ZBasic

Application Note

AN-219 Implementing I2C and SPI Slaves

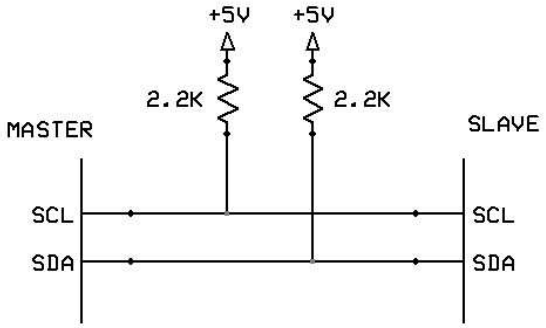
Introduction

With the introduction of native mode ZX devices it became possible to implement a broader range of applications, notable among which are applications that require a customized interrupt service routine (ISR). Due to the interface timing requirements, implementing an I2C or SPI slave requires a customized ISR (in most cases). This application note illustrates how to implement both an I2C slave and an SPI slave on a native mode ZX device. The source code accompanying this application note contains example I2C and SPI slave implementations along with simple test drivers (i.e. a master) for each slave. The example code may prove useful as a starting point for implementing an I2C or SPI slave to meet your specific needs.

Overview of the I2C Interface

The I2C interface (more properly written I^2C) invented by Philips, is a multi-master, multi-slave synchronous serial, half-duplex interface that uses two signals for timing and data. The signals, SCL (serial clock) and SDA (serial data), are "open drain" thus allowing bi-directional communication on both lines. Consequently, both lines require a pull-up resistor the value of which depends on the capacitance of the signal line but a typical value is 2.2K ohms.

The simplified schematic below shows the electrical connections between an I2C master and slave. If multiple slaves are connected, each is connected in the same manner as shown, essentially in parallel with the slave shown. Only one set of pull-up resistors is used regardless of the number of slaves.



I2C Master-Slave Connection

Generally, communication on the I2C bus occurs between one master and one slave. The master controls the timing via the SCL signal which it generates but the slave can, if it needs more time to respond, stretch the clock cycle by holding the SCL line low. The SDA line is used for sending data from the master to the slave and vice versa.

An I2C master initiates a bus cycle by creating a specific sequence of transitions on the SDA and SCL lines (known as the start or repeated start signal) followed by transmitting an 8-bit address value serially on the SDA line. The seven most significant bits of that value comprise the slave address, indicating the slave with which the master wishes to communicate (but note that the special value of zero is reserved for addressing all slaves simultaneously). The least significant bit of the address value indicates whether the master wishes to write to the slave (zero) or to read from the slave (one). When the master is sending data to the slave, the address byte is followed by one or more data bytes generated by the master. When the master is receiving data from the slave, the address packet is followed by one or more data bytes generated by the slave. Each data byte is accompanied by a ninth bit, known as the acknowledge bit, that is generated by the receiver to indicate the acceptance of the data. When all data bytes have been transmitted, the cycle ends with a repeated start signal (to begin another read or write sequence) or with a special sequence on SDA and SCL called a stop signal.

Published June 2008

More detailed information about the I2C protocol is available at http://en.wikipedia.org/wiki/I2C and elsewhere. Note that Atmel refers to the I2C-compatible interface on its AVR chips by the term TWI (two-wire interface). This application note uses the terms I2C and TWI interchangeably.

Example I2C Slave Implementation

The accompanying source code contains a simple implementation of an I2C slave. The slave recognizes several different commands that return data of a fixed length, variable length or return no data at all. Most of the work in the I2C slave code is done within the I2C slave ISR and its subordinate routines. Because the ISR is be invoked for each byte that is transmitted and received, the ISR needs to maintain state information between invocations. Each time the ISR is invoked, it responds differently depending on the current state and the latest received data. This requirement leads naturally to an implementation based on a finite state machine (FSM). In this implementation, the state transitions and outputs of the state machine are controlled entirely by program statements as opposed to being table driven. (For more information on the FSM concept, see http://en.wikipedia.org/wiki/Finite_State_Machine.)

