I²C™ implementation: Accessing the 24LC01 Serial EEPROM

Introduction

This application note presents programming techniques for reading from and writing to a serial EEPROM using I²C data transfer protocol. This implementation uses the Parallax demo board and takes advantage of their *SX demo* software's UART and user interface features to allow simple access to the EEPROM contents.

Additions to the Parallax SX Demo interface

Three new commands have been added to the SX demo UART interface to access the EEPROM, as follows:

1) **S**tore:	(a) **S**	-	sample the analog to digital converter ADC1 and store it in current memory address
	(b) **S xx**	-	put the hex value **xx** into current memory address
2) **V**iew:	(a) **V**	-	display all currently stored values
	(b) **V xx**	-	display the value at hex memory address **xx**
	(c) **V FF**	-	display all of the EEPROM contents
3) **E**rase:	**E**	-	write zeroes to the entire EEPROM

How the circuit and program work

Thanks to the basic hardware requirements of the I²C protocol, the circuit is very simple, using only two port pins (PortA pins 0 and 1) of the SX to provide serial access to the 24LC01 EEPROM. PortA.0 functions as the serial data clock *SCL* which provides the timing reference for data transfer to and from the EEPROM, and PortA bit 1 is *SDA*, the actual data bit stream. As on the demo board, a 10K[1] pull-up resistor should be connected from the SDA[2] pin to V_{dd} since the EEPROM's data port is open-collector.

The two main functions of the program are to read to and write from the EEPROM. Data transfers to and from the 24LC01 are composed of 8 bit data bytes which can be read/written in a random access format (i.e. one byte at a time) or in a sequential[3] format, the latter not being implemented here.

To write to the 24LC01 in random access mode, the SX must initiate the write operation by sending the EEPROM a 'START' signal, followed by a control byte 10100000b (which identifies the 24LC01 as the device to be accessed and signals that the operation to be performed is a write), followed by the address where the byte is to be written to, followed by the data byte to be written, followed by a STOP signal. It should be noted that after each byte of this sequence is sent, the program toggles the I/O status of the SDA line to read an acknowledge signal (that a byte has been received) from the EEPROM. Both the write and read sequences, as implemented here, use *acknowledge polling*. This technique sends a repeating control byte query to the EEPROM until a valid

---

™ I2C is a registered trademark of Philips Semiconductor, Inc.

[1] A value of 10K is sufficient for the data transfer rate used here. For faster rates, the pull-up may need to be reduced in order to allow successful operation. If speed is not an important issue, the external pull-up may be eliminated entirely by increasing the *t_all* bus timing delay and using the SX's internal pull-up resistor feature (see SX data sheet for programming details) on the SDA port pin.

[2] No pull-up is needed for the SCL line since it is always driven high or low by the SX

[3] The maximum number of bytes allowed during a sequential write is 8 for the 24LC01, though sequential reads have no byte count limit.

acknowledge (ACK) signal is received, before sending the address byte and then writing or reading the data byte. This is done because the EEPROM enters into an internal write cycle after each write operation, and cannot be accessed until the preceeding write process is complete, which for the 24LC01 is on the order of 10 msec. Thus by using acknowledge polling, subsequent write or read operations are executed as soon as possible after a preceeding write.

To read from the 24LC01 in random access mode, the procedure is essentially identical to the write process except that after the initial control byte and address byte have been sent and an ACK received, a new START signal is then sent followed by a read control byte (10100001b). The SDA line is then switched to an input, and data is clocked in from the EEPROM instead of sent out. The procedure is signaled as complete, as during a write, by generating a final STOP signal.

A START signal is generated by toggling the SDA line from high to low (creating a falling edge) while the SCL line is held high. A STOP signal is generated in the same manner except that SDA is toggled from low to high, thus creating a rising edge. An ACK signal is received after 8 control, address or data bits have been sent, and is considered valid if the SDA line is held low during the following (i.e. the 9th) SCL toggle cycle.

During all operations, the timing between changes in the SCL and SDA lines is a crucial factor. In this case, a generic delay time has been selected for all required START, STOP, data I/O, and ACK delays. As given, the program is capable of reading the EEPROM at approximately 200kbps[4] with the SX in turbo mode.

When calling the *I2C_write* and *I2C_read* subroutines, the program register bank must be set to the *I2C* bank. For random access mode, the address of the byte to be written/read must be pre-loaded into the *address*[5] program register, and the sequential flag *seq_flag* must be set to low. For writes, the byte to be written must be also be pre-loaded into the *data* program register, and for reads, the *data* program register will contain the value received from the EEPROM upon completion of the read procedure.

Modifications and further options

To optimize access speed to the 24LC01, the specific event and signal timings should be taken from the 24LC01 data sheet, and the appropriate reduced delay values inserted into the various bit operation subroutines. The *Bus_delay* subroutine can be accessed to produce a customized delay by loading the W register with the delay value and then calling *Bus_delay:custom*. In turbo mode each custom call will cause the following timing delay: *delay [usec] = 1/xtal[MHz] * (6 + 4 * (W-1))*, where *xtal* is the oscillator frequency in MHz and *W* is the value pre-loaded into the W register. For example, a value in W=62 will cause a 5 usec delay at 50 MHz.

Performing sequential writes and reads will also speed up the rate at which the 24LC01 can be accessed, and especially significantly increase the rate at which the 24LC01 can be written (since up to 8 bytes can be written simultaneously, reducing the need for separate internal EEPROM write delays).

To perform a **sequential write**, a specific series of steps must be followed. First the sequential flag *seq_flag* must be set high. The first byte to be written is then written as usual, but the following bytes (up to 7 more) are written by calling the write routine at the *I2C_write:sequential* entry point. Take note that *seq_flag* must be reset to low before the final byte of the group is sent, though the entry point called to write this final byte is still *I2C_write:sequential*. This generates the required stop bit to initiate the EEPROM internal write sequence.

To perform a **sequential read**, a similar series of steps must be followed. First the sequential flag *seq_flag* must be set high. The first byte to be read is read as usual, but the following bytes (up to the length of

[4]Since this implementation of the I2C access is coupled with a with a program that uses the SX's internal RTCC interrupt, the actual timing of the EEPROM access will vary per read/write, depending on how often interrupts occur during the read/write sequence.
[5]Take care to set the appropriate register bank, if needed.

the EEPROM[6]) are read by calling the read routine at the *I2C_read:sequential* entry point. Take note that *seq_flag* must be reset to low before the final byte of the group is read, though the entry point called to read this final byte is still *I2C_read:sequential*. This generates the required stop signal to end the sequential read operation.

After any write/read operation, the internal address pointer of the EEPROM is set to the byte following the last byte written or read. To read this next byte without using sequential mode, the program may call the read subroutine at the *I2C_read:current* entry point. This provides a slight increase in speed over the normal random access entry point and also eliminates the need to pre-load the *address* register before the call.

[6]In practise, the length of sequential reads can be infinite and the address pointer will simply loop around to zero after the end of the EEPROM has been reached. This can be useful for implementing wave tables and similar repeating-loop data.