# 5

# Serial Interface Subsystems

This chapter will focus on serial communications interfaces because these form an important aspect of many embedded system designs. In particular industry standard communication protocols will be described together with the design of supporting software. Examples will feature RS232 Communications, inter-integrated circuit (I2C) and serial peripheral interface (SPI) protocols and the universal serial bus (USB). Although C code will be employed for the examples, particular aspects of the linked assembler code will be explained where this is significant for enhanced performance.

## 5.1 Introduction

Serial interfaces are required when the complexity and cost of a parallel interface cannot be justified such as where the communication distance is more than a few meters. In a serial interface the data bits are assembled into a serial pattern for transmission along a single wire to their destination. In order to recover the original data a synchronised clock is required in the receiver to extract the bits from the serial stream as they arrive. Some serial interfaces use a supplementary dedicated connection for clock, whereas others require the receiver to regenerate the clock locally, obviating the need for any extra connections.

*Digital Interface Design and Application*, First Edition. Jonathan A. Dell.

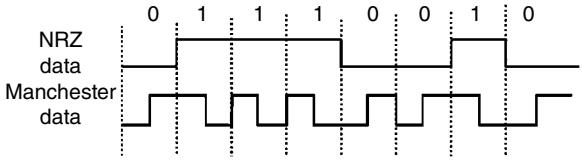


**Figure 5.1** Manchester code

For short distance serial interfaces a clock connection can be accommodated because the cost will not be significant but for longer distance communications a local clock regeneration arrangement is the only feasible alternative. Some serial systems employ an encoding process where the data is delivered in a form that will enable self-clocking to be established, an example of this is the Manchester code. In essence this uses a high to low transition at the bit time centre to encode a one and a high to low transition at the bit centre to encode a zero. It can be observed from Figure 5.1 that this requires many extra signal transitions and consequently a greater bandwidth in a communications channel. It is, however, used in the 10 Mbs Ethernet local area network implementation. As a reminder NRZ means non-return to zero between bits.

## 5.2 RS232 Universal Asynchronous Receiver/Transmitter (UART) Communications

Historically, RS232 was used to interconnect computers and distant terminals and was the first serial protocol to be widely accepted throughout industry for these longer distance communications. It uses a single wire pair; that is signal and ground and delivers reliable communication at low cost. To solve the clock regeneration problem this scheme uses a local clock that is triggered by the falling edge at the start of the signal group and synchronism is assumed to be accurate enough over a limited group of following bits. Before the block of 8 data bits is transmitted the serial line is held high so that the start of data is clearly defined. The serial signal format including the essential stop bit to return the line to the high state is shown in Figure 5.2. The special hardware module designed for this function is known as a universal asynchronous receiver transmitter (UART) and this includes the clock regeneration. In practice the clock regeneration is usually based on a measurement of the start-bit length and then the following bits are accepted at the approximate centre of their bit position, thus avoiding the inevitable distortion of the signal edges that accumulate over a lengthy connection path.

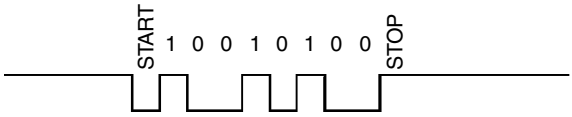


**Figure 5.2** RS232 data format

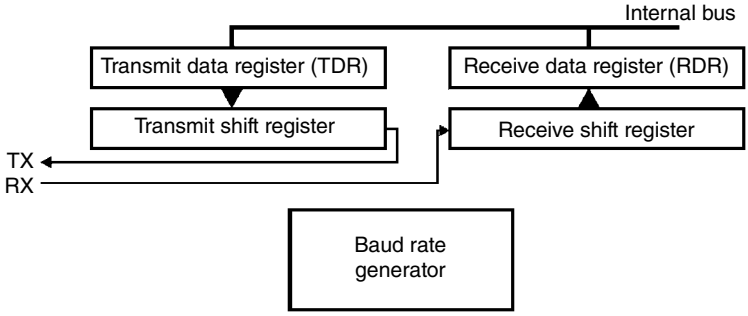


**Figure 5.3** UART module

The STM32F4 includes six blocks that can implement UART functions but four of them can also implement synchronous communications where a controlling clock signal is generated, so they are actually called universal synchronous/asynchronous receiver/transmitter (USART) modules [1].

The diagram in Figure 5.3 shows the main blocks of a typical UART module. The baud rate generator provides the bit timing clocks for transmitter and receiver shift registers. A typical rate of 9600 bps will require a transmitter clock period of 1/9600 almost exactly 104 µs, this is usually derived from the internal system clock and the UART reference manual provides values for the BRG (Baud Rate Generator) registers to achieve a variety of standard rates [1].

