I2C Virtual Peripheral Implementation

Application Note 29

November 2000

1.0 Preface

This document outlines the software needed to implement I^{2}C Slave, Master and Multi-Master with the SX communications controller from Ubicom, Inc. It also describes the operation of I^{2}C and its implementation on the SX device using the concept of Virtual Peripheral. This software may be used with other Virtual Peripheral modules from Ubicom, and with your own application code, you can achieve a high performance solution with a flexible and cost-effective communications controller.

2.0 I^{2}C-Bus Concept

In modern electronic systems there are a number of peripheral ICs that have to communicate with each other and the outside world. To maximize hardware efficiency and simplify circuit design, Philips developed a simple bi-directional 2-wire, serial data (SDA) and serial clock (SCL) bus for inter-IC control. It gives an economical board level interface between different devices such as microcontrollers, DACs, ADCs, EEPROM, etc. This I^{2}C-bus supports any IC fabrication process and, with the extremely broad range of I^{2}C-compatible chips from Philips and other suppliers, it has become a worldwide industry standard proprietary control bus.

Each device is recognized by a unique address and can operate as either a receiver-only device (e.g. an LCD driver or a transmitter with the capability to both receive and send information (such as memory). Transmitters and/or receivers can operate in either master or slave mode. A master is the device, which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave.

The I^{2}C-bus is a multi-master bus. This means that more than one device that is capable of controlling the bus, can be connected to it. As masters are usually microcontrollers, let's consider the case of a data transfer between two microcontrollers connected to the I^{2}C-bus (see Figure 2-1). This highlights the master-slave and receiver-transmitter relationships to be found on the I^{2}C-bus. It should be noted that these relationships are not permanent, but only depend on the direction of data transfer at that time.

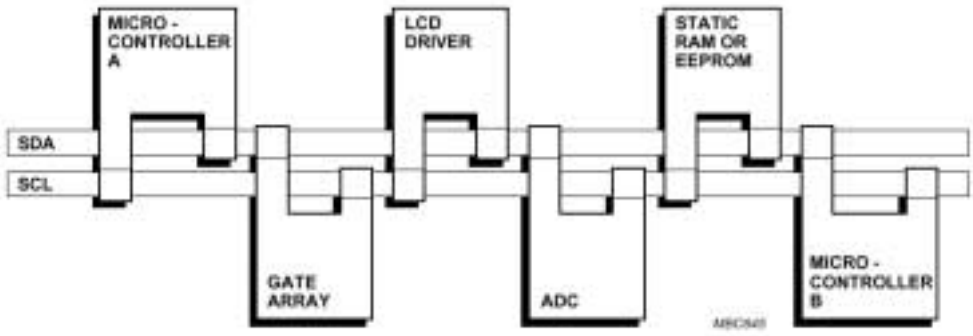

Figure 2-1. I2C-Bus Using Two Microcontrollers

The basic I^{2}C-bus, with a data transfer rate up to 100 kbits/s and 7-bit addressing, was originally introduced nearly 20 years ago.

3.0 General Description

Both SDA and SCL are bi-directional lines, connected to a positive supply voltage via a current-source or pull-up resistor. When the bus is free, both lines are high. The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function. Data on the I^2C-bus can be transferred at rates of up to 100 kbits/s in the Standard-mode, but higher rates are possible in faster modes. The number of interfaces connected to the bus is solely dependent on the bus capacitance limit of 400pF.

First lets clarify some of the terms used when talking about an I^2C-bus:

Master	The device which initiates a transfer, generates clock signals and terminates a transfer.
Slave	The device addressed by the master.
Multi-Master	More than one master can attempt to control the bus at the same time without corrupting the message.
Arbitration	The procedure to ensure if more than one master simultaneously try to control the bus, only one is allowed to do so and the sinning message is not corrupted.
Clock Synchronization	Procedure to synchronize the clock or more devices.

3.1 Start and Stop Conditions

Within the procedure of the I^2C-bus, unique situations arise which are defined as *start (S)* and *stop (P)* conditions. A high to low transition on the SDA line while SCL is high is one such unique case. This situation indicates a *start* condition. A low to high transition on the SDA line while SCL is high defines a *stop* condition (see Figure 3-1). *Start* and *stop* conditions are always generated by the master. The bus is considered to be busy after the *start* condition, and it is considered to be free again a certain time after the *stop* condition. Detection of *start* and *stop* conditions by devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the SDA line at least twice per clock period to sense the transition.

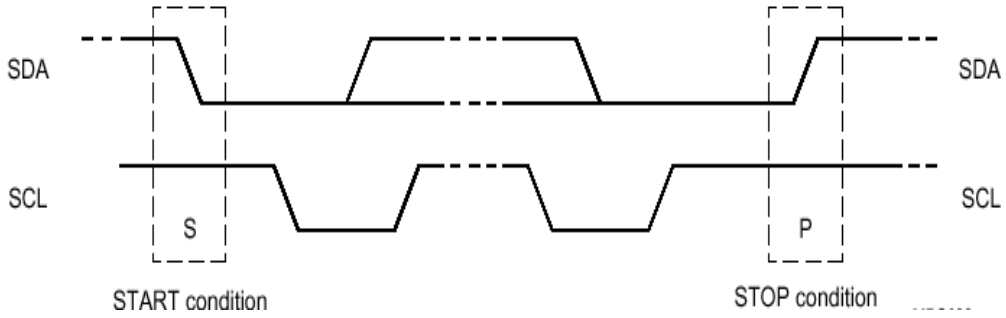

Figure 3-1. I^2C, *start* and *stop* Conditions

3.2 Transferring Data

This section contains a short description of the I^2C dataflow, acknowledge, clock synchronization, and arbitration. This is the also considered as the basic of the I^2C standard.

3.2.1 Byte format

Every byte put on the SDA line must be 8-bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an acknowledge bit.

Data is transferred with the most significant bit first (see Figure 3-2).

If a slave can't receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL low to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.

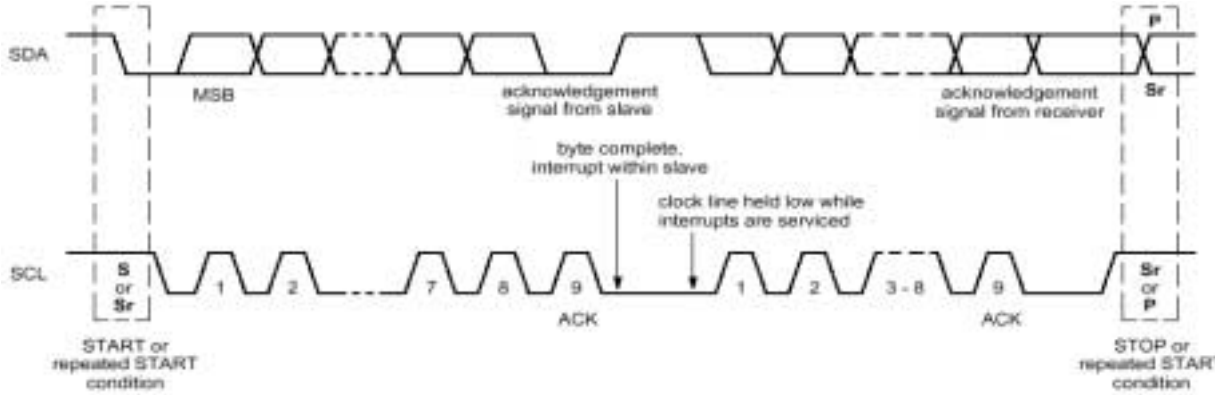

Figure 3-2. Data Transfer on the I^2C-bus

3.2.2 Acknowledge

Data transfers with acknowledge is obligatory. The transmitter releases the SDA line during the acknowledge clock pulse, and this signal is then pulled high (pull-up). The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable low during the high period of this clock pulse (see Figure 3-3).

