intel®

Interfacing an I²C Serial EEPROM to an MCS® 96 Microcontroller

ROBIN MANELIS
Technical Marketing Engineer
Intel Automotive Operations

CHRIS BANYAI
Technical Marketing Engineer
Intel Automotive Operations

March 1995

intel®

CONTENTS

1.0   INTRODUCTION

EEPROMs (electrically erasable, programmable read-only memories) are commonly used with microcontrollers in automotive applications to provide off-chip, nonvolatile, alterable data storage. Off-chip nonvolatile memory provides flexibility to microcontroller applications for two reasons. First, in the case of a catastrophic microcontroller failure, information stored in an off-chip device can be restored. This makes off-chip EEPROMs especially useful as crash recorders. Second, a wide range of EEPROMs are available with different operating modes and memory configuration. Although microcontrollers with on-chip nonvolatile memory are also available, their operating modes and memory configurations are limited.

In automotive applications, EEPROMs are useful for storing three categories of information: fixed parameters, adaptive parameters, and historical information. Fixed parameters include items like an automobile serial/model number, tire size, axle ratio, and vehicle weight. By storing this kind of system-specific information, manufacturers can use one control module, such as an anti-lock braking system (ABS) or an air-bag controller, for a number of different vehicles. Adaptive parameters keep track of system information that changes with time and use. An example of an adaptive parameter is brake-pad wear. With this information, control algorithms can adapt to a system's changing conditions. Historical information includes the storage of parameters such as fault codes, number of ABS stops, the number of key turns, and tire air pressure. This information is useful for diagnostic and failure analysis.

This application note demonstrates how to interface an SGS-Thomson ST24C02A (EEPROM) to Intel's 87C196JR (JR) using an I^2C bus. Although the JR was chosen for this example, any Intel MCS® 96 microcontroller with a synchronous serial I/O channel could be used.

2.0   HARDWARE OVERVIEW

Two I/O pins and the synchronous serial I/O (SSIO) peripheral on the JR are used to implement the I^2C bus interface (Figure 1). The JR's I/O pins, P6.6 and P6.7, were chosen because software can dynamically switch their function from low-speed I/O (LSIO) to SSIO. In SSIO mode, P6.6 is the synchronous serial channel 1 clock (SC1) signal and P6.7 is the serial data (SD1) signal. Therefore, P6.6 is connected to the EEPROM's serial clock pin and P6.7 is connected to the EEPROM's serial data pin. When the EEPROM receives data, it acknowledges the reception by pulling the data signal low. For this reason, the JR pins are configured as open-drain with external pull-up resistors. The minimum size of the pull-up resistors was determined from the EEPROM specifications ($V_{OL} = 0.4V$, $I_{OL} = 3mA$). The following sections give an overview of the I^2C protocol, describe the EEPROM read and write operations, and demonstrate how to use the JR to implement I^2C.
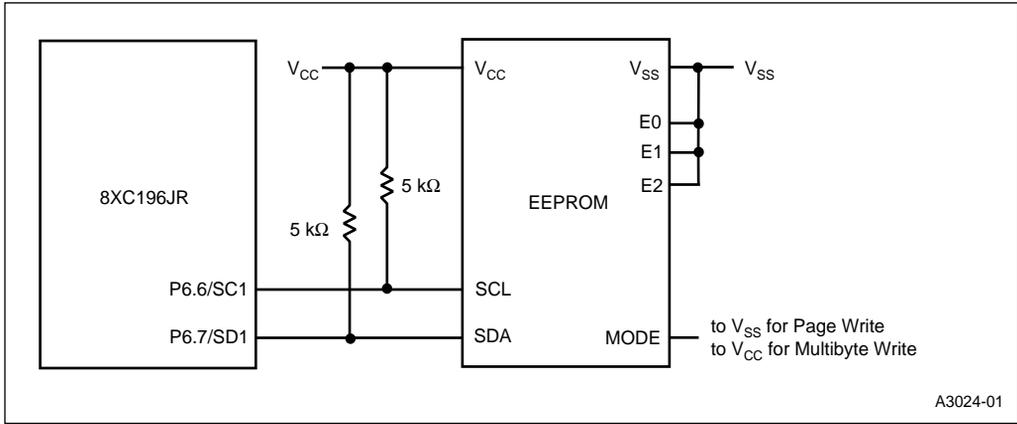
Figure 1. Circuit Diagram for Interfacing an EEPROM to an 87C196JR

2.1   I²C Protocol Overview

The I²C bus consists of two bidirectional signals: a serial data (SDA) and a serial clock (SCL), which carry information between devices connected to the I²C bus. The device that controls the data transfer is referred to as the master, and the device that the master is communicating with is the slave. In this example, the JR is the master and the EEPROM is the slave.

With this protocol, data is transferred as 8-bit bytes. Each transaction includes a start condition, one or more byte transmissions (most-significant bit first), an acknowledge condition, and a stop condition. The acknowledge condition indicates a successful data transfer. After receiving 8 bits of data, the receiver (either master or slave) creates an acknowledge condition by driving SDA low. This makes an open-drain pin configuration necessary. The SDA and SCL signals are normally high, and data is latched on the rising edge of SCL. Transitions on SDA with respect to SCL create start, stop, and data change conditions. Figure 2 illustrates these conditions.

Figure 2. Start, Stop, and Data Change Conditions

2.2 The EEPROM's Read and Write Operations

The ST24C02A EEPROM was chosen for this example because it is a typical I^{2}C EEPROM. This particular EEPROM consists of 256 bytes of memory organized as 256 × 8 bits; however, a wide range of memory sizes and configurations are available to satisfy a range of applications.
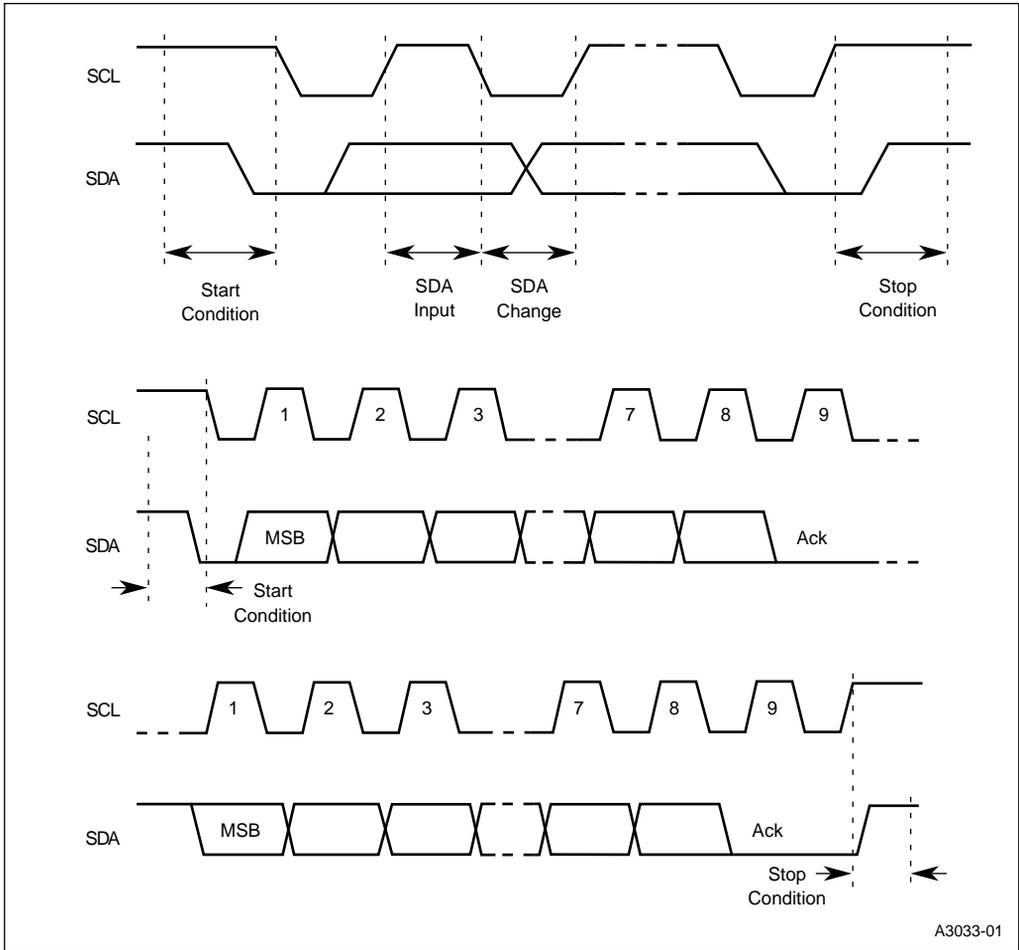
The EEPROM supports three write and four read operations. In this example, the JR as the master controls data transfers by providing the serial clock (SCL) and selecting one of the read or write operations. The JR selects one of the read or write operations by sending a device-select byte to the EEPROM.