In the default mode the equation for the baud rate divider (BRDIV) is $BRDIV = \frac{fck}{8 \times 2 \times BRATE}$ when $f_{ck}$ is set to 8 MHz the divider becomes 52.0625. In many designs the fractional part is ignored resulting in a slight but insignificant error. In the STM32F4 USART implementation the fractional part of the divider is set in the least significant 4 bits $(0.0625 \times 16) = 1$ so the resulting value of baud rate register (USARTDIV) is 0x0341 [1]. This provides a considerable improvement in clock accuracy. The STM32F4 interface utilities actually perform this calculation for the user so only the actual baud rate required is submitted in the initialisation function call.

On the STM32F4DIS-BB base board USART6 is available and a special higher voltage buffer SP3232EEY to implement the standard ±12*V* signal is included for the 9-pin D connector COM1. Code to connect the pins and initiate USART6 is shown next. Note the USART nomenclature used on the STM32F4 to emphasise the additional synchronous option of this module [1].

```
void GPIO_set_ioports(void)
{
   /* USART6 uses Alternate Function 8 */
   /* TX is on GPIOC Pin6 and RX is on GPIOC Pin7 */

   GPIO_InitTypeDef GPIO_InitStructure;

   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

   /* TX output and RX input*/
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

   GPIO_Init(GPIOC, &GPIO_InitStructure);

   /* select Alternate functions (GPIOC->AFR[0] =
0x88000000;)*/
   GPIO_PinAFConfig(GPIOC, GPIO_Pin_6, GPIO_AF_USART6);
   GPIO_PinAFConfig(GPIOC, GPIO_Pin_7, GPIO_AF_USART6);
}

void USART6_setup(void)
{
   USART_InitTypeDef  USART_InitStruct;

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART6,
ENABLE);

   USART_InitStruct.USART_BaudRate = 9600;
   USART_InitStruct.USART_WordLength = USART_
WordLength_8b;
   USART_InitStruct.USART_StopBits = USART_StopBits_1;
   USART_InitStruct.USART_Parity = USART_Parity_No;
   USART_InitStruct.USART_Mode = USART_Mode_Rx |
USART_Mode_Tx;
```

```
    USART_InitStruct.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;      /* if using
both _RTS_CTS */

    USART_Init(USART6, &USART_InitStruct);
    USART_Cmd(USART6, ENABLE);
}
```

As the serial interface is quite slow compared to code execution the handling of transfers to the transmit data register (TXDR) and from the receive data register (RXDR) must be constructed carefully to prevent possible overruns in the case of transmission or reception not being completed. The code that follows shows how the transmit register empty (TXE) flag is tested in the case of transmission, transmission complete (TC) is also tested to safeguard the process. For reception receiver not empty (RXNE) is tested to wait for the buffer to be full, then the received data register can be accessed.

```
void char_print(char p)
{
/* outut to serial interface on UART6 */
       /* wait for TXE */
       while (USART_GetFlagStatus(USART6, USART_
FLAG_TXE) != 1)
       {
       }
       USART_SendData(USART6, p);
       while (USART_GetFlagStatus(USART6, USART_
FLAG_TC) != 1)
       {
       }
}

char get_char(void)
{
    /* input via serial input on UART6 */
    char p;
    /* wait till buffer is full */
    while (USART_GetFlagStatus(USART6, USART_FLAG_
RXNE) != 1)
    {
    }
```

```
    p = USART_ReceiveData(USART6);
    return(p & 0x7f);
}
```

These functions could quite easily be implemented in assembler code because the USART6 registers addresses and the bit test positions are well documented. However, there would not be much advantage in using this approach as the data transfer rate is quite slow in any case.

Note that when the PC Hyper-terminal is used it will usually be essential to defeat the hardware flow control, which is implemented by default. This is achieved by supplying a signal for the request to send (RTS), pin 7 on the 9-way D-type connector, to activate transmission from the keyboard. This can be obtained from the clear to send (CTS), pin 8 on the D-type, so a wire link between these pins on the back of the Discovery base board will satisfy this requirement.

## 5.3 The I2C Interface

The I2C was developed for short interconnections between components, offering a considerable economy in situations where multiple connections for data, address and control would have been used otherwise. This greatly simplifies the printed circuit board (PCB) design as it only requires two connections, apart from the common ground. This is shown in the diagram Figure 5.4.

