Interfacing the H8/3644 to a Serial E^2PROM

How to use the SCI Interface to emulate an SPI interface

HITACHI

Revision 1.1
8/4/98
Hitachi Semiconductor (America) Inc.
Leonard Haile

Contents

Summary

This application note provides assistance and source code to ease the design process of interfacing a Hitachi H8/3644 microcontroller with a Serial Peripheral Interface (SPITM) compatible serial E^{2}PROM. The Serial communication interface 1 (SCI1) hardware on the H8/3644 provides a simple three-wire connection to an SPI serial E^{2}PROM. No external “glue” hardware logic is required, but there are two design issues that the system designer must address.

1. SCI1 latches receive data at the rising edge of the serial clock. SCI1 outputs transmit data from one falling edge of the serial clock until the next rising edge. To meet these timing requirements, a designer must specify a SPI serial E^{2}PROM that uses the clock polarity/clock phase settings for Mode 1,1.
2. SCI1 transmits and receives data starting from the least significant bit. SPI serial E^{2}PROMs transmit and receive data starting from the most significant bit. For proper operation the H8/3644 application firmware must provide a function to reverse the data bits.

This application note addresses these two issues. First, background information on the SPI interface is provided to clarify the hardware specifications and make choosing an SPI serial E^{2}PROM straightforward . Secondly, an example hardware design with software is provided to reduce the system designer’s learning curve.

Section 1 Introduction

1.1 Serial E^{2}PROM Overview

Many microcontroller applications today need a small amount of non-volatile memory to store some data when the power is off. Serial E^{2}PROMs are a popular choice for this non-volatile storage because they have the following characteristics:

- low cost
- small size
- low I/O pin count
- High write/erase endurance
- byte level addressing
- low power consumption

Serial E^{2}PROM devices are available in a variety of densities, operating voltages, and packaging options. E^{2}PROM vendors offer a full line of serial E^{2}PROMs covering several industry standard serial communication protocols. When you look through the data sheets, the various serial E^{2}PROMs are listed as two-wire, three-wire, and SPI interface devices. A 2-wire interface product conforms to the I^{2}CTM bus hardware specification. A 3-wire product conforms to the MICROWIRETM specification. An SPI interface product conforms to one the several modes possible under the SPI hardware specification.

I^{2}CTM is a trademark of Phillips Corporation

MICROWIRETM is a trademark of National Semiconductor Corporation

1.2 SPI Overview

SPI is a general purpose synchronous serial interface. During an SPI transfer, transmit and receive data is simultaneously shifted out serially and shifted in serially. A serial clock line synchronizes shifting and sampling of the information on two serial data lines. Motorola created the SPI port in the mid 1980's to use in their microcontroller product families. The SPI is mainly used to allow a microcontrollers to communicate with peripheral devices such as E^{2}PROMs, A/D converters, and displays.

The SPI port is similar to the MICROWIRE interface created by National Semiconductor for their microcontrollers. Both interfaces use similar command protocols, however, SPI devices clock data in and out differently from the MICROWIRE devices. MICROWIRE devices clock data out and in on the same clock edge. SPI bus devices clock data in and out on opposite edges of the clock.

Section 2 SPI Details

2.1 SPI Pin Descriptions

The SPI interface between a microcontroller and serial E^2PROM has three lines that control data transfer and one general purpose I/O pin to control the E^2PROM's chip select. The names used for the SPI hardware lines can vary from one manufacturer's data sheet t to the next. Motorola calls their data out line MOSI (Master Out Slave In) and calls their data in line MISO (Master In Slave Out). Most of the data sheets from other manufacturers call these lines SO and SI. Table 2.1 is a summary of the SPI pin names from several manufacturers.

Table 2.1 SPI Pin Names

Manufacturer	Clock	Data In	Data Out
Motorola	SCK	MOSI	MISO
Hitachi (H8/3644)	SCK_1	SI_1	SO_1
Microchip	SCK	SI	SO
Xicor	SCK	SI	SO
SGS-Thomson	C	D	Q

Serial E2PROMs that have data lines labeled DI and D0 are MICROWIRE parts and are not compatible with the synchronous serial port on the H8/3644.