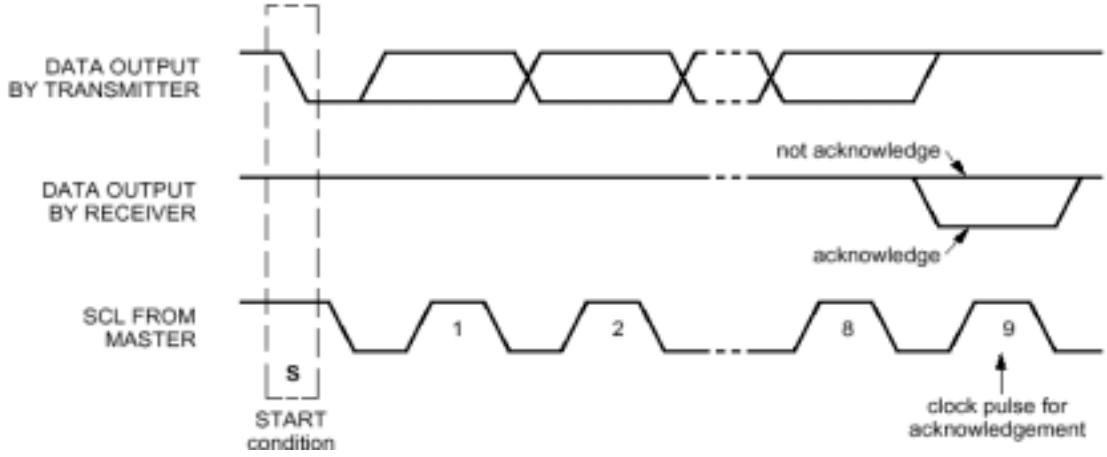

Figure 3-3. Acknowledge on the I^{2}C-bus

Of course, set-up and hold times must also be taken into account. When a slave does not acknowledge the address, the slave has to leave the data line high (for example, it's unable to receive or transmit because it's performing some real-time function). The master can then generate either a *stop* condition to abort the transfer, or a repeated *start* condition to start a new transfer. If a slave-receiver does acknowledge the slave address but some time later in the transfer cannot receive any more data bytes, the master must again abort the transfer. This is indicated by the slave generating the not-acknowledge on the first byte to follow. The slave leaves the data line high and the master generates a *stop* or a repeated *start* condition.

If a master-receiver is involved in a transfer, it must signal the end of data to the slave- transmitter by not generating acknowledge on the last byte that was clocked out of the slave. The slave-transmitter must release the data line to allow the master to generate a *stop* or repeated *start* condition.

3.2.3 Clock Synchronization and Arbitration

All masters generate their own clock on the SCL line to transfer messages on the I^2C-bus. Data is only valid during the high period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place. Clock synchronization is performed using the wired-AND connection of I^2C interfaces to the SCL line. This means that a high to low transition on the SCL line will cause the devices concerned to start counting off their low period and, once a device clock has gone low, it will hold the SCL line in that state until the clock high state is reached (see Figure 3-4).

However, the low to high transition of this clock may not change the state of the SCL line if another clock is still within its low period. The SCL line will therefore be held low by the device with the longest low period. Devices with shorter low periods enter a high wait-state during this time. When all devices concerned have counted off their low period, the clock line will be released and go high. There will then be no difference between the device clocks and the state of the SCL line, and all the devices will start counting their high periods. The first device to complete its high period will again pull the SCL line low. In this way, a synchronized SCL clock is generated with its low period determined by the device with the longest clock low period, and its high period determined by the one with the shortest clock high period.

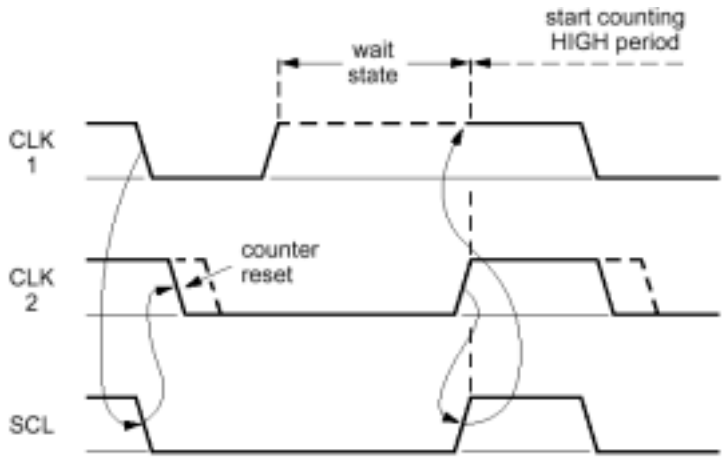

Figure 3-4. Clock Synchronization During the Arbitration Procedure

A master may start a transfer only if the bus is free. Two or more masters may generate a *start* condition within the minimum hold time of the *start* condition, which results in a defined *start* condition to the bus. Arbitration takes place on the SDA line, while the SCL line is at the high level; the master that transmits a high level while another master is transmitting a low level, will switch off its data output stage, because the level on the bus doesn't correspond to its own level.

Arbitration can continue for many bits. The first stage is comparison of the address bits. If the masters are each trying to address the same device, arbitration continues with comparison of the data-bits if they are master-transmitter, or acknowledge-bits if they are master-receiver. Since address and data information on the I^2C-bus, are determined by the winning master, no information is lost during the arbitration process. A master that loses the arbitration can generate clock pulses until the end of the byte in which it loses the arbitration. If a master also incorporates a slave function and it loses arbitration during the addressing stage, it's possible that the winning master is trying to address it. The losing master must therefore switch over immediately to its slave mode. Figure 3-5 shows the arbitration procedure for two masters.