As noted earlier, most of the work in the I2C slave code is performed within the I2C ISR aided by a small helper routine. The commands supported by the example I2C slave may be divided into two basic groups: those that require a reply to the master and those that do not. We now turn our attention first to the latter group by considering the following excerpt from the ISR.

```
'----------------------------
' slave receive mode states
'----------------------------
Case I2C_SR_SLA_ACK         ' slave has been addressed for receiving
    twcr = twcr Or TWINT

Case I2C_SR_DATA_ACK        ' received data with ack
    data = Register.TWDR
    twcr = twcr Or TWINT
    Select Case commState

    Case STATE_IDLE         ' in the idle state, the data is a command byte
        commState = data
        commIdx = 0

    Case STATE_SEND_CHAR    ' send the received byte to Com1
        Call PutQueue(txQueue, data, 1)
        commState = STATE_IDLE

    Case STATE_SEND_STR     ' received byte is character count
        commIdx = CInt(data)
        commState = IIF(commIdx = 0, STATE_IDLE, STATE_SEND)

    Case STATE_SEND         ' send received bytes to Com1
        Call PutQueue(txQueue, data, 1)
        commIdx = commIdx - 1
        If (commIdx <= 0) Then
            commState = STATE_IDLE    ' done receiving
        End If

    Case Else               ' unknown state, return to idle state
        commState = STATE_IDLE

    End Select
```

This excerpt consists of two cases of the outermost `Select` statement in the ISR. The first case, denoted by the constant `I2C_SR_SLA_ACK`, occurs each time the slave recognizes its slave address with the LSB low indicating

that the master is invoking a cycle to write one or more bytes to the slave (i.e. the slave is in receive mode). For this particular case, all that is necessary to do is to reset the TWINT flag in the TWCR register. This is done by writing a 1 to the bit position corresponding to TWINT as described in the Atmel documentation; the statement following the I2C_SR_SLA_ACK case prepares for doing so later on.

The more interesting case, denoted by the constant I2C_SR_DATA_ACK, occurs for each data byte received from the master, the first of which is always a command byte when the FSM state is STATE_IDLE. Depending on the command, one or more data bytes follows the command byte. The two commands implemented in the excerpt above are STATE_SEND_CHAR (accompanied by one additional byte) and STATE_SEND_STR (accompanied by a byte count datum and zero or more data bytes comprising the characters of the string). The code for STATE_SEND_CHAR is quite simple: the received byte is sent to the serial port and the FSM state is set back to STATE_IDLE to await the next command. For STATE_SEND_STR, the first received byte is the number of bytes to follow (which may be zero). The code for that state saves the byte count in commIdx and then sets the FSM state to either STATE_SEND, if the count is non-zero, or STATE_IDLE. Finally, in the STATE_SEND state, the each received string character is sent to the serial port and the length counter commIdx is decremented. When the counter commIdx reaches zero the FSM state is set to STATE_IDLE.

We consider now the second sub-group of commands implemented by the slave – those that elicit the return of data by the slave to the master. The code that implements this set of commands appears in the excerpt below.

```
    '----------------------------
    ' slave transmit mode states
    '----------------------------
    Case I2C_ST_SLA_ACK          ' ready for the first data byte to be sent
        twcr = sendDataToMaster (twcr)

    Case I2C_ST_DATA_ACK         ' master replied with ACK, ready for the next byte
        twcr = sendDataToMaster (twcr)

    Case I2C_ST_DATA_NACK        ' master replied with NAK, terminate cycle
        twcr = twcr Or (TWINT Or TWEA)

    Case I2C_ST_LAST_DATA        ' the last data byte has been sent
        twcr = twcr Or (TWINT Or TWEA)
```

The first case, denoted by the constant I2C_ST_SLA_ACK, occurs when the slave recognizes its slave address on the bus with the LSB set to 1 indicating a read cycle, i.e. slave transmit mode.