The interfaced components have a master/slave relationship and the connections are named serial clock (SCL) and serial data (SDA). Both use an open-drain configuration with pull-up resistors so that drive can be activated

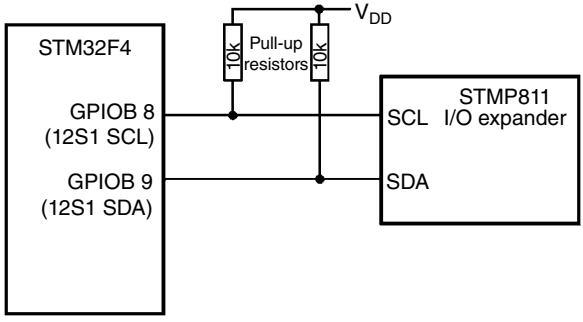


**Figure 5.4** An I2C interface circuit

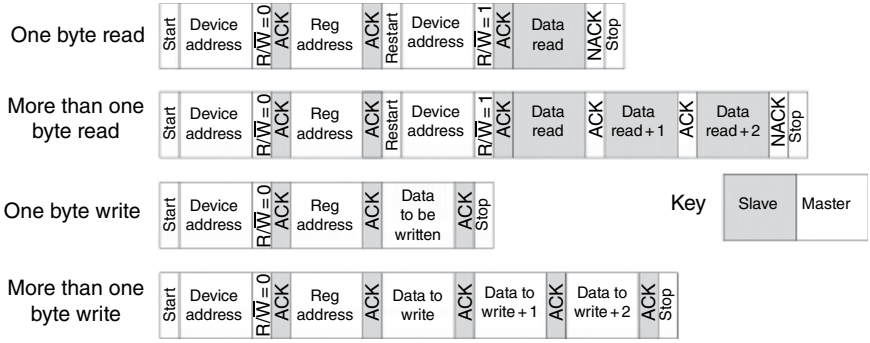


**Figure 5.5** I2C read and write modes

in either device at appropriate times. The STM32F4 has three independent I2C interface modules and two further modules that can take on the I2C role if required. Read and write data transfers across the I2C bus are defined by the diagrams shown in Figure 5.5.

The start condition is defined when SCL is high and SDA falls and the stop condition is defined when SCL is high and SDA rises. The register address is 7 bits and the eighth least significant bit (LSB) indicates whether a read or write (R/W) operation follows. The STM32F4 support utilities file stm32f4xx_i2c.c provides software resources to initialise the I2C hardware block in preparation for communication and but it is up to the user to assemble message sequences correctly in software.

The acknowledge bit (ACK) is generated by the receiving device and the master provides an extra clock cycle for this to happen. Note that a new start (RESTART) is needed to switch the direction on SDA in the read processes. To terminate the read transfers the master issues a negative acknowledgement (NACK) on the final read cycle. This ensures that the slave releases the SDA line so that the master can generate the stop condition. In the write transfers the master is in control so it can terminate the sequence by creating the stop condition.

The STM32F4 I2C interface also offers a 10-bit addressing mode for components with extended capability.

### *5.3.1 Using the Touch Screen with an I2C Interface*

A useful example of an I2C interface is set up on the STM32F4 DIS_LCD board forming an interface with the touch screen controller STMP811. This component also contains a temperature sensor so this forms the subject of this example. The code that follows shows how the I2C module connections are

set up and the initialisation is achieved. Notice that the pins are defined as open-drain (OD) so that the SCL and SDA lines can perform correctly. Also in this case the I2C clock speed is set to 100 kHz.

```
void I2C_InitPinsandI2C(void)
{
   I2C_InitTypeDef I2C_InitStructure;
   GPIO_InitTypeDef GPIO_InitStructure;

   RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

   /* set up B8 (SCL) and B9 (SDA) */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 |
GPIO_Pin_9;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

   GPIO_Init(GPIOB, &GPIO_InitStructure);

   GPIO_PinAFConfig(GPIOB, GPIO_Pin_8, GPIO_AF_I2C1);
   GPIO_PinAFConfig(GPIOB, GPIO_Pin_9, GPIO_AF_I2C1);
/* Note: the ST code module is incorrect so a
direct method is used here */
   GPIOB->AFR[1] = 0x044;

   I2C_InitStructure.I2C_ClockSpeed = 100000;
   I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
   I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
   I2C_InitStructure.I2C_OwnAddress1 = 0x00;
   I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
   I2C_InitStructure.I2C_AcknowledgedAddress =
I2C_AcknowledgedAddress_7bit;

   I2C_Init(I2C1, &I2C_InitStructure);
   I2C_AcknowledgeConfig(I2C1, ENABLE);
   I2C_Cmd(I2C1, ENABLE);
}
```

A particular sequence of actions is required to set up the STMP811 initialising and reading the various registers. It has no less than 49 internal registers

**Table 5.1** STMP811 system control register 2

| SYS_CR2 Bit # | Name | Function |
|---|---|---|
| 3 | TS_OFF | Temperature gauge off |
| 2 | GPIO_OFF | I/O expansion off |
| 1 | TSC_OFF | Touch screen controller off |
| 0 | ADC_OFF | Analogue to digital converter off |

**Table 5.2** Temperature gauge control register

| TEMP_CTRL Bit # | Name | Function |
|---|---|---|
| 4 | THRES_RANGE | 0 – greater than or equal threshold<br>1 – otherwise |
| 3 | THRES_EN | Temperature threshold enable |
| 2 | ACQ_MODE | 0 – once only<br>1 – every 10 ms |
| 1 | ACK | Acquire temperature |
| 0 | EN | Enable |

controlling its four main functions, a GPIO expander, an analogue to digital (AD) converter, a touch-screen interface and a temperature gauge. System control register two at address 0x04 can select the parts that are actually needed as detailed in Table 5.1.