The EEPROM's device-select byte (Figure 3) consists of a device code, a chip-enable code, and a read/write (R/$\overline{W}$) bit. The device code, which is unique to the ST24C02A, is 1010. The chip-enable code corresponds to the chip-enable pins. There are three chip-enable pins, E2–E0, making it possible to interface up to eight slave EEPROMs. In this example, these signals are strapped low; therefore, the chip enable code is 000. The read/write bit, as the name implies, selects a read or a write operation.

The EEPROM's three write operations are illustrated in Figure 4. The *byte write* operation allows the master to send one byte of data; the *multibyte write,* up to four bytes of data; and the *page write,* up to eight bytes of data. For page write operations, the data bytes must be located in the same row of memory (i.e., the five most-significant address bits, A7–A3, must be the same). The write operations are selected by clearing the read/write bit of the device-select byte and driving the EEPROM's MODE pin. The MODE pin must be tied to either V$_{CC}$ or V$_{SS}$ for byte write operations, to V$_{CC}$ for multibyte operations, and to V$_{SS}$ for page write operations. As shown in Figure 4, each write operation is initiated by a start condition, followed by a write device-select byte, and terminated by a stop condition. After each byte transfer, the receiver (which in this case is the slave) sends an acknowledge signal.
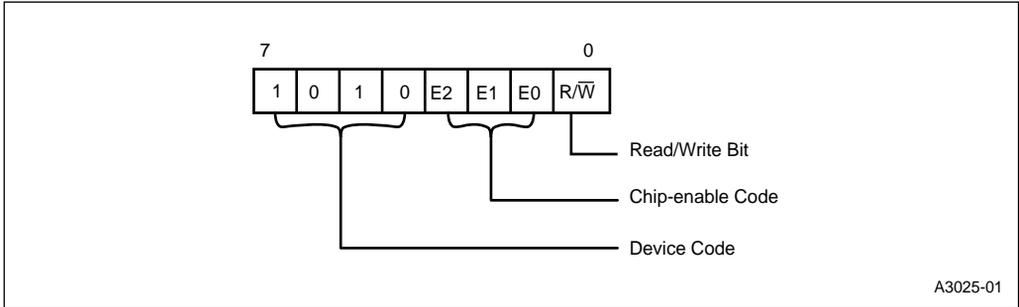
Figure 3. Device-select Byte
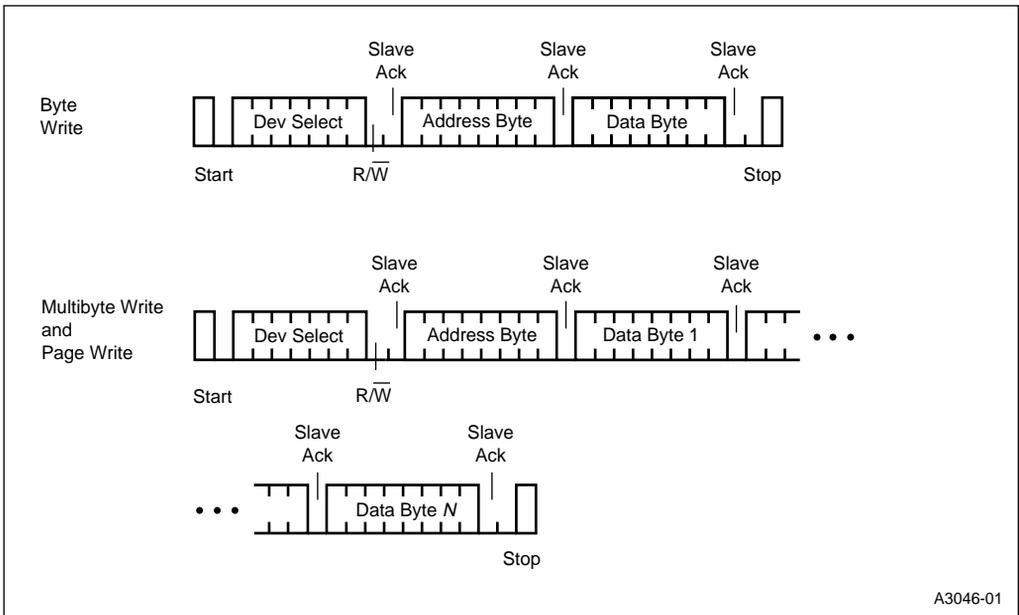
Figure 4. Write Operations

The EEPROM's read operations are illustrated in Figure 5. The read operations allow you to read one or more bytes of data at the current address or a specified address. The EE-PROM has an internal byte address counter. After a byte is read or written, the counter is incremented. The *current address read* allows the master to read one data byte at the current address, while the *sequential current address read* allows the master to read one or more data bytes starting at the current address. The *random address read* allows the master to read a data byte at a specified address, while the *sequential random address read* allows the master to read one or more data bytes starting at the specified address.

A start condition initiates a read operation. The sequence for current address read and random address read operations are different. The **current** read operations read data bytes starting at the address specified by the EEPROM's internal address counter. In these operations, after a start condition, the master sends a read device-select byte to the EEPROM. The **random** read operations read data bytes starting at the address specified by the address byte. In these operations, after a start condition, the master sends a write device-select byte, followed by an address byte (the specified address). The master then sends an additional start condition, followed by a read device-select byte. For all read operations, the EEPROM acknowledges device-select bytes and the JR acknowledges received data bytes (all but the last one). To terminate a read operation, the JR sends a stop condition without acknowledging receipt of the last data byte.