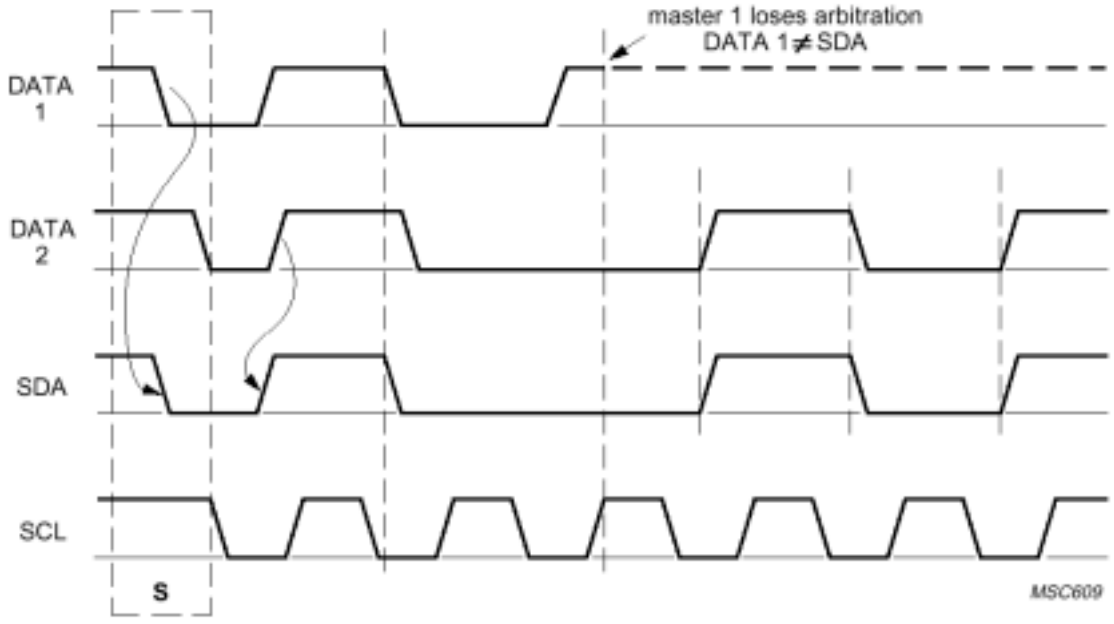

Figure 3-5. I^2C Arbitration Procedure of Two Masters

Of course, more may be involved (depending on how many masters are connected to the bus). The moment there is a difference between the internal data level of the master generating DATA 1 and the actual level on the SDA line; its data output is switched off, which means that a high output level is then connected to the bus. This will not affect the data transfer initiated by the winning master. Since control of the I^2C-bus is decided solely on the address or master code and data sent by competing masters, there is no central master, nor any order of priority on the bus. Special attention must be paid if, during a serial transfer, the arbitration procedure is still in progress at the moment when a repeated *start* condition or a *stop* condition is transmitted to the I^2C-bus. If it's possible for such a situation to occur, the masters involved must send this repeated *start* condition or *stop* condition at the same position in the format frame.

In other words, arbitration is not allowed between:

- A repeated *start* condition and a data bit
- A *stop* condition and a data bit
- A repeated *start* condition and a *stop* condition.

Slaves are not involved in the arbitration procedure.

For more information on the I^2C specification, visit:

http://www-us.semiconductors.philips.com/i2c/facts/#specification

http://www.philips.semiconductors.com/products/all_other.html (#33, Basic I^2C specification)

4.0 Ubicom I²C Virtual Peripheral Implementation

Implementation of the I²C specification on SX devices is achieved by using the available I²C Virtual Peripheral modules. There are three I²C Virtual Peripheral modules, implemented according to "vp_guide_1.02.src", available at www.scenix.com:

- I²C slaveImp i2cs.src
- I²C master i2cm.src
- I²C multi-master i2cmm.src

By using these Virtual Peripheral modules the user has the ability to implement a variety of I²C peripheral combinations. This is similar to what is done with hardware implemented I²C peripherals. The advantage of using the Ubicom Virtual Peripheral concept is ability to change these software peripherals to suit the users exact requirements, without paying more in hardware. This flexibility means that many different I²C solutions can be achieved using the same SX device.

The hardware required to interface the I²C bus specification to the SX device is very simple. All that is required is 2 pull-up (4.7K when operating at 5V) resistors, one on each of the I²C bus lines. For demo description, see Section 6.0.

4.1 I²C Slave Virtual Peripheral (iscs.src)

The I²C slave Virtual Peripheral, allows any SX device with the interface required to operate as an I²C slave. The way, in which this peripheral has been written, has been with the intent to give the user simple access subroutines to call and need not worry about the inner workings of the peripheral code.

4.1.1 Using the I²C Slave Virtual Peripheral

In order to use the I²C Slave Virtual Peripheral all that is required is to follow the steps listed below.

1. Obtain the latest version of i2cs.src.
2. Modify the pin definitions of the SDA (`i2csSda`) and SCL (`i2csScl`) lines, and the port assignment (`i2csPort`) for rX, to suit your application (where X is the port name, see datasheet for the appropriate SX device).
3. Ensure the correct port is getting updated within the interrupt service routine, update port section.
4. Set the I²C slave-address. The SX slave will respond to this address only.
5. Set the correct string (data) that can be read by a master in `i2csString`. Max 16 words (or write your own main routine to put data to the `i2csString`).
6. The I²C slave peripheral is now ready to be used.

The current I²C slave mainline code was written as a part of a demo, which interfaces to the I²C master. This code can be changed as required to meet your application requirements. At present all the slave mainline code does, is to respond by putting the byte of data on the I²C bus when a master asks for it from the slave. This is exactly how an I²C EEPROM device would operate.

To receive data from an I²C master the following steps need to be followed:

1. Check the i2csRxFlag. If it is true then the I²C slave has received some data.
2. Check the `i2csBeingReadFlag`. If false then the master is trying to send data to the I²C slave.
3. The byte will automatically be received into the `i2csDataInRegister`. The data can now be read out of this register and processed as required.
4. Clear the i2csRxFlag to indicate that the data has been processed.

Sending data to an I²C master:

1. Load the data you wish to send to the I²C master into the `i2csDataOut` register.
2. Set the `i2csDataValid` flag so that the I²C slave state machine knows to send the data contained in `i2csDataOut`.
3. The next time an I²C master attempts a read from this slave, the data contained in the `i2csDataOut` register will be automatically sent to the master.
4. Check the i2csDataNeeded flag. This flag is set if the I²C master has tried to do a multiple read on this slave and there is no valid data present in the i2csDataOut register. The slave will hold the clock line low until the i2csDataValid flag is set, indicating valid data.

4.1.2 Function description

The only function implemented on the I²C Slave is the i2csInit. This subroutine should be called to initialize the Virtual Peripheral. It initializes the variables that are critical to the operation of the I²C slave state machine.

4.1.3 I²C Slave Virtual Peripheral Description

The I²C slave Virtual Peripheral uses a state machine to change between all the required states within any I²C operation. This state machine operates solely within the timer interrupt service routine. Since the state machine can be executed asynchronously, it is also possible to run the code from within the mainline if required. This enables the user to select when to run the slave state machine and possibly increase transmission speed. The timer interrupt service routine determines how often the state machine is executed. The frequency of execution, influences the maximum transfer rate to and from the slave. Currently the interrupt frequency is set so that the slave can run at 100 kHz. To further increase the performance of the slave state machine, it is possible to combine two states together, which need not be separate. For example, a state that only performs processing, may just as well be executed at the end of the previous state. This would result in better state machine efficiency and would enable a reduction in the interrupt frequency. Description of each of the states in the I²C slave interrupt service routine state machine is as described in the following sections.

4.1.3.1 I²C Slave Interrupt-Driven State Machine

The current state of this state machine is stored in the registers `i2csState` and `i2csSubState`. These are the discrete states of this state machine.

4.1.3.2 i2csIdle

`i2csIdle` is the state used when no start condition has occurred and the device has not been addressed. The I²C slave will stay in this state until the state is changed by the independent subroutine `i2csGetStartStop`.

4.1.3.3 i2csWaitForSclLow

This state simply waits until SCL goes low. It is entered once `i2csGetStartStop` encounters a start condition. Once SCL goes low, the state is incremented.

4.1.3.4 i2csGetAddress

This state prepares the `i2csReadByte` routine to get 8-bits of data. The state is incremented to `i2csReadByte` before exiting.

4.1.3.5 i2csReadByte

This state reads a byte of data. `i2csBitCount` needs to be loaded with #8 before entering this state, or it will not increment to the next state after 8 bits of data have been loaded.

4.1.3.6 i2csProcessAddress

This state simply performs a quick calculation to figure out if this slave was just addressed. If this slave was addressed, then this state prepares to either read data or write data, depending on bit one of the first byte received. If it was not, then it changes the state back to `i2csIdle`.

4.1.3.7 i2csSendAck

This state outputs an ACK pulse, to tell the master that data was received correctly. It pulls SDA low while SCL is pulsed high and low by the master.

4.1.3.8 i2csReadData

This state prepares `i2csReadByte` to receive 8 bits of data. Before it exits the state is incremented.

4.1.3.9 i2csMakeIdle

This state puts the I²C slave back into idle mode.