2.2 SPI Transfer Formats

For general purpose SPI microcontroller hardware, software can select one of four combinations of serial clock phase and polarity using two bits in a SPI control register. The clock polarity (CPOL) control bit selects an active high or active low clock. The clock phase (CPHA) control bit selects when the data in line is sampled and when the data out line is updated. The clock phase and the clock polarity must be the same for both the microcontroller and the specified SPI serial E^2PROM.

Table 2.2 CPOL Options

CPOL	Description
0	Clock line idles low
1	Clock line idles high

The synchronous serial port on the H8/3644 conforms to the CPOL = 1 setting because the SCK_1 line idles high.

Table 2.3 CPHA Options

CPHA	Description
1	SI data latched on rising edge of SCK SO data updated on falling edge of SCK
0	SI data latched on falling edge of SCK SO data updated on rising edge of SCK

The synchronous serial port on the H8/3644 conforms to the CPHA = 1 setting. The SI_1 line is sampled on the rising edge of SCK_1, and the SO_1 line is updated on the falling edge of SCK_1.

The selection of register bits CPOL =1 and CPHA = 1 is called SPI mode 1,1. This is the SPI mode that the H8/3644 can emulate. Mode 1,1 SPI serial E^2PROMs can be read and written using the synchronous serial port SCI_1 on the H8/3644

Section 3 SPI E^{2}PROM Details

3.1 The Commands

A microcontroller/SPI serial E^{2}PROM interface is command driven. For example, to read data from the memory:

1. the microcontroller asserts the chip select line low
2. sends a READ command
3. sends the address of the desired data
4. Uses the clock line to shift the data in
5. At the end of the transaction, the microcontroller raises the chip select line

For an E^{2}PROM there are also commands to WRITE the memory array, read and write to a status register, and set and clear a write enable latch. See Table 3.1 for a list of the commands for the 25xx040 E^{2}PROM that is used as the example in this application note.

Table 3.1 Instruction Set

Instruction Name	Instruction Format	Description
READ	0000 A_8 011	Read data from memory array beginning at selected address
WRITE	0000 A_8 010	Write data to memory array beginning at selected address
WRDI	0000 0100	Reset the write enable latch (disable write operations)
WREN	0000 0110	Set the write enable latch (enable write operations)
RDSR	0000 0101	Read status register
WRSR	0000 0001	Write status register

Note: A_8 is the 9th address bit necessary to fully address 512 bytes.

3.2 The 25xx040 E^{2}PROM

The Microchip 25xx040 is an example of a 4K bit (512 byte) Serial E^{2}PROM that can interface directly with the SCI1 port on the H8/3644 microcontroller. As a review, the bus signals required are a clock input (SCK), data in (SI) and data out (SO) lines. Access to the device is controlled by a chip select (CS) input. The SCK is used to synchronize the communication between the H8/3644 and the 25xx040. Instructions, addresses, or data present on the SI pin are latched on the rising edge of the SCK input. Data is shifted out through the SO pin on the falling edge of the SCK.

The CS pin must be low and the HOLD pin must be high for the entire operation. The WP pin must be held high to allow writing to the memory array.

Communication to the device can be paused via the hold pin (HOLD). While the device is paused, transitions on its inputs will be ignored, with the exception of chip select, allowing the host to service higher priority interrupts. Also, write operations to the device can be disabled via the write protect pin (WP).

3.3 Protocol

For the 25xx040 device because you need 9 bits to address the 512 bytes of memory, the most significant address bit (A8) is located in the instruction byte.

3.3.1 Read Command Bus Timing

See Figure 3.1 for an example of what the READ command looks like on a logic analyzer. The first byte transferred on the MOSI (SO_1) pin is the READ command. This is a command to read address 0x51 so A_8 , the most significant bit in the Address, is 0 in this case. So the first byte is 0x03. The second byte transferred on the MOSI pin contains the 8 low order address bits which are 0x51. The E^2PROM puts the last byte, 0x33, on the MISO (SI_1) pin.