The EEPROM bit-timing waveforms and specifications relevant to this example are shown in Figure 6 and Table 1.
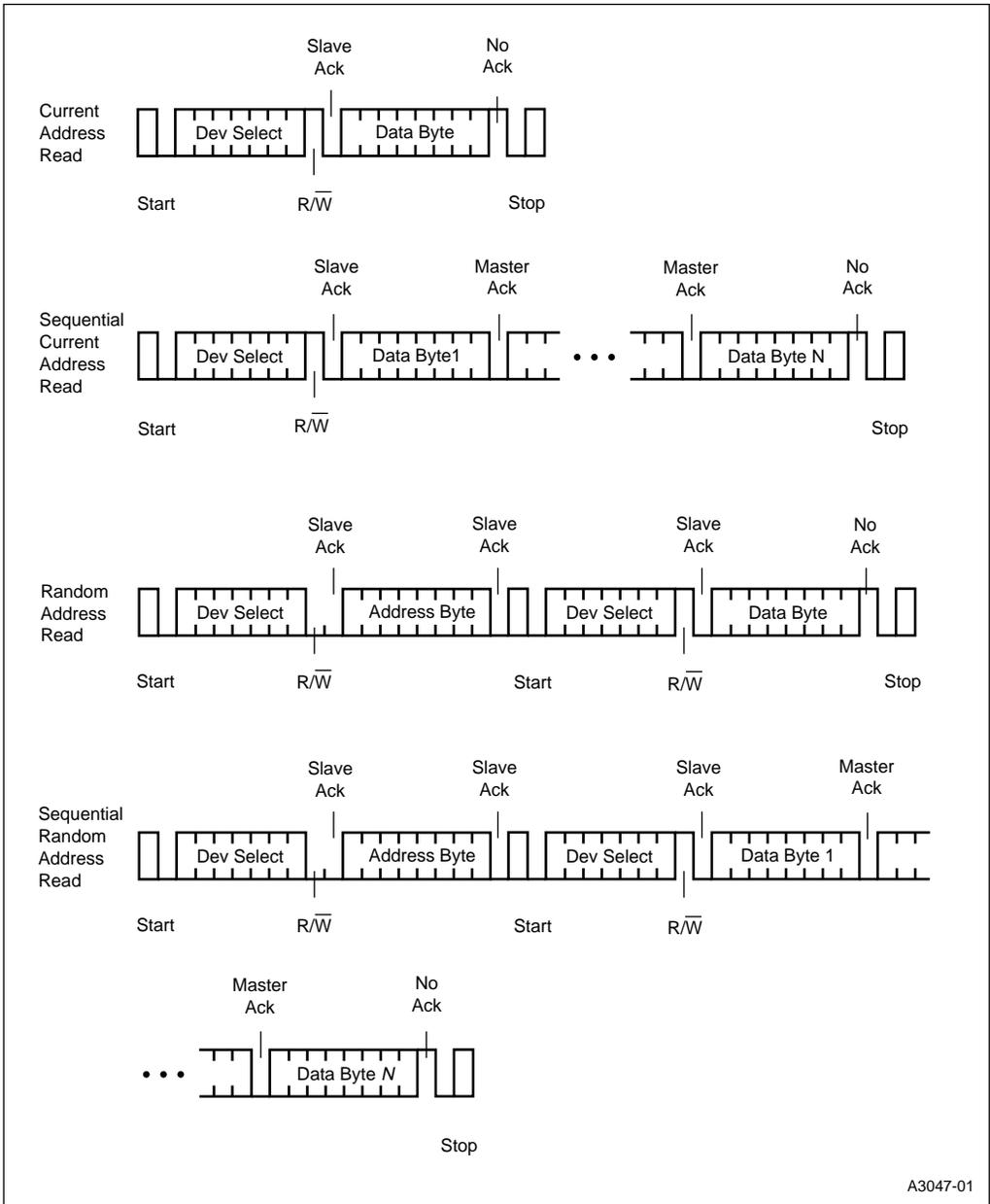
Figure 5. Read Operations

A3047-01

Figure 6. EEPROM AC Waveforms

Table 1. EEPROM AC Specifications

Symbol	Parameter	Min	Max	Unit
T_{CHCL}	Clock Pulse Width High	4		µs
T_{CLCH}	Clock Pulse Width Low	4.7		µs
T_{DLCL}	Input Low to Clock Low (start)	4		µs
T_{DXCX}	Input Transition to clock Transition	250		ns
T_{CHDH}	Clock High to Input High (stop)	4.7		µs
T_{CHDX} (1)	Clock High to Input Transition	4.7		µs
T_{CLDX}	Clock Low to Input Transition	0		µs
T_{DHDL}	Input High to Input Low (bus free)	4.7		µs
T_{CLQV}	Clock Low to Output Valid	0.3	3.5	µs
T_{CLQX}	Clock High to Output Transition	300		ns
T_W (2)	Write Time		10	ms
F_C	Clock Frequency		100	kHz

NOTES:
1. For a start condition, or following a write cycle.
2. In the multibyte write mode only, if addressed bytes are on two consecutive rows (upper five most-significant bits are the same) the maximum programming time is doubled to 20 ms.

The MCS 96 microcontroller product family operating with a 16 MHz crystal and an SSIO baud rate of 100 kHz satisfies these timings.

2.3 Using the JR to Implement I²C

Two pins on the 87C196JR (JR) are used to implement the I²C bus. These pins were chosen because their function can be controlled either by software or by the synchronous serial I/O (SSIO) peripheral. The JR's flexible port structure allows for dynamic switching between low speed I/O (LSIO) and special functions, which in this case is the SSIO clock and data signals. Start, stop, and acknowledge conditions are generated through software when the port pins are configured as LSIO. Data is transmitted or received via the SSIO peripheral when the port pins are configured for SSIO operation. Using the JR's SSIO provides a faster, more efficient method for transmitting and receiving data than the traditional "bit-banging" method. If a particular application uses the SSIO, a similar method could be implemented using the SIO peripheral in mode 0.

3.0 SOFTWARE OVERVIEW

As previously mentioned, three operations are available for writing data to the EEPROM and four operations are available for reading data from the EEPROM. These operations are implemented by the BYTE_WR, MULTI_WR, PAGE_WR, CURRENT_RD, RANDOM_RD, SEQ_CUR_RD, and SEQ_RAN_RD operation routines shown in the "Source Code Listing" on page 15.

Typically an application would use one write and one read operation. The application requirements determine which write and read operation to use (Table 2).

Table 2. Determining Which Write or Read Operation to Use

If an application requires that ...	then use the ...
a single byte of data be written to the EEPROM per control loop	byte write operation (BYTE_WR)
four bytes of data be written to the EEPROM per control loop	multibyte write operation (MULTI_WR)
eight bytes of data be written to the EEPROM per control loop	page write operation (PAGE_WR)
a single byte at the current address[†] be read from the EEPROM per control loop	current address read operation (CURRENT_RD)
a specific number of bytes at the current address[†] be read from the EEPROM per control loop	sequential current address read operation (SEQ_CUR_RD)
a single byte at a specific address be read from the EEPROM per control loop	random address read operation (RANDOM_RD)
a specific number of bytes at a specific address be read from the EEPROM per control loop	sequential random address read operation (SEQ_RAN_RD)

[†] The current address refers to the address of the EEPROM's internal address counter. This internal address counter is incremented by one every time the EEPROM is written or read.