Firstly the elements I/O Expander are all switched on by writing zero to register 0x04, secondly the temperature subsystem mode is selected by writing to register TEMP_CTRL at address 0x60 where the bits have the functions shown in Table 5.2.

Finally the temperature value is read from register 0x61 (MSB) and 0x62 (LSB).

```
IOE_write_reg(I2C1, 0x04, 0);       /* SYS_CR2 */
IOE_write_reg(I2C1, 0x60, 7);     /* TEMP_CTRL */

temp1 = IOE_read_reg(I2C1, 0x61);  /* MS Byte */
temp0 = IOE_read_reg(I2C1, 0x62);  /* LS Byte */
```

The code for writing to a register within the structure of the STMPE811 I/O expander is shown next in the function IOE_write_register() [2]. This requires five I2C steps as determined by the 1-byte write sequence in Figure 5.5. Each

operation must check appropriate I2C event status bits to confirm that the appropriate operations have been accomplished before the next one is attempted.

```
void IOE_write_reg(uint8_t DeviceAddr, uint8_t
RegisterAddr, uint8_t RegisterValue)
{
    /* 1 send start */
    I2C_GenerateSTART(I2C1, ENABLE);
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
MODE_SELECT) != 1)
    {}
    /* 2 send device address */
    I2C_Send7bitAddress(I2C1, DeviceAddr, I2C_
Direction_Transmitter);
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
TRANSMITTER_MODE_SELECTED) != 1)
    {}
    /* 3 send register address */
    I2C_SendData(I2C1, RegisterAddr);    /*ident
register select*/
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
BYTE_TRANSMITTED) != 1)
    {}
    /* 4 send register value */
    I2C_SendData(I2C1, RegisterValue);
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
BYTE_TRANSMITTED) != 1)
    {}
    /* 5 send stop condition */
    I2C_GenerateSTOP(I2C1, ENABLE);
}
```

For reading the contents of a register the code next shows the seven step sequence required as shown in the 1-byte read case in Figure 5.5. This starts off in the same way as the write operation but at step 4 a RESTART must be issued so that the I2C interface can be switched to a read mode. Also it will be observed that the last I2C cycle will have to generate the negative acknowledge so the configuration is set to I2C_NACKPosition_Current just before the single read operation in step 6.

```
uint8_t IOE_read_reg(uint8_t DeviceAddr, uint8_t
RegisterAddr)
   {
   uint8_t temp;

   /* 1. generate START */
   I2C_GenerateSTART(I2C1, ENABLE);
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
MODE_SELECT) != 1)
   {}
   /* 2. ADDRESS and WRITE */
   I2C_Send7bitAddress(I2C1, DeviceAddr, I2C_
Direction_Transmitter);
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
TRANSMITTER_MODE_SELECTED) != 1)
   {}
   /* 3. REG address */
   I2C_SendData(I2C1, RegisterAddr);

   /*outp register select*/
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
BYTE_TRANSMITTED) != 1)
   {}
   /* 4. RESTART */
   I2C_GenerateSTART(I2C1, ENABLE);
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
MODE_SELECT) != 1)
   {}

   /* 5. ADDRESS and READ */
   I2C_Send7bitAddress(I2C1, DeviceAddr, I2C_
Direction_Receiver);
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
RECEIVER_MODE_SELECTED) != 1)
   {}
   /* 6. READ byte, NACK follows */
   I2C_NACKPositionConfig(I2C1, I2C_NACKPosition_
Current);

   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
BYTE_RECEIVED) != 1)
   {}
```

```
    temp = I2C_ReceiveData(I2C1);

    /* 7. send STOP*/
    I2C_GenerateSTOP(I2C1, ENABLE);

    return(temp);
}
```

It will be seen that the I/O Expander can also generate an interrupt from various sources within its logic such as the touch sensor that can be useful in its overall management.

Debug of interface modules of this type, where a number of different actions have to be taken, is quite difficult to approach. It would be advantageous to include a route to exit from the while loops if an error occurs or if the procedure takes too long to complete. This will avoid the problem when the code gets stuck for no apparent reason.

