ISA to Single-Channel I2C Multi-Master Implementation

SCENIX

Application Note 21
Wing Poon
September 1999

1.0 Introduction

The I2C-bus is a common synchronous serial protocol used by many Integrated Circuits to communicate with each other. The ISA-bus is commonplace in Intel x86-based PCs and embedded controllers. The I2C protocol dictates that a device is, at any time, either a master or a slave. A master always communicates with a slave, and vice versa. Furthermore, a class of I2C Master devices – I2C Multi-Masters – are cable of collision detection and arbitration, thus allowing them to share the I2C-bus with other Multi-Masters.

In this document, we examine an implementation of a I2C Multi-Master that is interfaced through the ISA-bus. This implementation provides the bridge for which the PC can then communicate with I2C Slave devices.

A related document, “Interfacing the SX to the ISA-XT Bus”, which should be read first, describes the hardware and software components that allow the SX28AC to talk over the ISA-bus. This document builds on that platform, showing the integration of an I2C Multi-Master Virtual Peripheral in that same SX28AC communications controller.

The complete source-code (ISA_I2C.ASM) for this project is available. Please refer to it as you work your way through the rest of this document.

2.0 Source Code Structure

The source program is divided into sections, beginning with the **`Device`** section. This merely specifies the fuse-bits in the SX device.

The next section is **`Variables. Global`** variables are bank-independent and are used to contain frequently accessed data. Bank 0 ($10-$1F) is a bank dedicated for I2C VP.

The **`ISR`** section contains the entire I2C and ISA VPs. In other words, both VPs are entirely interrupt-driven, with no mainline-code required, except for module initializations. Upon entry into the ISR, the I2C code is executed first, followed by the ISA code.

The **`Main Program`** section calls the VPs initialization code, then the mainline program spins in an infinite-loop, doing nothing.

3.0 Interrupt-Service Routine

The I2C VP operates as a state-machine. There are major-states and minor-states. Transitions between minor-states is always fixed (increasing minor-state numbers). Transitions between major-states is dynamically determined by external stimuli.

The I2C state-machine resides in the function **`I2CM_ISR.`** There are 12 major-states in this state-machine. For a description of each state, please refer to the comments in the source code. The current state is stored in the variable **`I2CM_state.`** The idle state has a state value of zero. Within each major-state, there may exist one or more minor-states. The current minor-state is stored in the variable **`I2CM_sub_state.`**

Interfacing to the I2C VP and state-machine is done through the following variables: **`I2CM_state, I2CM_address, I2CM_num_bytes, I2CM_data_buf`** and **`I2CM_ctrl_stat.`** This is the interface between the ISA VP and the I2C VP.

The ISA VP first checks the CS* pin and does nothing if it is not asserted. However, if it is, it proceeds to check the IOWC* and IORC* pins to determine if a read or write cycle is in progress. In both cases, SA0 and SA1 are then looked at next to determine which of the four ISA addressable registers (0 – 3) are being addressed.

A read cycle from three (0 – 2) of the four available registers merely retrieves the current values of the **`I2CM_address, I2CM_num_bytes`** and **`I2CM_ctrl_stat`** variables. A read of the fourth register (3) retrieves one byte in the 7-byte **`I2CM_data_buf`** array. There exists an index, **`I2CM_data_ptr,`** into this data array, which is used to select which byte is read. After the byte has been read, the pointer is incremented by one.

A write cycle into registers 0 and 1 merely updates the values of the **`I2CM_address`** and **`I2CM_num_bytes`** variables. A write to register 3 will update one byte in the **`I2CM_data_buf,`** again pointed to by **`I2CM_data_ptr`**, and will also cause the pointer to be incremented. A write to register 2, the **`I2CM_ctrl_stat`** register, is special because the act of writing this register will initiate an I2C-bus transaction. The actual value that is written to this register is irrelevant and will be ignored.

4.0 PC - I2C Interface

The PC's interface to the ISA/I2C VP is done through the following four registers: **I2CM_address** (base_addr+0), **I2CM_num_bytes** (base_addr+1), **I2CM_ctrl_stat** (base_addr+2) and **I2CM_data** (base_addr+3). This is almost identical to the ISA – I2C interface except for **I2CM_data_buf,** which is a 7-byte length variable, and **I2CM_data,** which is a single-byte variable.