4.1.3.10 i2csProcessData

This state processes a byte of data that was just received. It moves the byte that was just received into the `i2csDataIn` register, and sets the `i2csEventFlag` to indicate an I²C Slave event and sets the `i2csRxFlag` to indicate that the slave received a byte of data.

4.1.3.11 i2csSendData

This state prepares the `i2csWriteByte` state to send the 8-bits of data in the `i2csDataOut` register.

4.1.3.12 i2csWriteByte

This state outputs a byte of data, clocked out by the SCL pin. It must be prepared to send out a byte by having the `i2csByte` register loaded with valid data and having the `i2csBitCount` register loaded with #8.

4.1.3.13 i2csGetAck

This state gets an ACK from the I^2C master. If an ACK is received, the state will try to send another byte of data from the i2csDataOut register. If no ACK is received, the slave will be put back into its idle state.

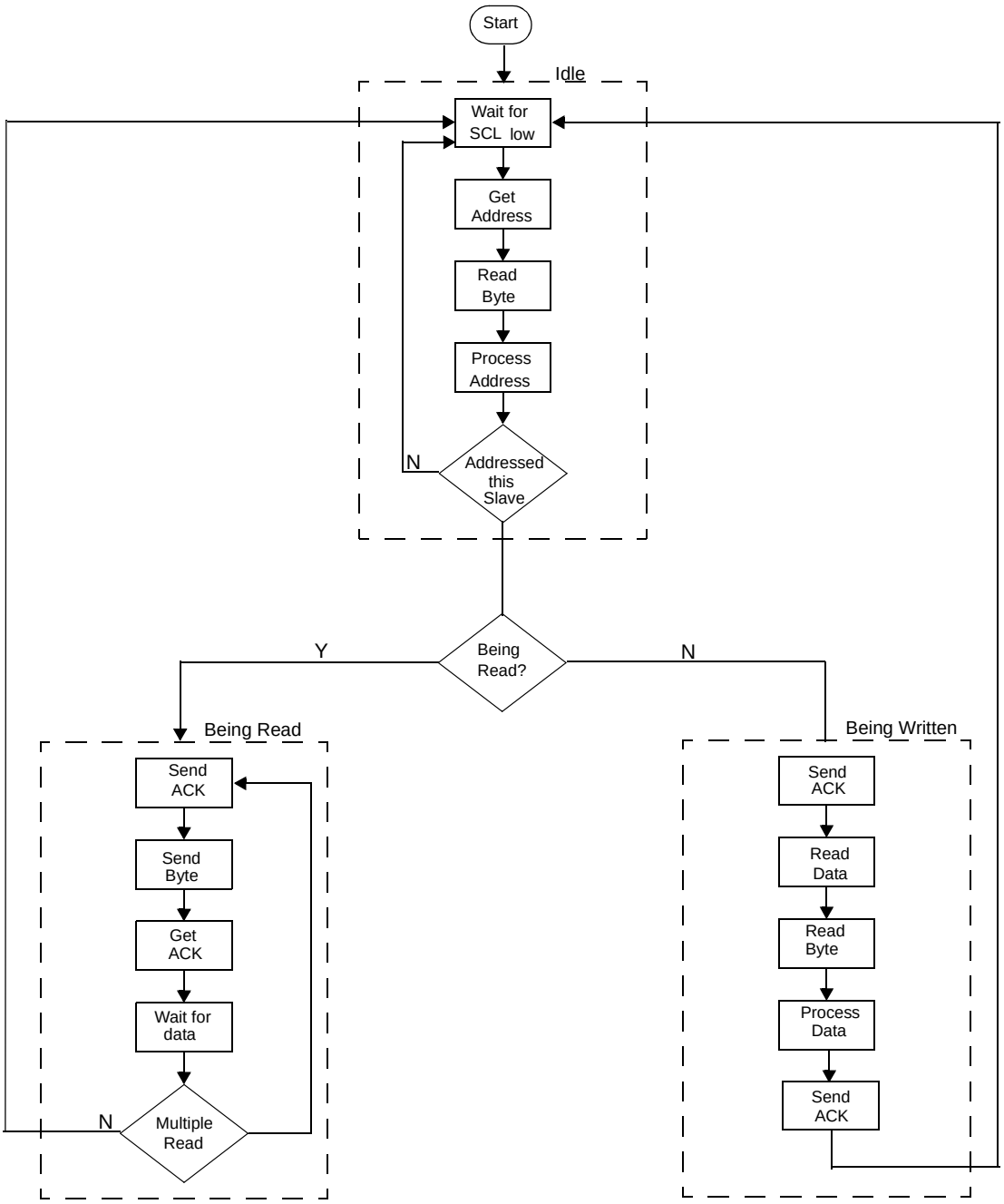

Figure 4-1. I^2C Slave State Machine

4.2 I²C Master Virtual Peripheral (i2cm.src)

The I²C master Virtual Peripheral has been written to enable the user to simply and easily operate the SX as I²C master device. The way, in which this Virtual Peripheral has been written, was with the intent to give the user simple access subroutines to call and need not worry about the inner workings of the peripheral code.

The I²C master uses the same principles used for the I²C slave. It run as a state machine in the timer driven interrupt service routine and different I²C master operations are performed by calling access subroutines from the mainline code. These subroutines are described in Section 4.2.2 and the state machine of the I²C master is discussed Section 4.2.4.

4.2.1 Using the I²C Master Virtual Peripheral

To configure the I²C master, set the port assignment to match your application needs (see Section 4.1.1). However, you are restricted to use pin 0 and 1 on the port you have assigned for I²C, but you are allowed to switch what pin is SCL and SDA. For pin and port setup, refer to the documentation for the demo board you are using. The I²C master has been set to run at 100 kHz. To change the clock speed, simply recalculate the RTCC reload value at the end of the interrupt service routine.

The mainline code written currently reads data out of a slave device at address A0h (on board EEPROM address on some demo boards). This same code can read data out of an I²C Slave Virtual Peripheral if the address is changed to 40h (or to the value set in the I²C Slave Virtual Peripheral code). The mainline code is very small and utilizes the I²C master access subroutines. These subroutines are briefly described below:

4.2.2 Function description

This section describes the interface to the I²C Master Virtual Peripheral.

4.2.2.1 i2cmSendByte

This routine sets up the I²C Master state machine to write the byte of data in the `i2cmDataBuf` register. Before entering this routine, make sure that the I²C Master state machine is in its idle state (use the `i2cmWaitNotBusy` subroutine) and that the `i2cmAddress` is loaded with the address of the slave that this byte is going to, and that `i2cmDataBuf` is loaded with the data to send. This function is actually calling `i2cmSendBytes` with argument 1.

4.2.2.2 i2cmSendBytes

This routine sets up the I²C Master state machine to write the bytes of data in the `i2cmDataBuf` register. Before entering this routine, make sure that the I²C Master state machine is in its idle state (use the `i2cmWaitNotBusy` subroutine) and that `i2cmAddress` is loaded with the address of the slave that this byte is going to, that the buffer `i2cmDataBuf` is loaded with the data to send, and that the `i2cmNumBytes` register is loaded with the number of data bytes to send.

4.2.2.3 i2cmWaitNotBusy

This routine polls the `i2cmState` register until it is not busy. It returns when the I²C master state machine becomes idle. It returns a (0) in the `w` register if the transfer appeared successful (i.e. The slave returned an ACK when addressed), and a (1) in the `w` register if the slave did not return an ACK when addressed/written.

4.2.2.4 i2cmGetByte

This routine gets one byte of data from the slave at address `i2cmAddress`. Before calling this routine, ensure that the I²C Master State Machine is in its idle state (use the `i2cmWaitNotBusy` subroutine) and that the `i2cmAddress` register is loaded with a valid address. The routine returns with the byte received in the w register and in the `i2cmDataBuf` register.