If one of the timers is used and set to a period of 100 ms, for example the code shown next illustrates a possible design, here the timeout() function returns zero until a specified time has elapsed. Different return values can be used to identify the error reporting code section.

```
    if (timeout())
    {
        DEBUG_PRINT("RX Timeout error\r\n");
        return(1);
    }
```

## 5.4 SPI Interface

The SPI was developed for similar short distance applications in embedded systems as I2C but offering a considerably higher data throughput based on a clock rate of several megahertz and considerably reduced protocol overhead. One of its many possible applications is to form the basic mode communication link with an SD memory card. It forms a synchronous data link performing full duplex communication between devices having a Master/Slave relationship. The Master always initiates the communication process and also provides the clock allowing one or multiple Slave units to be accommodated.

The SPI link requires four connections, as shown in Figure 5.6, apart from the common ground, these are serial clock (SCK in the figure), master out/slave in (MOSI), master in/slave out and active low slave select ($\overline{SS}$). Each of

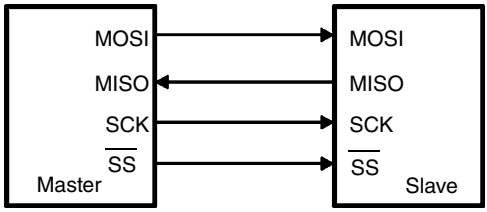


**Figure 5.6** SPI interface connections

the three available SPI interface modules on the STM32F4 implement a single $\overline{SS}$ select and this is clearly redundant in many cases if only a single slave is envisaged in the design.

The SPI interface connections are mapped through GPIOA, GPIOB and GPIOC as shown in the STM32F4 data sheet Table 8 Alternate Function Mapping [2]. Taking SPI3 the assignment is mainly on GPIOC, SCL uses pin 10, MISO (Master In/Serial Out) uses pin 11, MOSI uses pin 12 and finally the slave select is assigned to GPIOA pin 15. The interface is quite straightforward to set up as shown by the initialisation code next. The main issue to consider is the clock edge on which you require data to become valid, this will usually be determined by the particular peripheral in question and its data sheet should be studied carefully.

```
void init_SPI_pins(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;
   SPI_InitTypeDef SPI_InitStruct;

   RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI3, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC,
ENABLE);

   /* GPIOD Configuration: (SCK GPIOC10, MISO C11,
MOSI C12) */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10 |
GPIO_Pin_11 | GPIO_Pin_12;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

   GPIO_Init(GPIOD, &GPIO_InitStructure);
```

```
    GPIO_PinAFConfig(GPIOC, GPIO_Pin_10, GPIO_AF_SPI3);
    GPIO_PinAFConfig(GPIOC, GPIO_Pin_11, GPIO_AF_SPI3);
    GPIO_PinAFConfig(GPIOC, GPIO_Pin_12, GPIO_AF_SPI3);
/* note this function mishandles AFR[1] assignment */
    GPIOC->AFR[1] = 0x00066600;

    SPI_InitStruct.SPI_Direction = SPI_
Direction_2Lines_FullDuplex;
    SPI_InitStruct.SPI_Mode = SPI_Mode_Master;
    SPI_InitStruct.SPI_DataSize = SPI_DataSize_16b;
    SPI_InitStruct.SPI_CPOL = SPI_CPOL_Low;
    SPI_InitStruct.SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitStruct.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStruct.SPI_BaudRatePrescaler = SPI_
BaudRatePrescaler_2;
    SPI_InitStruct.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStruct.SPI_CRCPolynomial = 0;

    SPI_Init(SPI3, &SPI_InitStruct);

    SPI_Cmd(SPI3, ENABLE);
}
```

### *5.4.1 SPI Interface to an Analogue to Digital Converter*

Although there are many components equipped with comprehensive implementations of the SPI interface many others use a more application oriented implementation set up to accomplish specific design objectives. In this example application the 16-bit AD7680 was designed to operate up to 100k samples per second so a dedicated SPI interface implementation was used. This AD converter only has a MISO connection besides the clock and its select signal is used to trigger new conversions. The circuit diagram is shown in Figure 5.7.

The SPI master mode will be used to generate clocks, although no command to the slave is actually required, so a dummy 0xff data byte is conventionally transmitted. To start with the converter has to be brought out of its power-down mode before a valid conversion can be made, this can be achieved by issuing a dummy 16 clock cycles with SS low. Conversion will start the next time SS goes low. It will be essential to generate the SS signal by manipulating a simple output because when the SPI is operating in master mode the slave select signal generated internally remains low continuously. For this

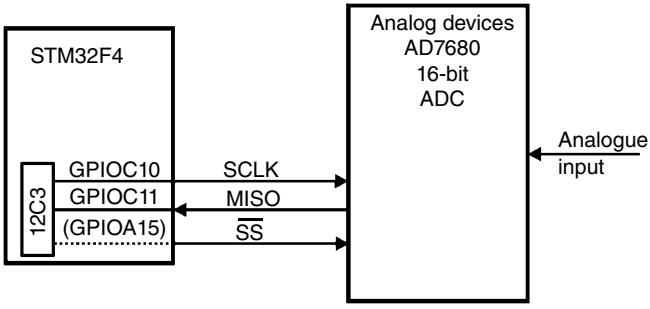


**Figure 5.7** AD7680 connections

application GPIOA pin 15 is assigned as a simple output so that it can be set low before clocks are generated and return high afterwards.