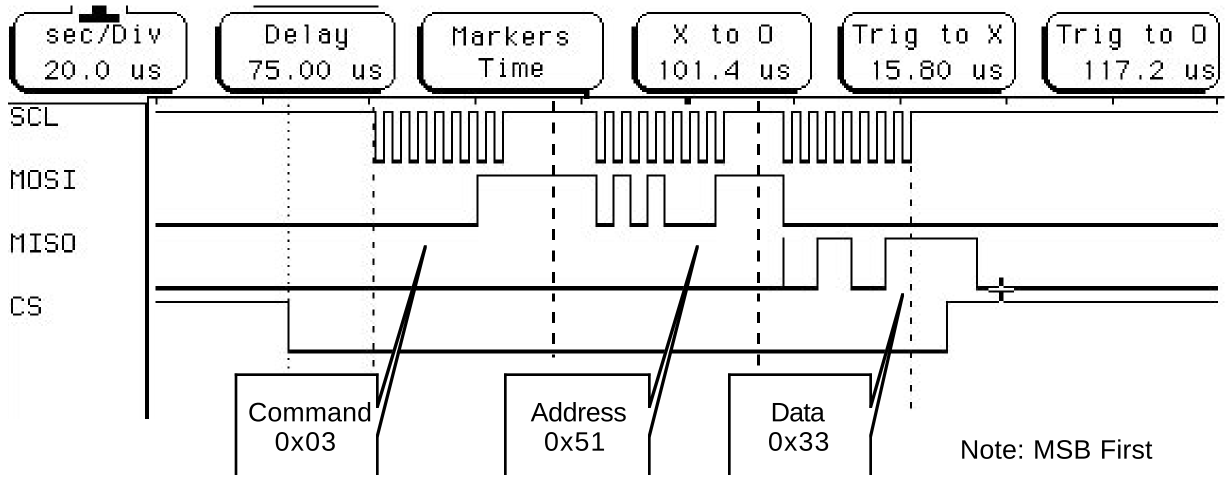


Figure 3.1 Read Command Timing

3.3.2 Write Command Bus Timing

See Figure 3.2 for an example of what the timing looks like on a logic analyzer to write data to an SPI E^2PROM. The first byte transferred on the MOSI (SO_1) pin is the write latch enable (WREN) command , 0x06. The next byte transferred on the MOSI (SO_1) pin is the WRITE command. This is a command to write to address 0x51 so $A_8 = 0$. The second byte is 0x02. The third byte transferred contains the 8 low order address bits which are 0x51. The last byte, 0xa3, is the data to be written into address 0x51.

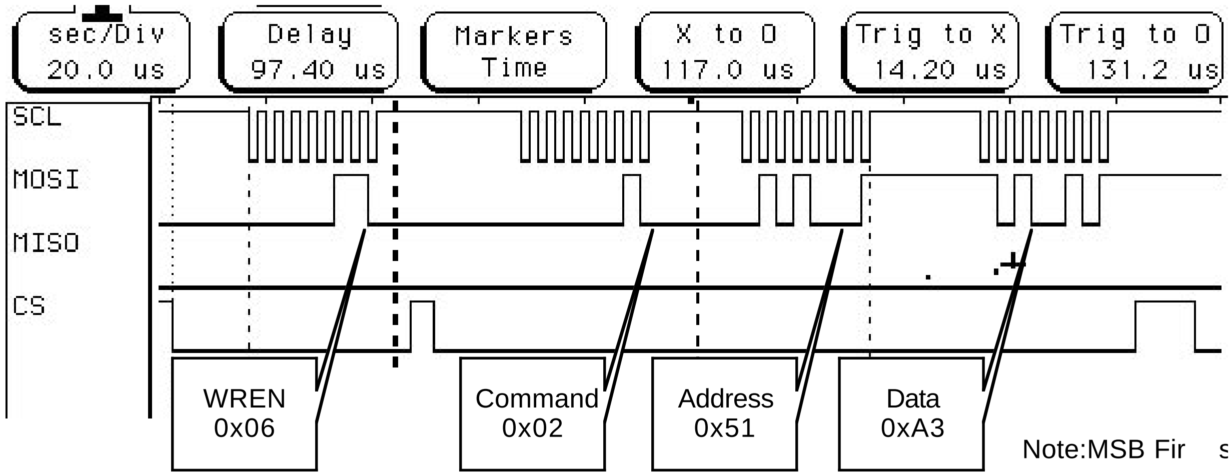


Figure 3.2 Write Command Timing

3.3.3 Write Command Polling

See Figure 3.3 to get an idea of the relative time it takes to write one memory location, This logic analyzer trace shows that total time from the WREN command until the read status register command (RDSR) says that the write is complete is 1.96 ms.