4.2.2.5 i2cmGetBytes

This routine gets i2cmNumBytes of data from the slave at address `i2cmAddress`. Before calling this routine, ensure that the I²C Master state machine is idle by using the `i2cmWaitNotBusy` subroutine, that `i2cmAddress` register contains the address of the slave to be read from, and that the `i2cmNumBytes` register is loaded with the number of bytes of data to receive. The received bytes will be contained in the buffer `i2cmDataBuf.`

4.2.2.6 i2cmInit

This subroutine should be called on start-up. It initializes the registers that are critical to the operation of the I²C Master state machine.

4.2.3 Flags and variables

The following variables and flags may be useful.

4.2.3.1 I²C Master Flags

`i2cmNack`	This bit is set if the I²C master has not received an acknowledge from the slave after the last transaction.
`i2cmRxFlag`	This flag Indicates that the number of bytes requested has been received.

4.2.3.2 Variables

`i2cmStatein`	This indicates the state that the I2CM master is currently ins.
`i2cmSubState`	This indicates the sub-state that the I2C master is currently in.
`i2cmPortBuf`	This buffer holds the current state of the I2C port direction register
`i2cmBitCount`	Indicates the number of bits left to process in read/write
`i2cmBytet`	The byte currently being written/read by the I2C master
`i2cmIndex`	The index into the I2CM buffer, used for writing
`i2cmNumBytes`	Set this register to the number of bytes to send/receive. The bytes to be sent/received will be found in registers i2cmDataBuf
`i2cmBuffer`	The buffer uses the last 7 registers of this bank to store incoming or outgoing data. This could easily be increased if required_
`i2cmAddress`	This register remembers the address of the slave to which the master will be communicating

4.2.4 I²C Master Virtual Peripheral Description

The program is event-driven state machine, running at a fixed interrupt rate. The behavior is controlled by the main program, which calls sub routines implemented in the I²C Master. These access subroutines set the state of the I²C master state machine and handle all register manipulation. Each of the possible states of the state machine is described in the sections that follow.

Currently the interrupt frequency is set so that the slave can run at 100 kHz. The user can change the main program to improve functionality or change the usage. In this example following the i2cm.src file, the main program is sending the command ‘READ’ to the slave at the address defined in `i2cmSlaveAddress`. After the command is send, it’s waiting for incoming characters and when they arrive, they are written to a buffer in bank 7 called `i2cmRecvString`. The loop is reading one byte at a time, but this could be changed to read more bytes.

4.2.4.1 I²C Master Interrupt-Driven State Machine

This is the I²C Master Interrupt Service Routine. It is an interrupt-driven state machine which allows all of the actions of the I²C Master controller to be carried out, Virtual Peripheral style, with virtually no interaction from the mainline program.

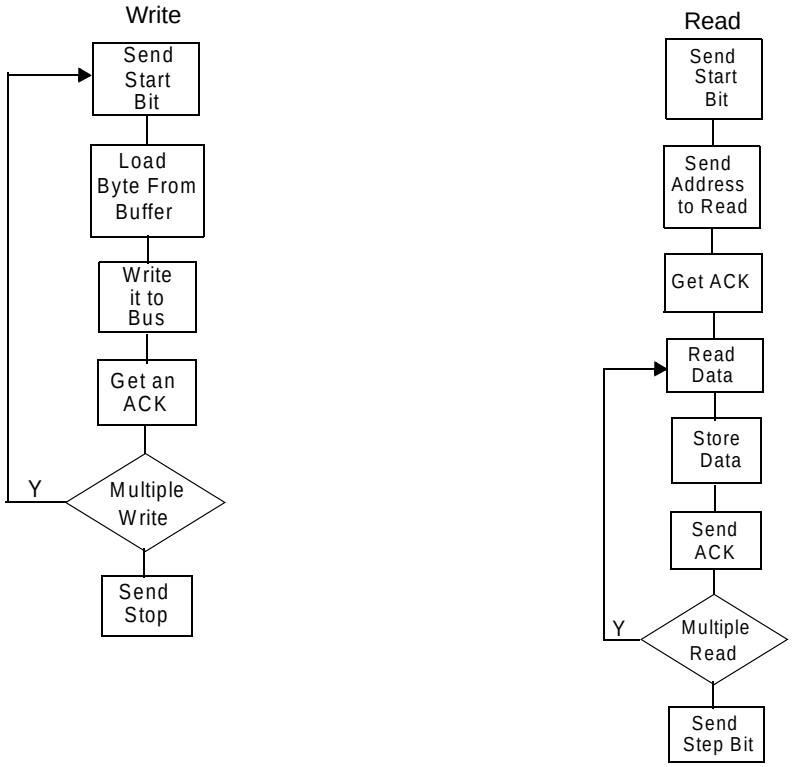

Figure 4-2. I²C Master State Machine (Multiple read/write is default "N" in demo code)

4.2.4.2 i2cmIdle

This is the state that the I²C Master is usually in when it is not in use. It just ensures that the `i2cmPortBuf` SCL and SDA are both set high

4.2.4.3 i2cmStart

When any mainline program wants to use the I²C master, it puts the master into start mode. This mode creates a start condition on the I²C bus. A start condition is created when SDA goes from high to low while SCL stays high.

4.2.4.4 i2cmStartWrite

This state performs some pre-processing which allows the `i2cmWrite` state to do its work. It sets up the bit count, gets the next piece of data from the buffer and prepares to send it.

4.2.4.5 i2cmWrite

This state writes the data in `i2cmByte` to the I²C bus

4.2.4.6 i2cmGetAck

This state gets an ACK from the slave device. If no ACK is received, the I²C Master state machine puts a stop condition on the bus and the `i2cmFlags` register is loaded to indicate that a NACK has occurred.

4.2.4.7 i2cmWriteRepeat

This state determines, after one byte of data is sent, whether or not there is another byte to be sent. If so, this state goes back to `i2cmStartWrite` and sends the next byte.

4.2.4.8 i2cmStop

This state puts a stop condition on the I²C bus and resets the state machine back to its idle state. A stop condition is when SDA goes from low to high while SCL is high.

4.2.4.9 i2cmStartRead

This state simply loads the contents of the i2cmAddress register into the i2cmByte register and sets up the i2cmWrite state to output the address of the slave to read.

4.2.4.10 i2cmReadData

This state prepares the I²C Master read routine so it can read from the slave device. It initializes the bit count, etc.

4.2.4.11 i2cmRead

This state read 8 bits of data from the slave device.

4.2.4.12 i2cmStoreByte

This state stores the byte just read into the buffer.

4.2.4.13 i2cmSendAck

This state sends an ACK if there is data left to write, and a NACK if there is no data left to write.

4.3 I²C Multi-Master Virtual Peripheral (i2cmm.src)

The I²C multi-master Virtual Peripheral has been written to enable the user to simply and easily operate the SX as an I²C master in multi-master mode. Arbitration is used to determine whether or not a master is able to use the bus at any given time.

4.3.1 Using the I²C Multi Master Virtual Peripheral

To configure the I²C multi master, set the port assignment to match your application needs (see Section 4.1.1). However, you are restricted to use pin 0 and 1 on the port you have assigned for I²C, but you are allowed to switch what pin is SCL and SDA. For pin and port setup, refer to the documentation for the demo board you are using.

Please note that you do not have to set a text string for this multi master demo.

For test purposes, it is also recommended that LEDs are connected to the pins RC.0, RC.1 and RC.2 (default selected). These can be used to verify correct operation of the multi-master system.