The write and read operations consist of different combinations of start, stop, acknowledge, receive, and transmit conditions (Figures 7–9). These conditions are implemented by the START, TRANSMIT, RECEIVE, SLAVE_ACK, MASTER_ACK, NO_ACK, and STOP subroutines, which are shown following the operation routines in the "Source Code Listing" on page 15.

intel.
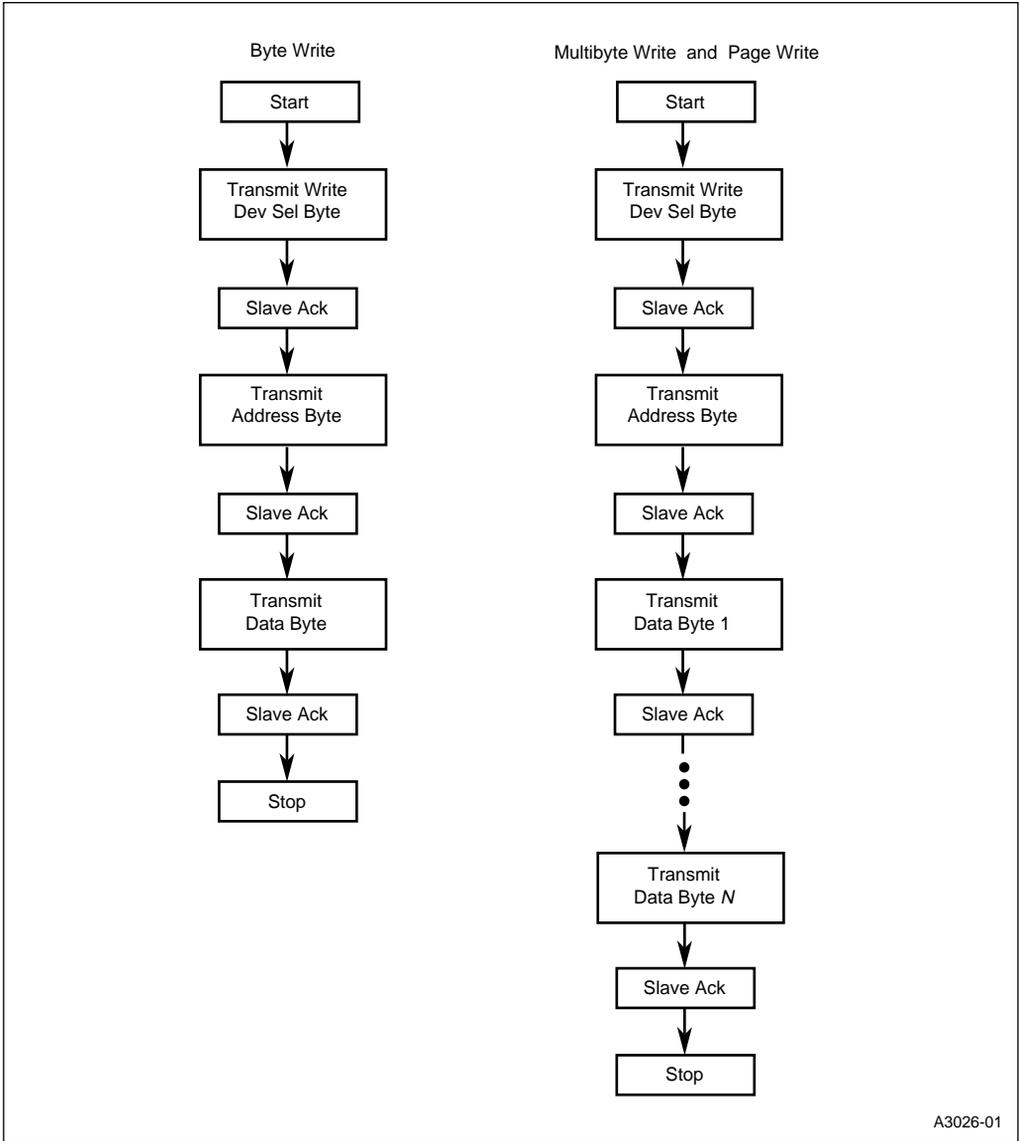
Figure 7. Write Operation Flow Charts
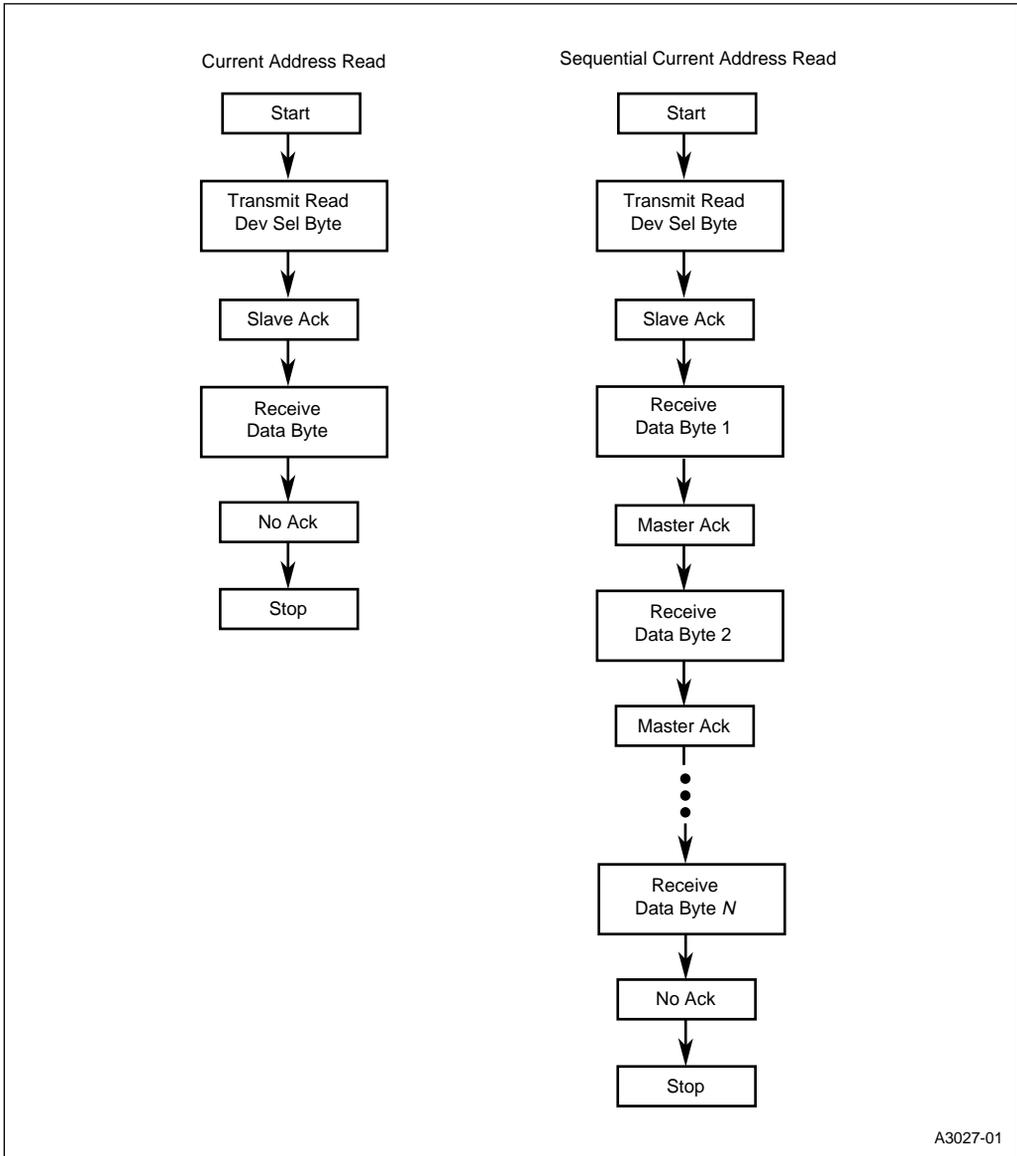
Figure 8. Read Operation Flow Charts
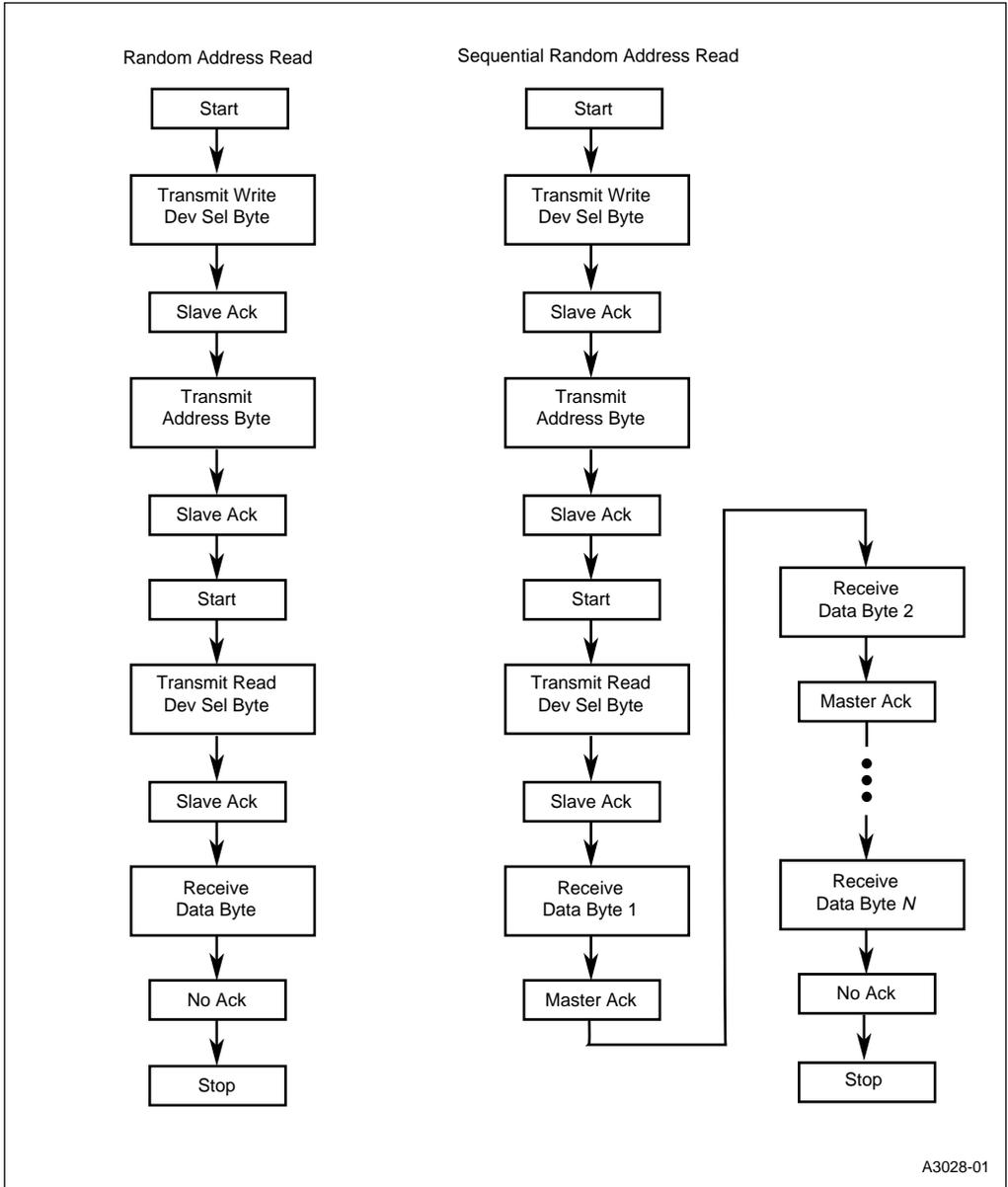
Figure 9. Read Operation Flow Charts (Continued)