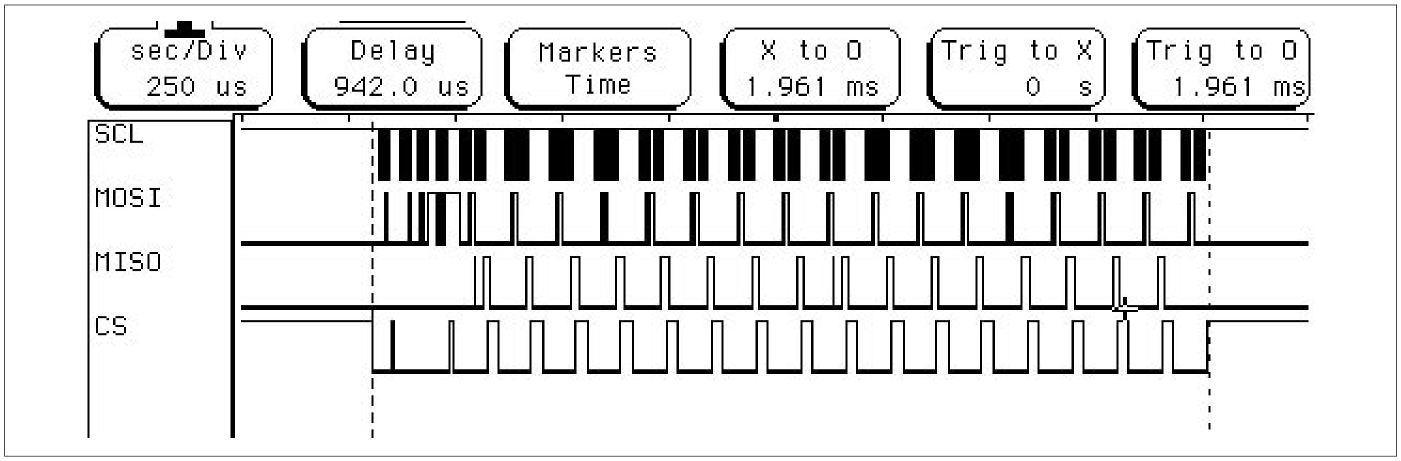


Figure 3.3 Write Command Timing (To Completion)

Section 4 SCI1 Operation

4.1 Hardware Registers

SCI1 on the H8/3644 operates only in the synchronous mode. Table 4.1 lists the hardware registers for SCI1.

Table 4.1 SCI1 Registers

Name	Abbrev.	R/W	Initial Value	Address
Serial control register 1	SCR1	R/W	H'00	H'FFA0
Serial control status register 1	SCSR1	R/W	H'9C	H'FFA1
Serial data register U	SDRU	R/W	Not fixed	H'FFA2
Serial data register L	SDRL	R/W	Not fixed	H'FFA3

4.1.1 Serial Control Register Bits

Serial Control Register 1 (SCR1) is the only SCI1 register that needs to be setup to initialize the H8/3644 to interface to a SPI E^2PROM. Below are the settings for the bits in the SCR1register that are used in this application note.

SCR1: 0xFFA0

7	6	5	4	3	2	1	0
SNC1	SNC0	MRKON	LTCH	CKS3	CKS2	CKS1	CKS0
0	0	0	0	0	1	0	0

Note: CKS2 = 1,CKS1 = 0,CKS0 = 0 $\rightarrow$ 312.5 kHz SCK_1 (F = 5MHz)

The SCI1 port can transmit 16-bit data or 8-bit data. SCI1 should be setup to send 8-bit data to read and write to an SPI E^2PROM. Set SNC1 = 0 and SNC2 = 0 for 8-bit data transfers.

SCR1 can control multiple ICs using the Synchronized Serial Bus (SSB) communication’s protocol. Make sure that MRKON = 0 to disable the SSB communication’s protocol.

SCI1 can operate with an internal or external clock selected as the clock source. The H8/3644 should generate the clock from the internal prescaler and output the clock signal on the SCK_1 pin. Set the CKS3 = 0 to select the internal clock. Set CKS2-CKS0 to select a prescaler division value that gives the correct bit rate.