Once the SX device is programmed and running, it will read whatever data is stored in the first bank of the EEPROM until a null (string termination) is found. If there is more than one multi master both will start arbitration of the bus.

When you run the program in debug mode on the "SX-Key Assembler", you will see the data contained in the EEPROM being read into the last RAM bank. You can also see the text read and the slave address being read from, in a watch window (see Parallax SX Key/Blitz development system manual on how to use watch directive).

4.3.2 Function description

See description for the Master in Section 4.2.2 for a detailed description about the I²C Multi Master functions. These are the same except that the names are different; i2cm<*Name*> is here called i2cmm<*Name*>.

4.3.3 Flags and variables

The following variables and flags may be useful. Please note the similarities with the master.

4.3.3.1 I²C Multi Master Flags

`i2cmmNack`	This bit is set if the I²C master has received a NACK from the slave
`i2cmmRxFlag`	Indicates that the number of bytes requested have been received
`i2cmmLostArb`	Indicates that this master has lost arbitration
`i2cmmDoingWrite`	Used to remember whether doing a read or write when recovering from loss of arbitration
`i2cmmInControl`	Indicates that this master is in control of the I²C bus

4.3.3.2 Variables

`i2cmmIndex`	The index into the I²C Master buffer, used for writing
`i2cmmNumBytes`	The index into the I²C Master buffer, used for reading
`i2cmmStrtCtr`	Counts the number of SCL high cycles before sending a start bit onto the I²C–bus
`i2cmmBuffer`	The buffer uses the last 7 registers of this bank (pre-increments, so put I²C Master buffer here.)
`i2cmmAddress`	The address to read/write to. Loaded with the value from the i2cmmSlaveAddr at startup.
`i2cmmDataBuf`	Data buffer

4.3.4 I²C Multi Master Virtual Peripheral Description

This program relies on two aspects for correct multi-master operation. The first is an arbitration algorithm that executes every second ISR. This arbitration algorithm is used to detect when a master may communicate on the bus, or if it must wait for another master to finish. The second aspect which is important for correct operation of the I²C specification on the SX, is keeping a defined duration of the start-bit between masters. This ensures that a start may always be detected and that SX masters on the bus may synchronize their I²C clocks before beginning a transmission.

Once the SX device is programmed it will read whatever data is stored in the first bank of the EEPROM, until a null is found. Since each SX Multi Master on the bus will be trying to do this at the same time, arbitration is used. The SX masters will wait until the bus is idle, or a start bit is being put onto the bus. In either of these cases the master will begin transmission. Whichever master pulls the SDA line low first will now win arbitration as the I²C-bus is operated using open collector outputs. The master that loses arbitration will now wait a random length of time and then try again. The time that the master waits may need to be adjusted to suit your application. The way arbitration is detected is by setting a flag within the ISR, allows the user to choose what to do once a loss of arbitration is detected. At present, the master simply waits up

to 85 I^2C clock cycles before retrying the transmission. The length of this wait is totally dependant off the maximum length of any operation to take place on the I^2C-bus.

Currently the interrupt frequency is set so that the multi master can run at 97 kHz.

The formula to calculate the maximum bus speed for this

$$\text{Bus speed} = \frac{\text{Clock speed} * \text{Thread rate}}{\text{Bus over sampling rate} * \text{Cycles in thread}} = \frac{50.000.000 * 1/2}{3 * 86} = 96.899\text{kHz}$$

To make the Multi Masters bus speed run at 100kHz, the “worst-case” cycle count should have been reduced from 86 to 83 (or less).

4.4 Important Considerations

Depending on the speed you wish to operate the I^2C-bus at, it may be necessary to change the pull-up resistor values. This is also an idea if your test set-up or (your I^2C bus) are highly capacitive, i.e. when the test set-up implies "long" wires connecting two evaluation boards. If you are having trouble with the Multi Master code at 97kHz, you should consider reducing the pull-ups resistors as an option (i.e. to 2.2k).

5.0 Specifications

5.1 I^2C Slave Virtual Peripheral Requirements

Operational mode:	Random Byte access
Clock speed:	50MHz
Max MIPS usage:	32MIPS
Max cycles in each thread:	79
Bus over sampling rate:	4
Bus speed at current rate:	100kHz
ISR service rate:	400kHz
RAM usage:	10 bytes + 16bytes in i2csStringBank
Pin usage:	2 I/O pins for the I^2C bus (SDA and SCL)
RTCC setting:	Timer interrupt running every 2.6us for 100kHz- bus speed
Ubicom mnemonics:	yes
Multithreaded:	no

5.2 I^2C Master Virtual Peripheral Requirements

Operational mode:	Random byte read from slave.
Clock speed:	50MHz
Max MIPS usage:	19MIPS
Max cycles in each thread:	64 cycles
Bus over sampling rate:	3
Bus speed at current rate:	100kHz
ISR service rate:	300kHz
RAM usage:	15 bytes of RAM + 16 bytes in i2cmRecvString
Pin usage:	2 I/O pins for the I^2C bus (SDA and SCL)
RTCC setting:	Timer interrupt running every 3.3us for 100kHz- bus speed
Ubicom mnemonics:	yes
Multithreaded:	yes

5.3 I^2C Multi-Master Virtual Peripheral Requirements

Operational mode:	Random byte read from slave and arbitration.
Clock speed:	50MHz
MIPS usage:	25MIPS
Max cycles in each thread:	86 cycles
Bus over sampling rate:	3
Bus speed at current rate:	97kHz
ISR service rate:	290kHz
RAM usage:	20 + 16 bytes in i2cmmRecvString
Pin usage:	5 (SDA, SCL, and 3 LEDs)
RTCC setting:	Timer interrupt running every 1.76us for 97kHz bus speed
Ubicom mnemonics:	yes
Multithreaded:	yes (86cycles = 97kHz bus speed)

Note:The Master and Multi-Master modules are multithreaded, which means that they can be combined with other Virtual Peripheral in a complete design. See Virtual Peripheral™ Users Manual for details about multithreading.

6.0 I²C Master/Slave demo description

The master and slave Virtual Peripheral modules have been written such that if you connect two SX devices as shown in Figure 6-1, it will be possible to execute a simple I²C demo. The pin configuration has been set up such that these Virtual Peripheral modules are compatible with the Ubicom I²C/UART demo board.

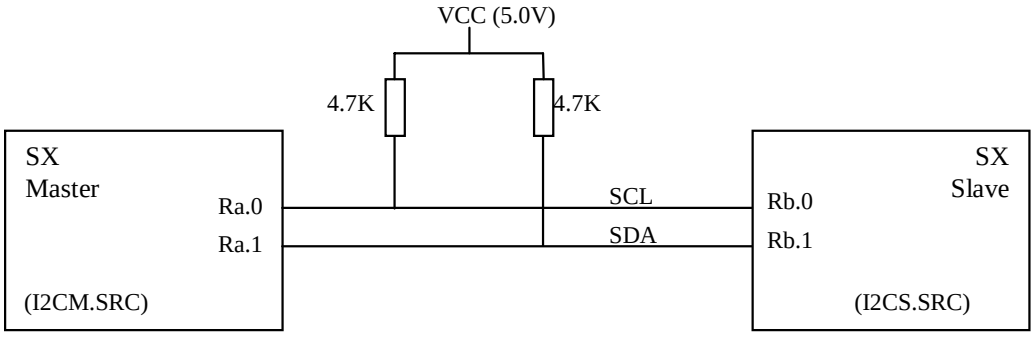

Figure 6-1. I²C Virtual Peripheral Demo Connection Diagram

In the demo program, the master SX continuously reads data from the slave SX. The data is read from the slave SX and stored in bank 7. By default this bank is loaded with the letters 'I2C SLAVE'. This data will be read by the master SX and stored in bank 7. By running the SX master in debug mode, you will see the data loaded into this bank. It is also possible to put a scope onto the I²C data lines and watch the data being transferred back and forth at 100kHz.