The second case, denoted by the constant I2C_ST_DATA_ACK, occurs each time the master acknowledges the byte just sent by the slave. The third case, I2C_ST_DATA_NACK, occurs if the master, for some reason, responds with a negative acknowledgement. In this case, the transmission cycle is prematurely terminated. The final case, I2C_ST_LAST_DATA, occurs when the last data byte has been sent and the master has acknowledged its receipt. The first two cases utilize a helper routine, sendDataToMaster(), to do the actual work of determining the data byte to send back, sending it and preparing for the next, if any. The code for the helper routine is reproduced below.

```
Private Function sendDataToMaster(ByVal twcr as Byte) As Byte
    Dim dataOut as Byte
    Dim nextState as Byte

    ' set the default data output and "next state"
    dataOut = 0
    nextState = STATE_IDLE

    ' prepare outbound data depending on the state
    Select Case commState
```



```
    Case STATE_ID_LEN         ' sending ID string length
        dataOut = idStrLen

    Case STATE_ID_STR         ' sending ID string
        commIdx = commIdx + 1
        If (commIdx <= CInt(idStrLen)) Then
            dataOut = Asc(idStr, commIdx)
            If (commIdx <> CInt(idStrLen)) Then
                nextState = STATE_ID_STR
            End If
        End If

    Case STATE_RAM_SIZE  ' sending RAM size, low byte first
        If (commIdx = 0) Then
            dataOut = LoByte(Register.RamSize)
            commIdx = 1
            nextState = STATE_RAM_SIZE
        Else
            dataOut = HiByte(Register.RamSize)
        End If

    End Select

    ' update the state variable
    commState = nextState

    ' modify the value of the I2C control register
    twcr = twcr Or (TWINT Or TWEA)
    If (commState = STATE_IDLE) Then
        ' this is the last data byte, send NAK
        twcr = twcr And Not TWEA
    End If
    sendDataToMaster = twcr

    ' output the data
    Register.TWDR = dataOut
End Function
```

The cases of the `Select` statement correspond to FSM states and, also, to commands received. The first, `STATE_ID_LEN`, corresponds to a request to return the length of the device ID string. This is realized by setting `dataOut` to the previously determined string length, which value is later written to the `TWDR` register thus sending it back to the master. Since no other work is needed to implement this command, the default "next state" of `STATE_IDLE` is left undisturbed.

The second state, `STATE_ID_STR`, represents a request to return the actual characters of the device ID string. This state occurs repeatedly until the variable `commIdx` reaches the predetermined value `idStrLen`, each time returning a subsequent character of the device ID string. Until all of the characters of the device ID string have been returned, the default "next state" is set to `STATE_ID_STR` to allow the remaining characters to be sent on subsequent cycles. Clearly, `commIdx` serves as an auxiliary state variable that supplements the primary state variable `commState`.

The last state, `STATE_RAM_SIZE`, is a request to return the RAM size of the slave. Here again, `commIdx` serves as an auxiliary state variable to determine which byte of the two-byte RAM size value should be sent.

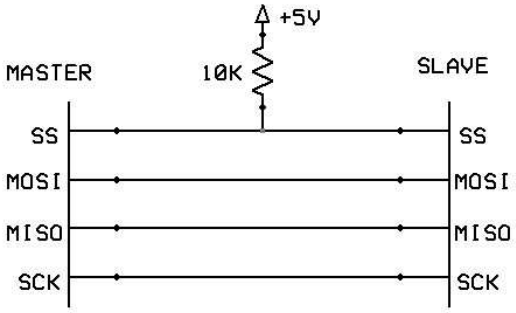
Following the `Select` statement, the remaining code of the helper routine updates the state variable, prepares an updated value to be returned (subsequently written to the `TWCR` register) and, finally, outputs generated data to the `TWDR` register.

Published June 2008

Overview of the SPI Interface

The SPI interface, devised by Motorola, is a single-master, multiple-slave synchronous serial, full-duplex interface that uses three common signals for timing and data plus a separate signal for each slave to indicate when it is being addressed. The SCK signal, generated by the master, controls the timing of the data transfer. The MOSI signal (generated by the master) conveys data from the master to the slave while the MISO signal (generated by the slave) conveys data from the slave to the master.

The simplified schematic below shows the electrical connections between an SPI master and slave. If multiple slaves are connected, each is connected in the same manner as shown, essentially in parallel with the slave shown, except that each slave utilizes a separate SS signal with its own pull-up.