Writing data to SCR1 when bit MRKON in SCR1 is cleared to 0 initializes the internal state of SCI1.

4.1.2 Port Mode Register 3

In addition, the I/O pins for SO_1, SI_1 and SCK_1 have to be enable in Port Mode Register 3 (PMR3) for use as serial port pins. See below.

PMR3: 0xFFFD

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
					SO1	SI1	SCK1
0	0	0	0	0	1	1	1

4.2 Data Transfer Operations

The protocol for an H8/3644 to communicate with an SPI E^2PROM requires the simultaneous transmission and reception of data using SCI1.

A simultaneous transmit/receive operation is carried out as follows:

1. Write transmit data in SDRL.
2. Set the SCSR1 start flag (STF) bit to 1.
   SCI1 starts operating.
   Transmit data is output at pin SO_1.
   Receive data is input at pin SI_1.
3. After data transmission and reception are complete, bit IRRS1 in IRR2 is set to 1.
4. Read the received data from SDRL

4.2.1 Serial Control/Status Register Bits

SCSR1: 0xFFA1

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	SOL	ORER				MTRF	STF
1	0	0	1	1	1	0	0

Note: Initial values

4.2.2 Interrupt Request Register 2

IRR2: 0xFFF8

7	6	5	4	3	2	1	0
IRRDT	IRRAD		IRRS1			MTRF	STF
0	0	0	0	0	0	0	0

Note: Initial values

IRRS1 is set to 1 when an SCI1 interrupt is requested. The flag is not cleared automatically when an interrupt is accepted. It is necessary to write 0 to clear the flag.

After data transmission is complete, the serial clock is not output until the next time the start flag is set to 1. During this time, pin SO_1 continues to output the value of the last bit transmitted.

Section 5 Test Hardware

Figure 5.1 is the schematic showing the basic components for a representative circuit that can connect a Hitachi H8/3644 microcontroller to a 25xx040 E^{2}PROM. A DS1233 low cost power supply monitor/reset IC improves the reliability of the E^{2}PROM data. If the power supply voltage drops below the minimum value for Vcc, then the H8/3644 will be reset. before it can execute random code that might write erroneous data to the E^{2}PROM.