The PC I/O space is the range of addresses between 100h – 3FFh. The four registers required to interface to the I2C VP can fall anywhere in this range, subject to the following conditions:

- the 4 registers must fall on consecutive addresses
- the base address must be an integer multiple of 4

In the demonstration circuit (see "Interfacing the SX to the ISA-XT Bus"), the base address is selectable by means of seven switches. For the rest of this discussion, a base address of 300h is assumed.

I2CM_address (base_addr+0)

All I2C devices have a 7-bit device address. The most-significant 7-bits of this register reflects that address. Bit0, the least-significant bit, should be a '0' if the I2C master (i.e. the SX) wishes to write to a slave device, or a '1' if the master wishes to read from a slave device.

Important Note: A write to this register (regardless of value) will reset the data buffer index pointer to point to the start of the buffer.

I2CM_num_bytes (base_addr+1)

This refers to the number of *data* bytes to be transferred in the upcoming I2C bus transaction. This number can be between 0 and 7 (the upper limitation is due to the current data buffer size of this I2C VP implementation). A write of zero data bytes is useful for 'pinging', or checking the presence of, slave devices on the I2C-bus.

Important Note: This register is modified by the I2C VP, and will read zero after completion of a read or write transaction.

I2CM_ctrl_stat (base_addr+2)

This register is a status register when read, and a control register when written. When read, the following bits each report a status condition (the bit is set to '1' if the condition is true):

Bit	Description
0	Busy flag
1	NACK flag
2	Reserved
3	Lost-Arbitration flag

The Busy flag indicates that the I2C state-machine is busy doing a read or write transaction. Before initiating a new transaction, your application *must* check that this flag is clear, or else it must wait. After starting the transaction (by writing a dummy value to this register), you can repeatedly read this register – but any other reads or writes to any other register is forbidden. If the requested transaction was a read, the clearing of this flag (by the I2C VP) indicates that the I2C read transaction is done and the data is in the internal data buffer.

The NACK flag indicates that the I2C Master (SX) did not receive an acknowledgment to the previous bus transaction.

The Lost-Arbitration flag indicates that the last transaction was aborted because a collision was detected and the SX lost control of the bus to another master.

When the PC writes to this register, the I2C state-machine is awakened from its idle state and will execute the bus transaction set up by the other three registers. A write to this register will automatically reset the data buffer pointer, as well as set the Busy flag.

I2CM_data (base_addr+3)

This register is used to present the data for transmission (in the case of a write request), or retrieve the data sent by the slave device (in the case of a read request). It accesses one byte in the 7-byte FIFO data buffer in the I2C VP. Each subsequent read or write access to this register will retrieve the next received byte or set the next byte to be transmitted, respectively. A write to **I2CM_address** will reset the internal pointer back to the start of the buffer.

4.1 IMPORTANT CONSIDERATIONS

- After requesting a read transaction, poll the Busy flag until it is cleared, at which point, you must do a dummy write to the **I2CM_address** register (to reset the buffer pointer), before starting to read **I2CM_data.**
- Severe problems will arise if you try to modify or read registers (with the exception of reading the **I2CM_ctrl_stat** register) while the I2C state-machine is busy.
- Remember there is a 7-byte limit to the number of data bytes per transaction.

5.0 Example 'C' Application Program

Appendix A shows a example application, written in 'C', that you can run on a PC to interface to this I2C controller. The program is DOS-based, and can be compiled with Borland C, Turbo C and DJGPP.

The demonstration circuit has a 24LC16B serial-EEPROM slave device connected to the I2C-bus. This program will read from and write to the EEPROM.

Upon starting, the program will print a help screen. There are commands to dump the contents of the EEPROM, read a specific address, or write to a specific address.

6.0 Summary

The Scenix SX communications controller is in a league of its own in terms of performance and versatility. The speed of the part allows the SX to interface very easily to the PC's ISA-bus. The versatility of the part comes from the ability for you, the designer, to incorporate complex Virtual Peripherals, such as an I2C Multi-Master, along with other VPs if desired, on one SX device, thereby reducing system cost and complexity.