The start, stop, and acknowledge (slave_ack, master_ack, and no_ack) subroutines implement the necessary conditions by configuring the pins as LSIO, then writing commands that either float the pins or drive them low. When the pins are configured as high impedance (floating), the external pull-up resistors drive the pins high. To create a start condition, software pulls the data signal low while the clock signal is high. To create a stop condition, software pulls the data signal high while the clock signal is high. To create a slave acknowledge condition, software creates a ninth clock pulse by writing to the clock pin. During this clock pulse, the data signal is put into a high impedance state. The slave (EEPROM) can then acknowledge receptions by pulling the data signal low. To create a master acknowledge condition, software creates a ninth clock pulse. During this clock pulse, the master (JR) pulls the data signal low via software to acknowledge receptions. Software creates a no acknowledge condition by simply creating a ninth clock pulse; the data signal remains unchanged. (The no acknowledge condition is used to signal the end of a sequential read operation.)

The transmit and receive subroutine configures the pins as SSIO, then initializes the transfer by writing to the SSIO control and baud rate registers. Transmissions are started by writing the transmit buffer. Receptions are started by setting the receiver enable bit in the SSIO control register. The subroutine then polls the SSIO done flag. Once the transmission or reception is complete, the processing returns to the original write or read operation.

See the source code comments for more detailed descriptions of each operation and subroutine.

3.1   Source Code Listing

```
$debug
I2C MODULE MAIN
$nolist
$include(auto.inc)      ;Equates for this program
$list
    p6pin_w1f       equ     0d7h:byte       ; (1fd7h)
    p6reg_w1f       equ     0d5h:byte       ; (1fd5h)
    p6dir_w1f       equ     0d3h:byte       ; (1fd3h)
    p6mode_w1f      equ     0d1h:byte       ; (1fd1h)
    ssio_baud_w1f   equ     0b4h:byte       ; (1fb4h)
    ssio_stcr1_w1f  equ     0b3h:byte       ; (1fb3h)
    ssio_stb1_w1f   equ     0b2h:byte       ; (1fb2h)
    ssio_stcr0_w1f  equ     0b1h:byte       ; (1fb1h)
    ssio_stb0_w1f   equ     0b0h:byte       ; (1fb0h)


;Constants defined for code readability.

    DevSelWr        equ     0a0h        ;EEPROM write select byte
    DevSelRd        equ     0a1h        ;EEPROM read select byte
    DataH           equ     080h        ;floats data line (P6.7)
    DataL           equ     07fh        ;drives data line low
    ClockH          equ     040h        ;floats clock line (P6.6)
    ClockL          equ     0bfh        ;drives clock line low
    DataClockH      equ     0c0h        ;floats clock and data line
    DataClockL      equ     03fh        ;drive clock and data line low
    ToLSIO          equ     03fh        ;configures pins P6.6 and P6.7 for LSIO
    ToSSIO          equ     0c0h        ;configures pins for SSIO

rseg at 1ch

    Addr_Pntr:      dsw     1               ;pointer for input and output data tables
    Temp_2:         dsw     1               ;counter for MULTI_WR and PAGE_WR routines
    Temp_1:         dsw     1               ;temp storage for output and input data
    Byte_Addr:      dsb     1               ;specifies the EEPROMs address
    Byte_Data:      dsb     1               ;output data for the BYTE_WR routine
    Num_Bytes:      dsb     1               ;indicates # of reads for seq rd routines
    Data_In:        dsb     8               ;input data table, used by read routines

cseg
    DATAOUT:        dcb     08h,09h ;output data table, data written to EEPROM
                    dcb     0ah,0bh ;  by the MULTI_WR and PAGE_WR routines
                    dcb     0ch,0dh ;
                    dcb     0eh,0fh ;

;Initialize the chip configuration bytes.

cseg at 2018h
    dcb 11001000b                       ;ccb
```

```
    dcb 20h                            ;not used

cseg at 201ah
    dcb 11011010b                      ;ccb1
    dcb 20h                            ;not used
```