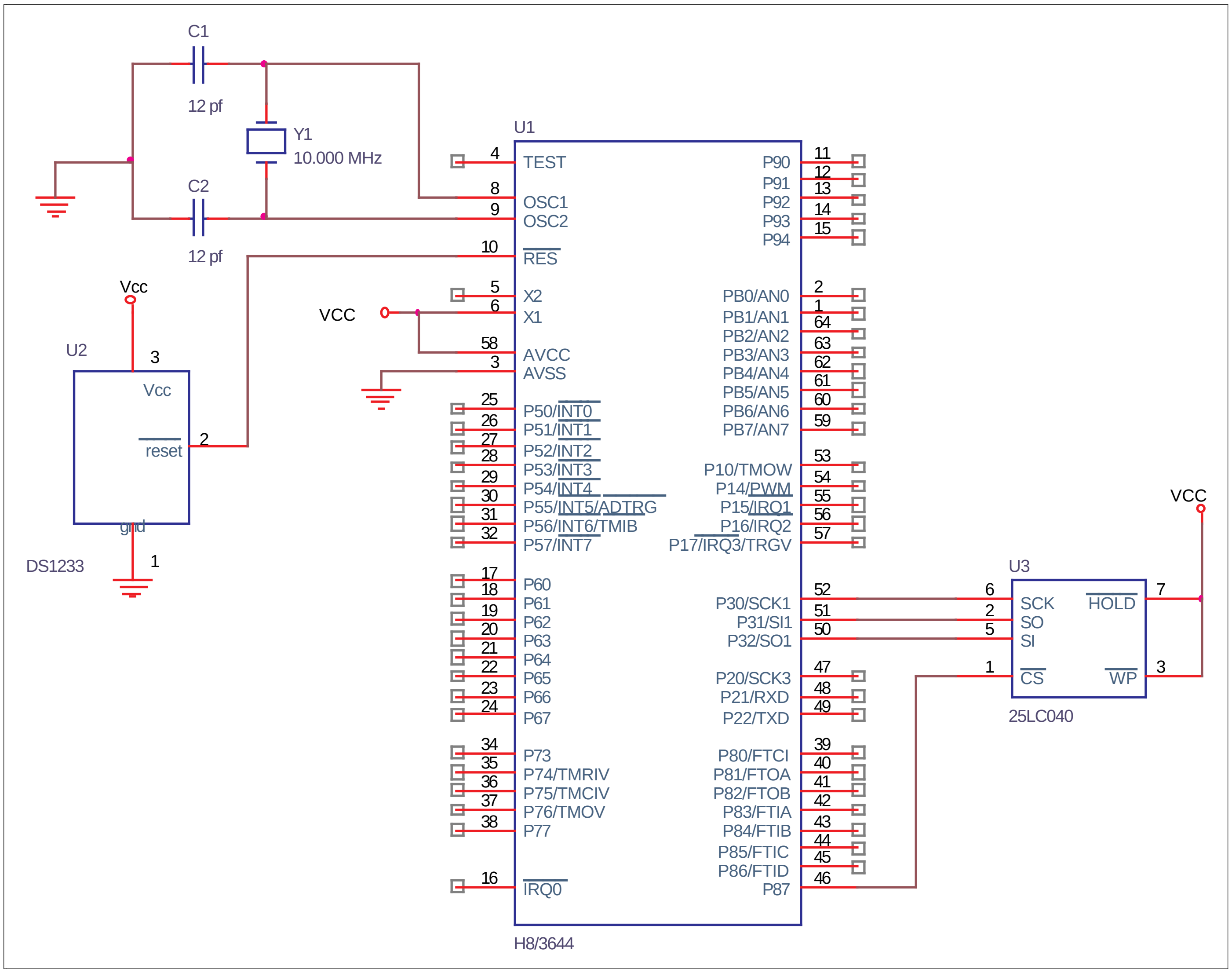


Figure 5.1 Hardware Block Diagram

In this design, I/O port 8 pin 7 is configured as an output and is used as the chip select for the 25LC040 E^{2}PROM. The HOLD and WP functions of the E^{2}PROM are not used in this example, and the control pins for those functions are tied to their inactive voltage state, Vcc.

Section 6 Software Details

6.1 Bit Reverse Function

The SPI serial E^{2}PROM communications protocol specifies that data will be transmitted starting with the most significant bit first. In synchronous mode the SCI interfaces on H8 microcontrollers shift data in and out starting with the least significant bit first. The application software can rotate the data bits to adjust for the difference in the two specifications. The example code in 6.1.1 below shows two ways to implement a bit-reversal function in assembly language that you can call from your C code. 6.1.4 shows a bit-reverse function in C code.

6.1.1 Bit-Reversal Source Code in Assembler

```
        .section P,CODE,ALIGN=2

        .export _mirror

        .export _reverse

_mirror:

        rotxr.b r0l

        rotxl.b r0h

        rotxr.b r0l

        rotxl.b r0h

        rotxr.b r0l

        rotxl.b r0h

        rotxr.b r0l

        rotxl.b r0h

        rotxr.b r0l

        rotxl.b r0h

        rotxr.b r0l

        rotxl.b r0h

        rotxr.b r0l

        rotxl.b r0h

        rotxr.b r0l

        rotxl.b r0h

        mov r0h,r0l
```

```
            rts
_reverse:
            mov.b   #8,r1l
loop:
            rotxr.b r0l
            rotxl.b r0h
            dec     r1l
            bne             loop
            mov             r0h,r0l
            rts
        .END
```

6.1.2 Example Declaration

Using the function mirror() as an example, the bit-reversal function is declared in you C code as:

```
extern unsigned char mirror(unsigned char);
```

6.1.3 Design Tradeoffs

The function, mirror(), is optimized for speed of execution. The function, reverse(), is optimized for code size. The C function mirror2() was added to make the bit-reversal function more portable. Table 6.1 highlights the differences between the three example functions.

