NOTE0;
                  break;
            }
            break;
```

```
            case NOTEON:
               switch(state)
               {
                  case NOTE0:
                     nextnote = (int)c << 5;
                     state = NOTE1;
                     break;
                  case NOTE1:
                     Disable();
                     if(c > 0)
                     {
                        currentnote = nextnote;
                        SendData(currentnote,0);
                        if(GateShadow & MIDI_GATE) //is gate already set?
                        {
                           GateShadow &= ~MIDI_GATE;  //retrigger gate
                           GATE_PORT = GateShadow;
                           LED_PORT = ~GateShadow;
                        }
                        GateShadow |= MIDI_GATE;
                     }
                     else
                     {
                        if(currentnote == nextnote)
                        {
                           GateShadow &= ~MIDI_GATE;
                        }
                     }
                     GATE_PORT = GateShadow;
                     LED_PORT = ~GateShadow;
                     Enable();
                     state = NOTE0;
                     break;
                  default:
                     break;
               }
               break;
            case NOTEPRES: //note pressure
               switch(state)
               {
                  case NOTE0:
                     state = NOTE1;
                     break;
                  case NOTE1:
                     state = NOTE0;
                     break;
               }
               break;
            case CHANPRES:    //channel pressure
```

```
            break;
         case CONTROL:        //control change
            switch(state)
            {
               case NOTE0:
                  state = NOTE1;
                  break;
               case NOTE1:
                  state = NOTE0;
                  break;
            }
            break;
         case WHEEL:       //pitch wheel change
            switch(state)
            {
               case NOTE0:
                  v = ((int)c);
                  state = NOTE1;
                  break;
               case NOTE1:
                  v |= ((int)c)<<7;
                  v >>= 2; //12 bits;
      //          v ^= 0x0800;
                  SendData(v,1);
                  state = NOTE0;
                  break;
            }
            break;
         case PATCH:       //patch change
            state = NOTE0;
            break;
      }  //end of switch(cmd)
   }  //end of if(c & 0x080)

}  //end of while(1) loop
return 0;
}
```

```
/*************************************************************
   Midi.H

   This is the header file for the MIDI driver interface

   Open(void)     initializes midi port
   GetC(void)     gets character from midi buffer
   PutC(int c) puts a character to midi buffer
   StatusIn(void) gets number of chars in buffer
   StatusOut(void)   gets number of chars left to transmit

   This code is copyright(c) 2002 by Jim Patchell
   However, anyone is free to use this as they see fit, as long
   as this text remains in the source code
*************************************************************************/

#ifndef MIDI__H
#define MIDI__H

extern void InitUart(int baud);  /* initialize UART      */
extern void Disable(void);    /* disable interrupts   */
extern void Enable(void);     /* enable interrupts */
extern void EnableRxIRQ(void);   /* enable reciever interrupt  */
extern void DisableRxIRQ(void);  /* disable reciever interrupt */
extern void EnableTxIRQ(void);   /* enable transmitter interrupt  */
extern void DisableTxIRQ(void);  /* disable transmitter interrupt */
extern int GetC(void);        /* get character from midi port in  */
extern void PutC(int i);      /* send character to midi port out  */


#endif
```

```
/*****************************************************************************
**
** smb.h
**
** This is the header file for the Synth Module Board AVR controller
**
*****************************************************************************/

#ifndef SMB__H
#define SMB__H

/************************************
** hardware defines
************************************/

#define GATE_PORT (*((volatile char *)0xc000))
#define LED_PORT  (*((volatile char *)0xc400))
#define IN_PORT   (*((volatile char *)0xc400))
#define DAC0_LOAD (*((volatile char *)0xa000))
#define DAC1_LOAD (*((volatile char *)0xa400))
#define DAC2_LOAD (*((volatile char *)0xa800))
#define DAC3_LOAD (*((volatile char *)0xac00))
#define DAC4_LOAD (*((volatile char *)0xb000))
#define DAC5_LOAD (*((volatile char *)0xb400))
#define DAC6_LOAD (*((volatile char *)0xb800))
#define DAC7_LOAD (*((volatile char *)0xbc00))

#define BAUD_9600 51
#define BAUD_MIDI 15

/************************************
** Defines for SPI Port
************************************/

#define SPI_SPCR_SPIE   0x80  /* interrupt enable  */
#define SPI_SPCR_SPE 0x40  /* SPI port enable      */
#define SPI_SPCR_DORD_LSB  0x20  /* SPI Data Order    */
#define  SPI_SPCR_MSTR  0x10     /* SPI Master  */
#define SPI_SPCR_CPOL   0x08     /* clock polarity */
#define SPI_SPCR_CPHA   0x04     /* clock phase    */
#define SPI_SPCR_RATE1  0x02     /* clock rate  */
#define SPI_SPCR_RATE0  0x01     /* clock rate  */

#define SPI_SPSR_SPIF   0x80  /* Interrupt Flag */
#define SPI_SPSR_WCOL   0x40  /* write colision flag  */

/* SPI Port access functions  */

extern void InitSPI(void);
```

```
extern int PortFull(void);
extern void SendData(int d,int port);


#endif
```

```
/*************************************************************************
**
** These are the routines for talking to the SPI port
**
** The SPI port will output a 16 bit integer to the DAC
**  It will then latch the data into the appropriate DAC register
**
*************************************************************************/
#include <io8515v.h>
#include <macros.h>
#include "smb.h"

/*************************************************************************
**
** This routine initializes the SPI port and gets it ready for use
**
*************************************************************************/
void InitSPI(void)
{
   DDRB |= 0xb0;  /* set SS pin to output */
   SPCR = SPI_SPCR_SPE | SPI_SPCR_CPHA | SPI_SPCR_MSTR;  /* enable SPI port   */
   MCUCR |= 0x80; /* enable external memory  */
}

/*************************************************************************
** Wait for data to exit from SPI port
** Returns true while data is still transmiting
*************************************************************************/

int PortFull(void)
{
   char a;
   int retval=0;
   a = SPSR;
   if(a & 0x80) retval = 1;
   return retval;
}

/*************************************************************************
This routine sends data d to DAC port port
*************************************************************************/
static const int DacLUT[8] = {
   0xa000,
   0xa400,
   0xa800,
   0xac00,
   0xb000,
   0xb400,
   0xb800,
```

```
   0xbc00
};


void SendData(int d,int port)
{
   union {
      int v;
      char b[2];
   }convert;
   convert.v = d;
   SPDR = convert.b[1]; /* get MSB of data   */
   while(!PortFull());        /* wait for data to be transmitted  */
   SPDR = convert.b[0];       /* get LSB of data   */
   while(!PortFull());     /* wait for data to be transmitted  */
   *((volatile  char *)DacLUT[port]) = 0;  //dummy write to load DAC  */
}
```