The SPI direction can be temporarily reassigned if required:

```
SPI_BiDirectionalLineConfig(SPI_Direction_1Line_Tx);
```

The code shown next will generate 16 clock cycles and wake up the converter.

```
   GPIO_ResetBits(GPIOA, SS_Pin);
   SPI_I2S_SendData(SPI3, temp);
   while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
TXE) != 1)
   {
   }
   SPI_I2S_SendData(SPI3, temp);
   while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
TXE) != 1)
   {
   }
   GPIO_SetBits(GPIOA, SS_Pin);
```

Data can then be retrieved from the converter as soon as the receiver register is full. However, the converter requires 24 clock cycles on SCL to complete the conversion and deliver all the data bits. Within this bit-stream the data sheet shows that there are four leading and four trailing zeros surrounding the 16-bit converter data. Three bytes will need to be transmitted to achieve this. Reading out the received byte can be interleaved with the transmission so that the receiver buffer is ready for the next reception.

```
GPIO_ResetBits(GPIOA, SS_Pin);
SPI_I2S_SendData(SPI3, temp);
while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
TXE) != 1)
{
}
SPI_I2S_SendData(SPI3, temp);
while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
RXNE) != 1)
{
}
temp1 = SPI_I2S_ReceiveData(SPI3);
while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
TXE) != 1)
{
}
while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
RXNE) != 1)
{
}
temp2 = SPI_I2S_ReceiveData(SPI3);
```

Further code along the same lines will be needed to complete the required 24 clock cycles. The slave select pin can be taken high again after all the clocks have been delivered.

## 5.5 HDLC Serial Communication

As the demand for high speed digital communication increased the limitations and overheads of the simple strategies described earlier became evident. Also with the advent of improved techniques such as the phase-locked-loop (PLL) it became much more straightforward to regenerate a reliable clock in the receiver using the data edges as a reference. The PLL has inherent inertia like its mechanical counterpart so its operation will not be compromised if some of the data edges are missing, such as when there are consecutive ones or zeros. All that must be insured is that such a constant one or zero situation is not allowed to persist too long. The High-Level Data Link Control (HDLC) scheme employs extra bits stuffed into the data stream every time there are five consecutive ones, these can easily be removed in the receiver to restore the original data. The start of a data frame is still required so a special Flag

symbol of six consecutive ones is used, this can never occur anywhere else in the frame. The frame is transmitted LSB first in the NRZI format where consecutive ones remain high and zeros change at the centre of bit time. This ensures a data transition at least every 6 bit times during the frame and every 7 during the flags. Specialised hardware in the transmitter and receiver interface modules handles these operations, which mark out clearly the frames boundaries and provide enough edges to stabilise the PLL clock. The resulting frame structure is shown in Figure 5.8.

The Address byte effectively forms a channel number to distinguish primary and secondary sources.

The control byte distinguishes three types of frame as shown in Figure 5.9, Information (I), Supervisory (S) and Unnumbered (U). In the information frame control byte there are both send N(S) and receive N(R) sequence numbers encoded in three bits so error control information can be included. In a later version the sequence numbers were extended to 7 bits to improve efficiency. The supervisory frame is used for flow and error control when there is no data to send, there are only four possible frames in this group. The U frames are used mainly for link set up and management functions.

The P/F bit has two functions, Poll when set by the primary to obtain a response from a secondary, and Final when set by the secondary to indicate a response or the end of a transmission sequence.

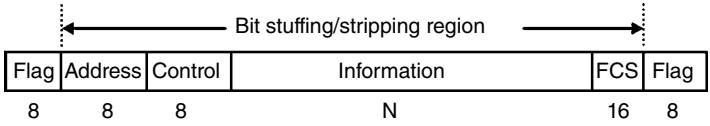


**Figure 5.8** HDLC frame structure

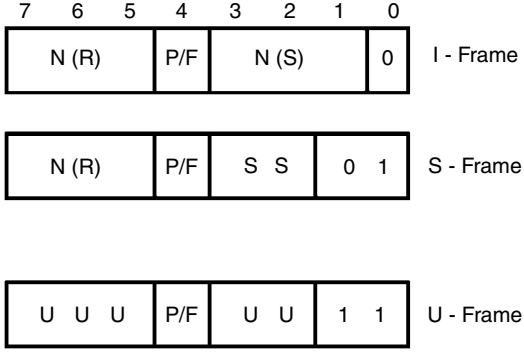


**Figure 5.9** HDLC frame control byte

The frame check sequence (FCS) is a cyclic redundancy check (CRC) calculation that forms a very effective test for errors on the complete frame. The frame always finishes with another Flag, which may sometimes form the start Flag for the next frame.