;************************* User Code Starts Here **************************

```
cseg at 2080h
    di                                 ;disable interrupts
    dpts                               ;disable PTS
    ld sp,#9000h                       ;initialize stack
    ldb wsr,#1fh                       ;maps SSIO1 regs into lower register file
    orb p6dir_w1f,#0c0h                ;config pins P6.6 and P6.7 as open drain
```

;*************************** BYTE_WR ************************************

```
; The BYTE_WR operation routine writes one byte of data to the EEPROM.  This
; routine uses the START, TRANSMIT, SLAVE_ACK, and STOP subroutines.  START
; generates a start condition.  TRANSMIT uses the SSIO to write the
; write-device-select byte, the address byte (the EEPROM address to which the
; data byte should be written), and the data byte to the EEPROM.  Before
; calling TRANSMIT, BYTE_WR writes the transmit data to Temp_1.  After each
; write to the EEPROM, SLAVE_ACK is called.  This routine generates a ninth
; clock pulse and checks to see that the EEPROM acknowledged each write.
; STOP generates a stop condition.

; Parameters passed to this routine:
; Byte_Addr <--- the EEPROM address to which the data byte should be written
; Byte_Data <--- the data byte
```

;***

```
BYTE_WR:
    scall START                        ;generate start condition
                                       ;  data low then clock low
    ldb Temp_1,#DevSelWr
    scall TRANSMIT                     ;transmit write-select byte (A0)
    scall SLAVE_ACK                    ;check to see if EEPROM acknowledged write
                                       ;  generate 9th clock pulse

    ldb Temp_1,Byte_Addr
    scall TRANSMIT                     ;transmit byte address
    scall SLAVE_ACK                    ;check to see if EEPROM acknowledged write
                                       ;  generate 9th clock pulse

    ldb Temp_1,Byte_Data
    scall TRANSMIT                     ;transmit data byte
    scall SLAVE_ACK                    ;check to see if EEPROM acknowledged write
                                       ;  generate 9th clock pulse
```

```
    scall STOP                          ;generate stop condition
                                        ;  clock high then data high
    ret

;*************************** MULTI_WR ************************************

; The MULTI_WR operation routine writes four bytes of data to the EEPROM.
; This routine uses the START, TRANSMIT, SLAVE_ACK, and STOP subroutines.
; START generates a start condition.  TRANSMIT uses the SSIO to send the
; write-device-select byte, the starting EEPROM address to which the data
; byte should be written, and the data bytes to the EEPROM.  Before calling
; TRANSMIT, MULTI_WR writes the transmit data to Temp_1.  After each write to
; the EEPROM, SLAVE_ACK is called.  This routine generates a ninth clock
; pulse and checks to see that the EEPROM acknowledged each write.  STOP
; generates a stop condition.

; Parameters passed to this routine:
; Byte_Addr <--- the EEPROM address to which data byte should be written
; DATAOUT <--- address of data table

;****************************************************************************

MULTI_WR:
    scall START                         ;generate start condition
                                        ;  data low then clock low
    ldb Temp_1,#DevSelWr
    scall TRANSMIT                      ;transmit write-select byte (A0)
    scall SLAVE_ACK                     ;check to see if EEPROM acknowledged write
                                        ;  generate 9th clock pulse

    ldb Temp_1,Byte_Addr
    scall TRANSMIT                      ;transmit byte address
    scall SLAVE_ACK                     ;check to see if EEPROM acknowledged write
                                        ;  generate 9th clock pulse

    ldb Temp_2,#4                       ;set up counter for four bytes
    ld  Addr_Pntr,#DATAOUT              ;set up Addr_Pntr as output data pointer

NEXT1:
    ldb Temp_1,[Addr_Pntr]+             ;get output data, increment pointer
    scall TRANSMIT                      ;transmit data byte
    scall SLAVE_ACK                     ;check to see if EEPROM acknowledged write
                                        ;  generate 9th clock pulse
    djnz Temp_2,NEXT1                   ;continue until 4 bytes have been sent

    scall STOP                          ;generate stop condition
                                        ;  clock high then data high
    ret
```

```
;**************************** PAGE_WR ************************************

; The PAGE_WR operation routine writes eight bytes of data to the EEPROM.
; This routine uses the START, TRANSMIT, SLAVE_ACK, and STOP subroutines.
; START generates a start condition.  TRANSMIT uses the SSIO to send the
; write-device-select byte, the starting EEPROM address to which the data
; byte should be written, and the data bytes to the EEPROM.  Before calling
; TRANSMIT, PAGE_WR writes the transmit data to Temp_1.  After each write to
; the EEPROM, SLAVE_ACK is called.  This routine generates a ninth clock
; pulse and checks to see that the EEPROM acknowledged the write.  STOP
; generates a stop condition.

; Parameters passed to this routine:
; Byte_Addr <--- the EEPROM address to which the data byte should be written
; DATAOUT <--- address of data table

; Note: The eight data bytes that this routine will write to the EEPROM must be
; located in the same row of memory (i.e., the five most-significant address
; bits, A7-A3, must be the same).

;****************************************************************************

PAGE_WR:
    scall START                     ;generate start condition
                                    ; data low then clock low
    ldb Temp_1,#DevSelWr
    scall TRANSMIT                  ;transmit write select byte (A0)
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ; generate 9th clock pulse

    ldb Temp_1,Byte_Addr
    scall TRANSMIT                  ;transmit byte address
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ; generate 9th clock pulse

    ldb Temp_2,#8                   ;set up counter for eight bytes
    ld  Addr_Pntr,#DATAOUT          ;set up Addr_Pntr as output data pointer

NEXT2:
    ldb Temp_1,[Addr_Pntr]+         ;get output data, increment pointer
    scall TRANSMIT                  ;transmit data byte
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ; generate 9th clock pulse
    djnz Temp_2,NEXT2               ;continue until 8 bytes have been sent

    scall STOP                      ;generate stop condition
                                    ; clock high then data high
    ret
```

```
;***************************** CURRENT_RD **********************************

; The CURRENT_RD operation routine reads one byte of data from the EEPROM at
; the current address.  The current address is the address specified by the
; EEPROM's internal address counter.  This routine uses the START, TRANSMIT,
; RECEIVE, SLAVE_ACK, NO_ACK, and STOP subroutines.  START generates a start
; condition.  TRANSMIT uses the SSIO to write the read-device-select byte to
; the EEPROM.  Before calling the TRANSMIT subroutine, CURRENT_RD writes the
; device-select byte to Temp_1.  SLAVE_ACK generates a ninth clock pulse and
; checks to see that the EEPROM received the read-device-select byte.
; RECEIVE uses the SSIO to read the data byte located at the EEPROM's current
; address (specified by the its internal address counter).  After the data
; byte is read from the EEPROM, NO_ACK is called.  NO_ACK generates a ninth
; clock pulse.  STOP generates a stop condition.

; Parameters modified by this routine:
; [Data_In] <--- byte read from the EEPROM

;****************************************************************************

CURRENT_RD:
    scall START                      ;generate start condition
                                     ;  data low then clock low
    ldb Temp_1,#DevSelRd
    scall TRANSMIT                   ;transmit read select byte (A1)
    scall SLAVE_ACK                  ;check to see if EEPROM acknowledged write