7.0 Appendix A

```
/* I2C_DEMO.C */
/* Wing Poon.8/17/99.(C) Scenix Semiconductor, Inc.*/

#include <stdio.h>
#include <string.h>

#define I2C_BASE_ADDR    0x300
#define I2C_ADDRESS      (I2C_BASE_ADDR+0)
#define I2C_NUM_BYTES    (I2C_BASE_ADDR+1)
#define I2C_CTRL_STAT    (I2C_BASE_ADDR+2)
#define I2C_DATA         (I2C_BASE_ADDR+3)

#define EEPROM_ADDR      0xA0
#define EEPROM_ADDR_WR   (EEPROM_ADDR & 0xFE)
#define EEPROM_ADDR_RD   (EEPROM_ADDR | 0x01)


void eepromSetAddr(unsigned char addr)           /* sets the current EEPROM address counter value */
{
  outportb(I2C_ADDRESS, EEPROM_ADDR_WR);
  outportb(I2C_NUM_BYTES, 1);
  outportb(I2C_DATA, addr);
  outportb(I2C_CTRL_STAT, 0);
  while(inportb(I2C_CTRL_STAT)&0x01);             /* wait for BUSY to be cleared */
}

unsigned char eepromSeqReadByte(void)            /* read the byte pointed to by the EEPROM address
counter */
{
  outportb(I2C_ADDRESS, EEPROM_ADDR_RD);
  outportb(I2C_NUM_BYTES, 1);
  outportb(I2C_CTRL_STAT, 0);
  while ( inportb(I2C_CTRL_STAT ) & 0x01 );      /* wait for BUSY to be cleared */
  outportb(I2C_ADDRESS, 0xA1);                   /* reset data buffer pointer */
  return(inportb(I2C_DATA));
}

unsigned char eepromRanReadByte(unsigned char addr)        /* read a byte at some random address */
{
  eepromSetAddr(addr);
  return(eepromSeqReadByte());
}

void eepromRanReadBytes(unsigned char addr, unsigned char data, unsigned nbytes) /* read several bytes */
{
  int i;

  eepromSetAddr(addr);
  for (i = 0; i < nbytes; i++) {
    *(data+i) = eepromSeqReadByte();
  }
}

void eepromWriteByte(unsigned char addr, unsigned char data)/* write a byte at some random address */
{
  outportb(I2C_ADDRESS, EEPROM_ADDR_WR);
  outportb(I2C_NUM_BYTES, 2);
  outportb(I2C_DATA, addr);
  outportb(I2C_DATA, data);
  outportb(I2C_CTRL_STAT, 0);
  while( inport(I2C_CTRL_STAT) & 0x1 );          /* wait for BUSY to be cleared */
  do {                                           /* wait for EEPROM-write to complete */
    outportb(I2C_NUM_BYTES, 0);
    outportb(I2C_CTRL_STAT, 0);
```

```
    while( inportb(I2C_CTRL_STAT) & 0x01 );
  } while( inportb(I2C_CTRL_STAT) & 0x02 );
}

void eepromWriteBytes(unsigned char addr, unsigned char *data, int nbytes)  /* write several bytes */
{
  int i;

  for (i = 0; i < nbytes; i++) {
    eepromWriteByte(addr+i, *(data+i));
  }
}

void printHelp(void)
{
  printf("\
  ISA - I2C Multi-Master Demonstration\n\
  ====================================\n\
\n\
  d <addr> <nbytes> : dumps <nbytes> starting at <addr>\n\
  w <addr> <value> : writes <value> at location <addr>\n\
  r <addr> : reads from location <addr>\n\
  q : quit\n\
\n");
}

int main(void)
{
  int i;
  unsigned u;
  unsigned char buf[256];
  char cmdLine[50];
  char cmd[2];
  unsigned op1, op2;

  printHelp();

  while (1) {
    printf("> ");
    gets(cmdLine);
    sscanf(cmdLine, "%s %x %x", cmd, &op1, &op2);

    if (strcmp(cmd, "d")==0) {
      eepromSetAddr(op1);
      for (i = 0; i < op2; i++) {
        printf("%.2X  ", eepromSeqReadByte());
      }
      printf("\n");
    } else if (strcmp(cmd, "w")==0) {
      eepromWriteByte(op1, op2);
      printf("\n");
    } else if (strcmp(cmd, "r")==0) {
      printf("%.2X\n", eepromRanReadByte(op1));
    } else if (strcmp(cmd, "q")==0) {
      return 0;
    } else {
      printHelp();
    }
  }
}
```

Lit#: SXL-AN21-02

Sales and Tech Support Contact Information

For the latest contact and support information on SX devices, please visit the Scenix Semiconductor website at www.scenix.com. The site contains technical literature, local sales contacts, tech support and many other features.

SCENIX

1330 Charleston Road
Mountain View, CA 94043
Tel.: (650) 210-1500
Fax: (650) 210-8715
E-Mail: sales@scenix.com
Web Site: www.scenix.com