If you are having difficulty, ensure the slave is reset and running before the master. By running the master in debug mode, it is also possible to see if the slave SX is not acknowledging.

Figure 6-2 illustrates the data format used to read data from the SX slave. This is identical to the format used when doing a random read from an I²C EEPROM device.

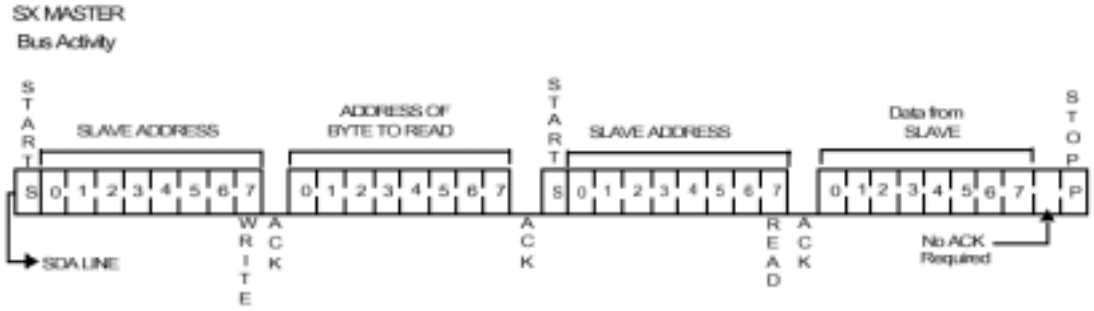

Figure 6-2. I²C Master Reading Data from Slave

7.0 Test Description

The Virtual Peripheral is tested and verified. This section describes the test environment together with a functional description of how the test was performed. For the user of the Virtual Peripheral, this section is an overview of the tests performed to do quality assurance (QA) of the Virtual Peripheral's functionality and integration.

7.1 Test Environment

During the qualification the Virtual Peripheral modules have been tested on several different demo and evaluation boards to ensure that the Virtual Peripheral are correct for different environments. The equipment used when testing the I^2C Virtual Peripheral modules are listed below:

Oscilloscope	:	Tektronix TDS 3034 (300MHz)
Demo board(s)	:	SX 28-52 Demo board
		SX Key Demo board (Parallax)
		SX Tech (Parallax)
I^2C tester	:	Micro Computer Control Cooperation
		RS232 To I^2C Host adapter with ASCII interface
		(Model MIIC-202)
SX Key assembler	:	Tested with SX 18/28 Key version 1.09 rev E/F
		Tested with SX 48/52 Key version 1.19
SASM assembler	:	Tested with SASM version 1.44.6
Debugger	:	Parallax SX Key rev. E

7.2 Functional test description

The functional test was based upon the functionality implemented in the Virtual Peripheral according to the I^2C specification (standard). There are some special considerations that affect the tests performed on the Virtual Peripheral modules:

- The I^2C Master Virtual Peripheral can only operate in a single master environment. There cannot be multiple master's connected to the bus at the same time.
- There can be several Multi-Masters connected to the I^2C bus at the same time.
- The I^2C Tester behaves as a Multi-Master and can therefore not be used together with a Master.
- All components in the test were connected to the same Ground (GND) level.

The schematic of the test environment is shown in the Figure 7-1 below:

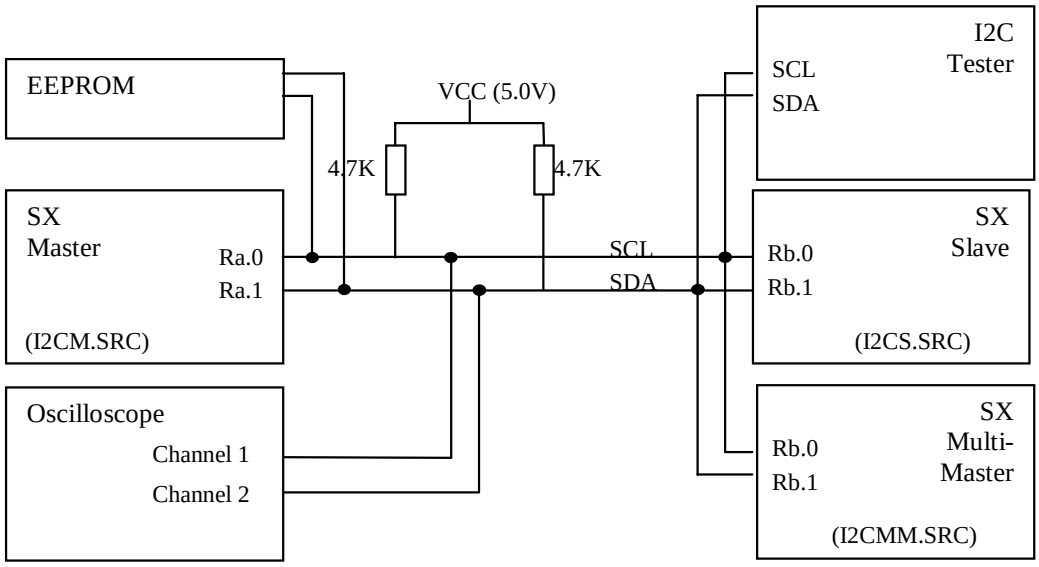

Figure 7-1. Overview of the Test Environment

Before the actual test was started we defined a set of scenarios to test to ensure that all the implemented function was tested. All functions could be tested with fewer scenarios, but there was added a couple of scenarios to test the integrity between Virtual Peripheral modules, the different scenarios used during the test are listed below:

1. I^2C Test -- I^2C Slave
2. I^2C Master -- I^2C EEPROM
3. I^2C Master -- I^2C Slave
4. I^2C Multi-Master -- I^2C EEPROM
5. I^2C Multi-Master -- I^2C Slave
6. I^2C Multi-Master -- I^2C Multi-Master -- EEPROM
7. I^2C Multi-Master -- I^2C Tester -- EEPROM

7.2.1 Important Test Considerations

The test setup included wiring between demo boards, and also from the I^2C tester and the oscilloscope; therefore additional pull-up resistors were needed.

In the description of the different scenarios, filenames have been used to show which code was programmed into the SX device, and these filenames indicates if the code is slave, master or multi-master:

i2cs.src - I^2C Slave source code

i2cm.src - I^2C Master source code

i2cmm.src - I^2C Multi-Master source code

Multi-Master was operated at 97kHz.

During the test the assemblers used were the SX-Key assembler (from Parallax, because of the debug possibilities) and the Ubicom Assembler (SASM).

A digital sampling oscilloscope was common for all the tests, and used to check the bit stream to ensure that the start/stop conditions and the acknowledge signal was according to the I^2C specifications.

7.2.2 Scenario: I²C Tester -- I²C Slave

The purpose of the scenario was to test the I²C slave. We used the I²C tester to read from the slave and check for correct operation.

1. The I²C tester was connected to the SX 28/52 Demo Board.
2. The i2cs.src was loaded into SX key assembler.
3. The source code was modified for the assembler and pins used for SCL and SDA
4. The debugger was started.
5. A terminal window was started to communicate with the I²C tester.
6. The slave receive address on the I²C tester was set to 40h same as default address on the I²C Slave. Additional I²C tester settings: Echo on, ASCII mode and open bus connection.
7. Value 00h was written to the I²C slave. This should set the internal address to 0.
8. One byte was read back from the I²C slave.
9. Value 01h was written to the slave. This should set the internal address to 1
10. And the next byte was read from the slave. The internal address was incremented until we were at the end of the string.