                                     ;  generate 9th clock pulse

    scall RECEIVE                    ;receive data byte
    stb ssio_stb1_w1f,Data_In        ;save the recieved data byte
    scall NO_ACK                     ;generate 9th clock pulse

    scall STOP                       ;generate stop condition
                                     ;  clock high then data high
    ret

;***************************** RANDOM_RD ***********************************

; The RANDOM_RD operation routine reads one byte of data from the EEPROM at
; the address specified by the address byte.  This routine uses the START,
; TRANSMIT, RECEIVE, SLAVE_ACK, NO_ACK and STOP subroutines.  START generates
; a start condition.  TRANSMIT uses the SSIO to write the read-device-select
; byte and the address byte to the EEPROM.  Before calling TRANSMIT,
; RANDOM_RD writes the transmit data to Temp_1.  SLAVE_ACK generates a ninth
; clock pulse and checks to see that the EEPROM received the
; read-device-select byte.  RECEIVE uses the SSIO to read the data byte
; located at the specifed EEPROM address.  After the data byte is read from
; the EEPROM, NO_ACK is called.  NO_ACK generates a ninth clock pulse.  STOP
; generates a stop condition.
```

```
; Parameters passed to this routine:
; Byte_Addr <--- EEPROM address of the data byte to be read

; Parameters modified by this routine:
; [Data_In] <--- byte read from the EEPROM

;******************************************************************************

RANDOM_RD:
    scall START                     ;generate start condition
                                    ;  data low then clock low
    ldb Temp_1,#DevSelWr
    scall TRANSMIT                  ;transmit write-select byte (A0)
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ;  generate 9th clock pulse

    ldb Temp_1,Byte_Addr
    scall TRANSMIT                  ;transmit byte address
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ;  generate 9th clock pulse

    scall START                     ;generate new start condition
    ldb Temp_1,#DevSelRd            ;load read-select byte
    scall TRANSMIT                  ;transmit read-select byte (A1)
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ;  generate 9th clock pulse
    scall RECEIVE                   ;receive data byte
    stb ssio_stb1_w1f,Data_In       ;save the received data
    scall NO_ACK                    ;generate 9th clock pulse

    scall STOP                      ;generate stop condition
                                    ;  clock high then data high
    ret

;*************************** SEQ_CUR_RD **********************************

; The SEQ_CUR_RD operation routine reads a specified number of bytes of data
; from the EEPROM at the current address.  The current address is the address
; specified by the EEPROM's internal address counter.  This routine uses the
; START, TRANSMIT, RECEIVE, SLAVE_ACK, MASTER_ACK, NO_ACK and STOP
; subroutines.  START generates a start condition.  TRANSMIT uses the SSIO to
; write the read-device-select byte to the EEPROM.  Before calling TRANSMIT,
; CURRENT_RD writes the transmit byte to Temp_1.  SLAVE_ACK generates a ninth
; clock pulse and checks to see that the EEPROM received the
; read-device-select byte.  RECEIVE uses the SSIO to read the data bytes
; starting at the EEPROM's current address (specified by its internal address
; counter).  After each data byte (except the last byte) is read from the
; EEPROM, MASTER_ACK is called.  This routine generates a ninth clock pulse
; and acknowledges the received bytes.  NO_ACK, which is called after the
; last byte of data is read, generates a ninth clock pulse.  STOP generates a
```

```
; stop condition.


; Parameters passed to this routine:
; Num_Bytes <--- number of bytes to be read from the EEPROM


; Parameters modified by this routine:
; [Data_In]+ <--- bytes read from the EEPROM

;*****************************************************************************


SEQ_CUR_RD:
    scall START                      ;generate start condition
                                     ;  data low then clock low
    ldb Temp_1,#DevSelRd
    scall TRANSMIT                   ;transmit write-select byte (A0)
    scall SLAVE_ACK                  ;check to see if EEPROM acknowledged write
                                     ;  generate 9th clock pulse

    ld Addr_Pntr,#Data_In            ;set up Addr_Pntr as input data pointer

NEXT3:
    cmpb Num_Bytes,#1                ;check for last byte; do not want to send
                                     ;  an acknowledge after last byte
    je NEXT4
    scall RECEIVE                    ;receive data byte
    stb ssio_stb1_w1f,[Addr_Pntr]+   ;save the received data byte, inc pointer
    scall MASTER_ACK                 ;acknowledge receipt of data byte
                                     ;  generate 9th clock pulse
    decb Num_Bytes                   ;decrement number of bytes
    sjmp NEXT3                       ;read next byte

NEXT4:
    scall RECEIVE                    ;read last byte
    stb ssio_stb1_w1f,[Addr_Pntr]    ;save the received data byte
    scall NO_ACK                     ;generate 9th clock pulse
    scall STOP                       ;generate stop condition
                                     ;  clock high then data high
    ret

;*************************** SEQ_RAN_RD ***********************************


; The SEQ_RAN_RD operation routine reads a specified number of bytes of data
; starting at a specified EEPROM address.  This routine uses the START,
; TRANSMIT, RECEIVE, SLAVE_ACK, MASTER_ACK, NO_ACK and STOP subroutines.
; START generates a start condition.  TRANSMIT uses the SSIO to write the
; read-device-select byte and the specified address to the EEPROM.  Before
; calling TRANSMIT, SEQ_RAN_RD writes the transmit data to Temp_1.  SLAVE_ACK
; generates a ninth clock pulse and checks to see that the EEPROM received
; each write to it.  RECEIVE uses the SSIO to read the data bytes starting at
; the specified EEPROM address.  After each data byte is read from the EEPROM
```

```
; (except the last one), MASTER_ACK is called.  This subroutine generates a
; ninth clock pulse and acknowledges received data.  After the last data byte
; is read from the EEPROM, NO_ACK is called.  NO_ACK generates a ninth clock
; pulse.  STOP generates a stop condition.

; Parameters passed to this routine:
; Num_Bytes <--- number of bytes to be read from the EEPROM
; Byte_Addr <--- starting EEPROM address of the bytes to be read

; Parameters modified by this routine:
; [Data_In]+ <--- bytes read from the EEPROM

;****************************************************************************

SEQ_RAN_RD:
    scall START                     ;generate start condition
                                    ;  data low then clock low
    ldb Temp_1,#DevSelWr
    scall TRANSMIT                  ;transmit write-select byte (A0)
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ;  generate 9th clock pulse

    ldb Temp_1,Byte_Addr
    scall TRANSMIT                  ;transmit byte address
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ;  generate 9th clock pulse

    scall START                     ;generate new start
    ldb Temp_1,#DevSelRd            ;load read-select byte
    scall TRANSMIT                  ;transmit read-select byte (A1)
    scall SLAVE_ACK                 ;check to see if EEPROM acknowledged write
                                    ;  generate 9th clock pulse
    ld Addr_Pntr,#Data_In           ;set up Addr_Pntr as input data pointer

NEXT6:
    cmpb Num_Bytes,#1               ;check for last byte, do not want to send
                                    ;  an acknowledge after last byte
    je NEXT7
    scall RECEIVE                   ;receive data byte
    stb ssio_stb1_w1f,[Addr_Pntr]+  ;save the received data byte
    scall MASTER_ACK                ;acknowledge receipt of data byte
                                    ;  generate 9th clock pulse
    decb Num_Bytes                  ;decrement number of bytes
    sjmp NEXT6                      ;read next byte

NEXT7:
    scall RECEIVE                   ;read last byte
    stb ssio_stb1_w1f,[Addr_Pntr]   ;save the received data
    scall NO_ACK                    ;generate 9th clock pulse
    scall STOP                      ;generate stop condition
```

```
                                      ;  clock high then data high
    ret

;############################## START ##################################

; START generates a start condition.  The pins are switched to their LSIO
; function, then put into a high impedence state so that the external
; resistors can pull the clock and data signals high.  The data signal is then
; pulled low.

; The number of NOPs required to meet the timing requirements was determined
; using a 16MHz crystal.  Execution time for a NOP is 4 state times.

; Fosc/2 * timing requirement = # of state times required.

;#####################################################################

START:
    andb p6mode_w1f,#ToLSIO          ;switch P6.6 and P6.7 to LSIO
    orb p6reg_w1f,#DataClockH        ;float clock and data lines so that they
                                     ;  can be pulled high by external resistor
    nop                              ;wait 4.7 us; see EEPROM timing spec Tchdx
    nop
    nop
    nop
    nop
    nop
    nop
    andb p6reg_w1f,#DataL            ;pull data line low
    nop                              ;wait 4 us; see EEPROM timing spec Tdlcl
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    andb p6reg_w1f,#ClockL           ;pull clock line low
    ret

;######################## RECEIVE AND TRANSMIT ##########################

; RECEIVE switches the pins to their SSIO function, starts the baud-rate
; generator, starts the receiver, and then transfers processing to SSIO_DONE.

; TRANSMIT initializes the transmitter, switches the pins to their SSIO
; function, starts the baud-rate generator, writes the transmit data to the
; transmit buffer to start the transmitter, and then transfers processing to
; SSIO_DONE.
```

```
; Parameters passed to TRANSMIT:
; Temp_1 <--- transmit data

; The SSIO baud rate was determined for 16MHz operation. For other frequencies,
; use the following formula to determine the value to write to the SSIO
; baud-rate register.  The most-significant bit must be set; it enables the
; baud-rate generator.

; baudrate = (Fosc/(100kHz * 8))-1.

; SSIO_DONE polls the done flag in the SSIO status register.  When the flag is
; set, processing returns to the calling routine.

;###########################################################################

RECEIVE:
    orb p6mode_w1f,#ToSSIO          ;switch lines to SSIO
    ldb ssio_baud_w1f,#093h         ;start baud-rate generator
    ldb ssio_stcr1_w1f,#088h        ;configure SSIO for receive mode
                                    ;  starts receiver
    sjmp SSIO_DONE

TRANSMIT:
    ldb ssio_stcr1_w1f,#0C8h        ;configure SSIO for transmit mode
    orb p6mode_w1f,#ToSSIO          ;switch lines to SSIO
    ldb ssio_baud_w1f,#093h         ;start baud counter
    stb Temp_1,ssio_stb1_w1f        ;write data to transmit buffer; this
                                    ;  starts transmitter

SSIO_DONE:
    andb Temp_1,ssio_stcr1_w1f,#01h
    cmpb Temp_1,#01h                ;wait for transmission/reception to finish
    jne SSIO_DONE

    orb p6reg_w1f,#DataH            ;float data line so that master or slave
                                    ;  can generate an acknowledge signal by
                                    ;  pulling it low (clk line already low)
    andb p6mode_w1f,#ToLSIO         ;switch lines to LSIO
    ret

;############################ SLAVE_ACK ####################################

; SLAVE_ACK floats the clock signal, generating the rising edge of the ninth
; clock pulse.  SLAVE_ACK then reads the data signal.  If the slave
; acknowledges the last write to it, the data signal will be low.  If the
; signal is low, SLAVE_ACK pulls the clock signal low, generating the ninth
; clock pulse falling edge.  If an acknowledge condition is not detected (data
; line is high) then processing is transfered to ACK_ERROR.

; The number of NOPs required to meet the timing requirements was determined
```

```
; using a 16MHz crystal.  Execution time for a NOP is 4 state times.

; Fosc/2 * timing requirement = # of state times required.

;############################################################################

SLAVE_ACK:
    orb p6reg_w1f,#ClockH            ;generate rising edge of 9th clock pulse
    nop                              ;need 4 us or 32 state times at 16 MHz
    nop                              ;  between clock high and clock low; see
    nop                              ;  EEPROM timing spec Tchcl
    andb Temp_1,p6pin_w1f,#80h
    cmpb Temp_1,#80h                 ;check to see if EEPROM acknowledged the
    je ACK_ERROR                     ;transmission; if not go to error routine
    andb p6reg_w1f,#DataClockL       ;generate falling edge of 9th clock pulse
    ret

ACK_ERROR:
    sjmp ACK_ERROR                   ;add error routine, EEPROM did not
                                     ;  acknowledge transmission

;########################### MASTER_ACK ####################################

; MASTER_ACK pulls the data line low, generating an acknowledge condition.
; MASTER_ACK the generates the rising edge of the ninth clock pulse, followed
; by the falling edge.

; NO_ACK generates the ninth clock pulse; the data signal is left unchanged.

; The number of NOPs required to meet the timing requirements was determined
; using a 16MHz crystal.  Execution time for a NOP is 4 state times.

; Fosc/2 * timing requirement = # of state times required.

;############################################################################

MASTER_ACK:
    andb p6reg_w1f,#DataL            ;acknowledge reception by pulling data
                                     ;  low
NO_ACK:
    orb p6reg_w1f,#ClockH            ;generate rising edge of 9th clock pulse
    nop                              ;wait 4 us; see EEPROM timing spec Tchcl
    nop
    nop
    nop
    nop
    nop
    nop
    andb p6reg_w1f,#ClockL           ;generate falling edge of 9th clock pulse
    ret
```

```
;############################## STOP ####################################

; STOP generates a stop condition.  The clock signal is put into a high
; impedance state so that the external resistor can pull it high.  The pins are
; then switched to their LSIO function.  The data signal is then put into a
; high impedance state so that the external resistor can pull it high.

; The number of NOPs required to meet the timing requirements was determined
; using a 16MHz crystal.  Execution time for a NOP is 4 state times.

; Fosc/2 * timing requirement = # of state times required.

;######################################################################

STOP:
    orb p6reg_w1f,#ClockH            ;float clock line so that ext resistor
                                     ;  can pull it high
    andb p6mode_w1f,#ToLSIO          ;switch lines to LSIO
    nop                              ;wait 4.7 us; see EEPROM timing spec Tchdh
    nop
    nop
    nop
    nop
    nop
    nop
    orb  p6reg_w1f,#DataH            ;float data line so that external resistor
                                     ;  can pull it high
    ret

;*************************** End of Code ********************************
END
```

3.2   Performance Information

Performance information is shown in Table 3. The execution times were measured using an 8XC196 emulator.

Table 3. Operation Execution Times

Subroutine	Execution Time	
	State Times	**Time at 16 MHz**
BYTE_WR (writes one byte to EEPROM)	2576	322 µs
MULTI_WR (writes four bytes to EEPROM)	4896	621 µs
PAGE_WR (writes eight bytes to EEPROM)	8240	1.03 ms
CURRENT_RD (reads one byte from EEPROM)	1680	210 µs
RANDOM_RD (reads one byte from EEPROM)	3456	432 µs
SEQ_CUR_RD† (reads eight bytes from EEPROM)	9520	1.19 ms
SEQ_RAN_RD† (reads eight bytes from EEPROM)	7360	920 µs

† This table indicates the execution time for reading eight bytes of data. However, the number of consecutive bytes that these operations can read is unlimited.