Only a few of the details have been discussed here but HDLC was the inspiration for the standard IEEE 802.2, which is widely employed in connecting to the Internet so its relevance is obvious.

## 5.6 The Universal Serial Bus (USB)

The USB protocol was introduced to provide a more flexible and reconfigurable interconnection arrangement for PC peripherals, that would allow items to be added or removed without the need to modify and adapt the whole system. USB is now very familiar to PC users allowing a wide variety of devices such as digital cameras, MP3 audio players, webcams and memory sticks to be connected. It is another serial communications protocol with many similarities to HDLC that has been widely adopted throughout the PC industry but is much more complex than the other techniques that have been covered in this chapter so far. A sketch of the USB connector is shown in Figure 5.10 but a wide range of connector physical sizes are available.

USB employs a four wire connector comprising of data lines (D+ and D−), power and ground. The D+ and D− form a differential connection path using a pair of twisted wires, which allows the USB to work at high speeds across a link up to 5 m in length. The included power connection provides a 5 V supply, supporting loads up to 100 mA, which can be used by the linked device if it does not have its own power source. The connector is designed carefully to ensure the order of connection so that 'hot swapping' can be accomplished safely. The cable shield is connected first, then the power and ground then finally the data lines. Three classes of data rate are recognised standards, these are high speed (HS) 480 Mbps, full speed (FS) 12 Mbps and low speed (LS) 1.5 Mbps.

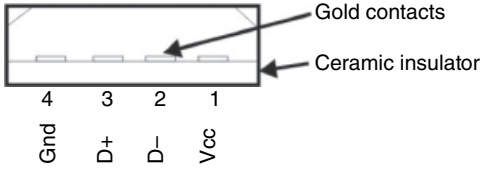


**Figure 5.10** USB connections

During USB communication data is transmitted as packets in NRZI format, where a sequence of ones remain high but zeros always toggle at the bit centre. The packet always starts with a special sync (00000001) and bit stuffing is used to break up sequences of one. A special terminating end-of-packet (EOP) sequence is also defined. Initially all packets are sent from the host, via the root hub, to devices. Some of those packets direct a device to send some packets in reply. There are three main types of packet and these are Hand-shake, Token and Data, which all have a unique packet identifier (PID) as the first byte. Table 5.3 shows a basic set

**Table 5.3** USB PID byte functions

| PID # | PID byte | Name | Notes |
|---|---|---|---|
| 0000 | 0000 1111 | | Reserved |
| 1000 | 0001 1110 | **SPLIT** | USB 2 split transaction |
| 0100 | 0010 1101 | **PING** | Does endpoint accept USB 2? |
| 1100 | 0011 1100 | **PRE** | Special for low-bandwidth USB |
| | | **ERR** | Split transaction error for USB 2 |
| 0010 | 0100 1011 | **ACK** | Data packet accepted |
| 1010 | 0101 1010 | **NAK** | Data packet not accepted; retransmit required |
| 0110 | 0110 1001 | **NYET** | Data not ready yet (USB 2) |
| 1110 | 0111 1000 | **STALL** | Transfer impossible; error recovery needed |
| 0001 | 1000 0111 | **OUT** | Host-to-device transfer address |
| 1001 | 1001 0110 | **IN** | Device-to-host transfer address |
| 0101 | 1010 0101 | **SOF** | Start of frame marker sent every millisecond |
| 1101 | 1011 0100 | **SETUP** | Host-to-device control transfer address |
| 0011 | 1100 0011 | **DATA0** | Even # data bytes |
| 1010 | 0101 1010 | **DATA1** | Odd # data bytes |
| 0111 | 1110 0001 | **DATA2** | USB 2 data packet |
| 1111 | 1111 0000 | **MDATA** | USB 2 data packet |

Key:

| | | |
|---|---|---|
| Hand-shake | Token | Data |

of PID values that have been extended to accommodate the USB 2 standard. The actual PID byte is transmitted LSB first and the last four bits are a complement of the first four. This helps with error checking when the packets are short.

### *5.6.1 Hand-shake Packets*

These packets only use the PID byte; error checking relies on the byte structure alone. The only possible response from the host is ACK; if it is not ready it won't ask for data to be sent. NYET and ERR were added for USB2 extensions. NYET allows the device to indicate that its buffers are full so the host can use a PING before proceeding rather than sending a whole frame that will be rejected.

### *5.6.2 Token Packets*

Tokens are only sent by the host and consist of the PID followed by an 11 bit field and a 5 bit CRC. The 7 bits of this data field are the device address and the last four give the number of data packets required. The appropriate hand-shake for **IN** and **OUT** tokens will be expected in response.