SPI Master-Slave Connection

It is important to note that on each cycle of SCK, the master sends a bit of data on MOSI and, simultaneously, receives a bit of data on MISO. Similarly, on each cycle of SCK, the selected slave both receives a data bit on MOSI and sends a data bit on MISO. Eight such cycles are required to transmit a complete byte simultaneously in both directions. It is important to note that even though data is transmitted in both directions, the master and slave may send meaningless data or ignore received data as deemed appropriate. If multiple slaves are connected, a particular slave responds only if its Slave Select signal is active (low).

More detailed information about the SPI protocol is available at http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus and elsewhere.

Example SPI Slave Implementation

The accompanying source code contains a simple implementation of an SPI slave. The slave recognizes the same set of commands as the I2C example. As with the I2C example, most of the work in the SPI slave code is done within the SPI slave ISR and its subordinate routines. Because the ISR is invoked for each receive/transmit cycle, the ISR needs to maintain state information between invocations. As with the I2C example, the primary state variable is `commState` and `commIdx` is an auxiliary state variable. One notable difference between the SPI slave state machine and the I2C slave state machine is that on each invocation of the ISR in the SPI case a data byte is received from the master (which may or may not be used) and a data value is prepared for transmission back to the master on the next cycle. This makes the ISR simpler than the one for I2C; all of the work can be done in the ISR without the need for a subordinate routine. Beyond that, the code is nearly identical to the I2C case.

```
ISR SPI_STC()
   Dim dataIn as Byte
   Dim dataOut as Byte
   Dim nextState as Byte

   ' set the default data output and "next state"
   dataOut = 0
   nextState = STATE_IDLE

   dataIn = Register.SPDR

   ' if the FSM is in the idle state, the received value is a command
```

```
    If (commState = STATE_IDLE) Then
        commState = dataIn
        commIdx = 0
    End If

    ' prepare outbound data depending on the state and compute the next state
    Select Case commState
    Case STATE_ID_LEN         ' sending ID string length
        dataOut = idStrLen

    Case STATE_ID_STR         ' sending ID string
        commIdx = commIdx + 1
        If (commIdx <= CInt(idStrLen)) Then
            dataOut = Asc(idStr, commIdx)
            If (commIdx < CInt(idStrLen)) Then
                nextState = STATE_ID_STR
            End If
        End If

    Case STATE_RAM_SIZE       ' sending RAM size, low byte first
        If (commIdx = 0) Then
            dataOut = LoByte(Register.RamSize)
            commIdx = 1
            nextState = STATE_RAM_SIZE
        Else
            dataOut = HiByte(Register.RamSize)
        End If

    Case STATE_SEND_CHAR      ' send the next received byte to Com1
        commIdx = 1
        nextState = STATE_SEND

    Case STATE_SEND_STR       ' received data is count, send next N characters to Com1
        nextState = STATE_GET_COUNT

    Case STATE_GET_COUNT      ' send the next received byte to Com1
        commIdx = CInt(dataIn)
        If (commIdx > 0) Then
            nextState = STATE_SEND
        End If

    Case STATE_SEND               ' send characters to Com1
        Call PutQueue(txQueue, dataIn, 1)
        commIdx = commIdx - 1
        If (commIdx > 0) Then
            nextState = STATE_SEND    ' there are more characters to receive
        End If
    End Select

    ' store the data for the next transmit cycle
    Register.SPDR = dataOut

    ' update the state variable
    commState = nextState
End ISR
```

Software

A key part of this application note is the associated ZBasic software, which is provided in four separate projects. The project i2c_slave.pjt is the slave side of the I2C connection while the project i2c_slave_test.pjt is a simple test driver for the master side. Similarly, The project spi_slave.pjt is the slave side of the SPI connection while the project spi_slave_test.pjt is a simple test driver for the master side.

Author

Don Kinzer is the founder and CEO of Elba Corporation. He has extensive experience in both hardware and software aspects of microprocessors, microcontrollers and general-purpose computers. Don can be contacted via email at dkinzer@zbasic.net.