The string read back from the slave was verified to be “I2C SLAVE”.

7.2.3 Scenario: I²C Master -- I²C EEPROM

The purpose of this scenario was to test the I²C master for correct operation.

1. The i2cm.src was loaded into SX-Key assembler.
2. The source code was modified for the assembler, and the slave receive address was set to A0h (hardware address on the EEPROM)
3. The debugger was started and the SX was reset and started.

Polling the registers in the debugger showed that the string in the EEPROM was read into `i2cmRecvString` (RAM bank 7) on the master.

7.2.4 Scenario: I²C Master -- I²C Slave

When the master was verified functional, we wanted to verify a correct operation of both the I²C Master and I²C Slave together. We added an additional demo board for the slave device, and the i2cs.src was downloaded to it. We removed the debugger from the slave board and connected the resonator.

1. SDA, SCL and ground were connected from the slave board.
2. The i2cm.src was loaded to the SX-Key assembler, and we set the slave receive address to 40h.
3. We started the debugger, and the SX was reset and started.

Polling the registers in the debugger showed that the string in the slave device was read into `i2cmRecvString` on the master (RAM bank 7).

7.2.5 Scenario: I²C Multi Master -- I²C EEPROM

The purpose of this scenario was to test the I²C Multi Master.

1. The i2cmm.src was loaded into SX-Key assembler.
2. The source code was modified for the SX-Key assembler, and the slave receive address was set to A0h (hardware address on the EEPROM)
3. The debugger was started, and the SX was reset and started.

Polling the registers in the debugger showed that the string in the slave device was read into `i2cmmRecvString` on the master (RAM bank 7).

7.2.6 Scenario: I²C Multi Master -- I²C Slave

This scenario is an extension of the previous scenario. When the Multi Master was verified, we wanted to verify a correct operation of both the I²C Multi Master and I²C Slave together.

We added an additional demo board for the slave device, which already contained the slave source code.

1. SDA, SCL and ground were connected from the slave board.
2. The i2cmm.src was loaded to the SX-Key assembler
3. The source code was modified for the SX-Key assembler.
4. The debugger was started, and the SX was reset and started.

Polling the registers in the debugger showed that the string in the slave device was read into `i2cmmRecvString` on the master (RAM bank 7).

7.2.7 Scenario: I²C Multi Master -- I²C Multi Master -- EEPROM

The Multi Master had now been tested as a single master. The purpose of this test was to verify correct arbitration in a multi master environment. We used the SX28-52 demo board, set up both the SX28AC and SX52BD as Multi Master, the on-board EEPROM was used as slave device.

1. The EEPROM contained a test string followed by a string termination (00h)
2. Both SX devices were programmed with the I²C Multi Master (i2cmm.src) and the slave address was set to A0h on both SX’s.
3. Both debuggers were started, and both SX’s were reset and started.

Polling the registers in the debuggers showed that the string in the EEPROM was read into `i2cmmRecvString` on both Multi Masters (RAM bank 7). This verifies a correct multi master operation.

7.2.8 Scenario: I²C Multi Master -- I²C Tester -- EEPROM

This scenario is an extension of the previous scenario where one of the multi masters was replaced with the I²C Tester.

1. The I²C tester was connected to the SX 28-52 demo board.
2. The i2cmm.src was loaded to the SX-Key assembler.
3. The source code was modified for the SX-Key assembler and the target SX was set to SX28. The slave receive address was modified to A0h
4. The debugger was started, and the SX was reset and started.
5. A terminal window was started to communicate with the I²C tester.
6. The slave receive address on the I²C tester was set to A0h (Hardware address on the EEPROM). Additional I²C Tester settings: Echo on, ASCII mode and open bus connection.
7. The SX was polled and the string from the EEPROM was read into the `i2cmmRecvString` and verified to be the same as in the previous test (7.2.7)
8. A new test string was written to the EEPROM with the I²C tester.
9. The SX was polled again to verify that the Multi Master read the new string into `i2cmmRecvString` (RAM bank 7).

A new string was written to the EEPROM with the I²C tester, and the registers were polled again verifying a correct multi master operation.

7.2.9 Summary

The purpose of these tests was to verify the I²C Virtual Peripheral modules and check for correct operation according to the I²C specifications. The I²C Slave, Master and Multi Master have been tested in a “single” master environment. The Multi Master has also been tested in on the I²C bus with other multi masters present, where both accessed a slave.

Below is a screenshot from a data transfer between a Multi Master and the Slave.

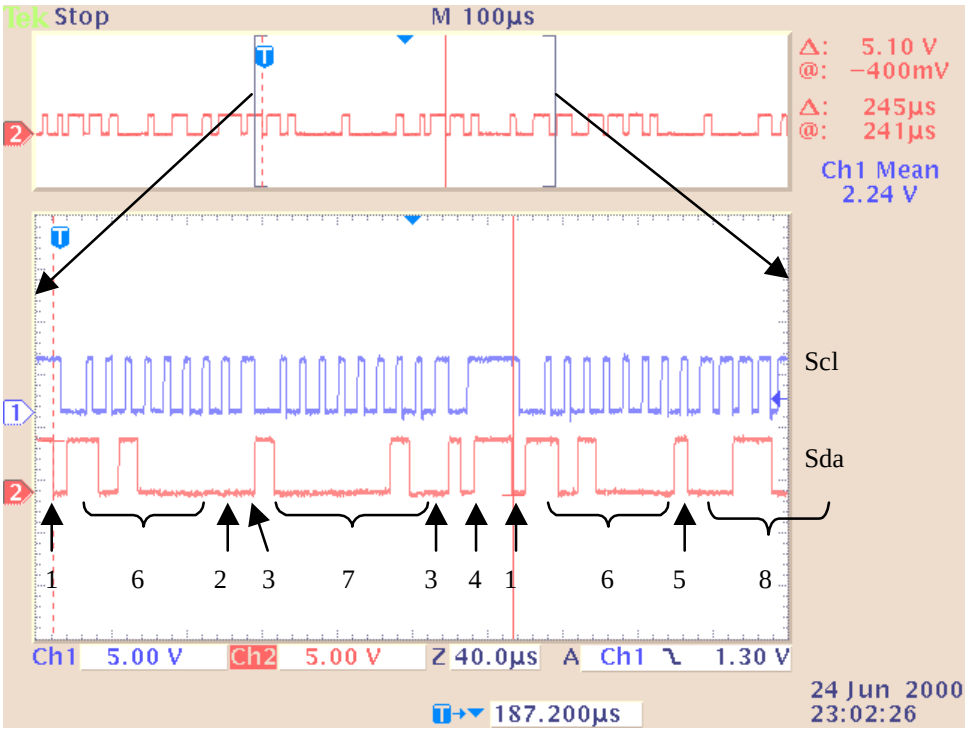

Figure 7-2. Screenshot of I²C Multi Master Reading EEPROM

The Figure 7-2 above shows a screenshot of SCL and SDA during an I²C Multi Master read from the EEPROM (and can be compared with Figure 6-2). The marks in the figure represent:

1. *Start* condition
2. Write bit (bit 7 in slave address)
3. ACK from slave
4. *Stop* condition from master
5. Read bit (bit 7 in slave address)
6. Slave address (7 bits)
7. Address of byte to read (8 bit)
8. Data from slave (1 byte, not shown)

Lit #: AN29-02

Sales and Tech Support Contact Information

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

1330 Charleston Road
Mountain View, CA 94043
Contact: Sales@ubicom.com
http://www.ubicom.com
Tel.: (650) 210-1500
Fax: (650) 210-8715