Table 6.1 Bit-Reversal Function Tradeoffs

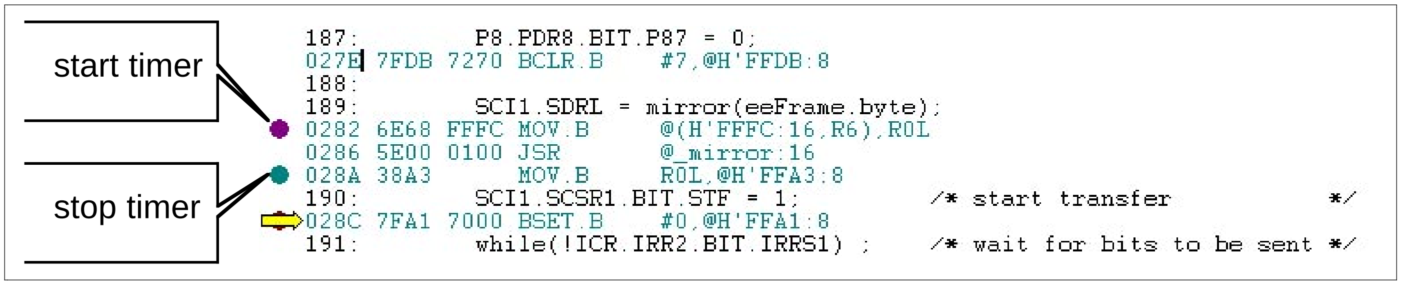
Function	Size (bytes)[1]	Execution Speed (µsec)[2] (F = 5 MHz)
mirror	44	11.2
reverse	22	21.2
mirror2[3]	42	38.8

Notes 1,2,3: Includes the overhead to call the function
Note 3: C code optimized for speed with register optimization on

Note: See Figure 6.1 for an example of how to fill-in the HiVIEW dialog box to generate C code that is optimized for speed with register optimization turned on.

This application note uses the function mirror() to reverse the bits. The bus timing examples show in Section 3 reflect the 11.2 µseconds of overhead required to reverse the bits before they are written to the SCI1 data register in the H8/3466.

Figure 6.1 shows that the 11.2 µseconds includes the overhead to call the function in C as well as the time it takes to execute the assembly language code in the functions themselves.



```
187:          P8.PDR8.BIT.P87 = 0;
027E 7FDB 7270 BCLR.B    #7,@H'FFDB:8
188:
189:          SCI1.SDRL = mirror(eeFrame.byte);
0282 6E68 FFFC MOV.B     @(H'FFFC:16,R6),R0L
0286 5E00 0100 JSR       @_mirror:16
028A 38A3      MOV.B     R0L,@H'FFA3:8
190:          SCI1.SCSR1.BIT.STF = 1;        /* start transfer          */
028C 7FA1 7000 BSET.B    #0,@H'FFA1:8
191:          while(!ICR.IRR2.BIT.IRRS1) ;   /* wait for bits to be sent */
```

Figure 6.1 Bit Reverse Function Timing Details

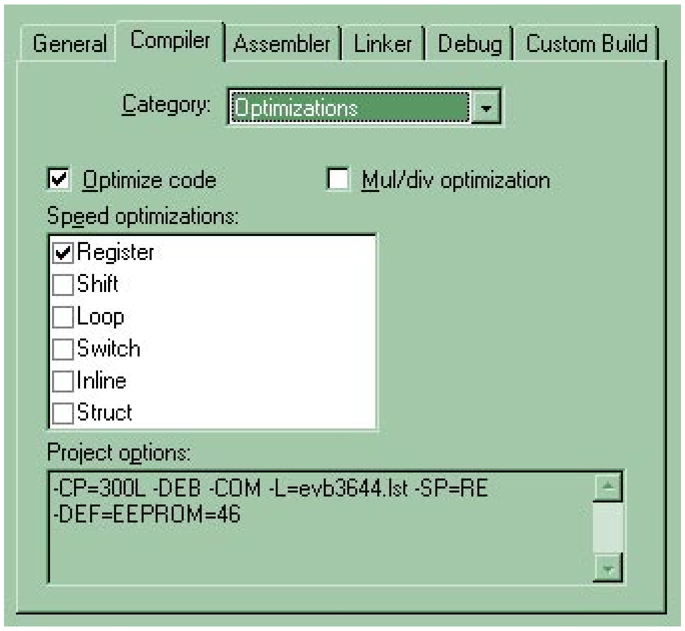
6.1.4 Bit-Reversal Function in C

```
unsigned char mirror2(unsigned char value )
{
        unsigned char inMask,outMask,i,reversed;

        inMask =        0x01;
        outMask =       0x80;
        reversed = 0;
        for(i=8;i>0 ;i--)
        {
                if(inMask & value)
                        reversed |= outMask;
                inMask <<= 1;

                outMask >>= 1;
        }
        return(reversed);

  }
```