The **SETUP** token operates like an **OUT** token but is followed by an eight byte **DATA0** packet with a special format.

The USB host sends out a start of frame **SOF** token, which contains an 11-bit incrementing number field, every millisecond to synchronise data transfers. In USB 2 seven extra **SOF** tokens are introduced to define 'micro-frames' containing 60 000 bit times for the fastest data rate.

### *5.6.3 Data Packets*

Up to 1024 data bytes can follow these PID values (64 for medium speed and 8 for low speed). The two forms DATA0 and DATA1 alternate and allow a better error management to be achieved. The receiver keeps track of the sequence so that when a packet is lost the sequence will be violated and a retransmission can be requested.

The most recent acknowledgement will indicate which type was received correctly. So unless the acknowledgement matches the last packet transmitted it is this that will need retransmission.

The PRE packet is a special preamble issued by the host just prior to a low bandwidth packet communication.

### *5.6.4 USB Protocol*

When the device is first connected the host performs a process called enumeration where it first resets the device, then assigns it an address and proceeds to interrogate it to determine the basic operating characteristics, such as device type and data rate. All subsequent data communications are initiated by the host and use a hand-shake exchange to confirm success. The host uses the address of the target device and specifies the class type and direction of data transfer required. As devices are enumerated, the host keeps track of the total bandwidth that all of the devices are requesting whether they are using the isochronous or interrupt driven modes. They can consume up to 90% of the total bandwidth available (i.e. with USB 2.0 480 Mbps and with USB 3.0 4.8 Gbps). After the 90% allocation is used up, the host denies access to any other isochronous or interrupt driven devices. Control packets and packets for bulk transfers use any bandwidth left over (i.e. at least 10%).

The STM32F4 device contains two USB interface modules that can be configured to operate as host or slave device according to the particular application envisaged. These are both fully implemented on the Discovery board and one of them is used to download code from the PC host and interact with the integrated development environment (IDE) in Debug mode.

The ST Microelectronics support package contains valuable USB examples using both the host mode and the device mode. One of the host examples in the file USB_Host_Examples implements an inertial mouse based on the Discovery board using the USB human interface device (HID) class and another in the file USB_Device_Examples provides an interface, using the USB mass storage interface class (MSC), to a memory microSD card. Other examples using the host and device modes are also provided giving a range of possible applications.

## 5.7 Programming Challenge

The development of an interface for the touch-sensitive screen is required; this is shown in Figure 5.11 and uses the I2C interface to communicate with the STMP811 controller chip on the LDC35RT display board. Whenever a user input is detected a corresponding set of X-Y coordinates should be delivered to the hyper terminal on the PC. Do not attempt to use the interrupt at this stage unless you are familiar with the material in Chapter 7.

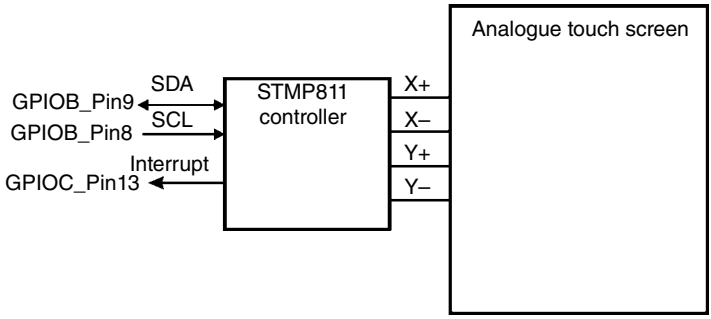


**Figure 5.11** Touch screen interface

## 5.8 Conclusion

Various serial interface protocols including RS232, I2C, SPI and USB, have been examined in this chapter. These are widely utilised in common practice for a huge variety of different applications and a thorough knowledge of them will be invaluable when a new design is considered or new system components are required.

The software applications illustrated utilise simple polling techniques to determine when data can be transmitted or when data has been received but it should be becoming evident that more efficient interface management would be a significant advantage particularly when there is a large volume of data to deliver or retrieve. The interrupt techniques essential in achieving this will be described in Chapter 6.

Only brief details of advanced protocols like USB have been provided in this chapter because very comprehensive and well documented examples are provided in the STM32F4 Discovery support software package [1]. These examples will form a useful starting point for new design requirements.

## References

[1] ST Microelectronics (2011) STM32F4 Reference Manual. RM009 (ARM Cortex-4) Reference Manual Doc ID 018909 Rev 1, www.st.com (accessed September 2014).

[2] ST Microelectronics (n.d.) STMPE811 Data Sheet, Doc ID 14489 Rev 5, www.st.com (accessed September 2014).

STM32F4 Peripheral Driver functions from the support library:
stm32f4xx_usart.c
stm32f4xx_i2c.c
stm32f4xx_spi.c
USB Protocol Documentation from various web sites.