Category: Optimizations

[✓] Optimize code
[ ] Mul/div optimization

Speed optimizations:
[✓] Register
[ ] Shift
[ ] Loop
[ ] Switch
[ ] Inline
[ ] Struct

Project options:

```
-CP=300L -DEB -COM -L=evb3644.lst -SP=RE
-DEF=EEPROM=46
```

Figure 6.1 C Code Optimized for Speed

6.2 E^{2}PROM Interface C Functions

After eepromInit() is called, the rest of the software for this application note is a just a collection of C functions that simultaneously transmit and receive 8 bit data using SCI1 on the H8/3644. The software functions are mostly just wrappers around the 25xx040 command set that is shown in Table 3.1. These basic example functions do not do error checking and do not return status codes.

Table 6.2 provides a summary for the C functions that interface the H8/3644 to the Microchip 25xx040 E^{2}PROM in this application note.

Table 6.2 E^{2}PROM Interface Functions Summary

Function	Description
void eepromInit(void);	Must be called first Initialize H8/3644 SCI1 for 8 bit synchronous operation Initialize I/O pin used for E^{2}PROM chip select Initialize E^{2}PROM by toggling the chip select to enter active state
BYTE Read(WORD address);	Uses the READ command to return the byte at an E^{2}PROM address
void Write(WORD address, BYTE value);	Uses the WRITE command to write a byte to an E^{2}PROM address The function does not return until the write operation has completed.
Void WriteEnable(void);	Enables WRITE commands Normally only called by the function Write() The WREN command must be sent to the E^{2}PROM before the E^{2}PROM will execute a WRITE command
void WriteDisable(void);	Uses the WRDI command to disable WRITE commands Normally does not need to be called by the user After executing a WRITE command, the E^{2}PROM automatically disables further WRITE commands.
BYTE ReadStatus();	Uses the RDSR command to read the status register The function Write() calls ReadStatus() repeatedly to poll the write in progress bit until the write operation completes
void WriteProtect(PROTECTION value);	Uses the WRSR command to write to the bits in the status register that write protect sections of the E^{2}PROM's memory array.

The example hardware/software design in this application note includes just the basics on how to use the H8/3644 SCI1 interface to read and write data using a SPI E^{2}PROM. If you feel you need additional write protection security, you can use software to control an I/O pin that is connected the WP pin on the 25xx040.

The source code for the E^{2}PROM interface functions can be found in Appendix A of this application note. The header file can be found in Appendix B.

Appendix A Code

```
#include "ee3644.h"

extern unsigned char mirror(unsigned char);

typedef unsigned char BYTE;
typedef unsigned int WORD;

typedef enum protectionn_tag
{
        NONE    = 0,            /* none            */
        QUARTER = 1,            /* 0x180 - 0x1ff  */
        HALF    = 2,            /* 0x100 - 0x1ff  */
        ALL             = 3                     /* 0x000 - 0x1ff  */

}PROTECTION;


typedef enum Instruction_tag
{
        READ  = 3,
        WRITE = 2,
        WRDI = 4,
        WREN = 6,
        RDSR = 5,
        WRSR = 1

}INSTRUCTION;


typedef union header_tag
{
        unsigned char byte;
        struct {
                unsigned int Space              : 4;
                unsigned int AB8                : 1;
                unsigned int Opcode     : 3;
        }fields;

}HEADER;

typedef union status_tag
{
        unsigned char byte;
        struct {
                unsigned char wk        :4;
                unsigned char BP        :2;
                unsigned char WEL       :1;
                unsigned char WIP       :1;
        }bits;
}STATUS;
```

```
/*******************************\
*  Function Prototypes                  *
\*******************************/
void eepromInit(void);
void WriteEnable(void);
void WriteDisable(void);
BYTE Read(WORD address);
void Write(WORD address, BYTE value);
BYTE ReadStatus();
void WriteProtect( PROTECTION value);
unsigned char mirror2(unsigned char value );



/***********************\
*  Global Variables              *
\***********************/

BYTE value;
INSTRUCTION instruction;
WORD address;
BYTE data;
PROTECTION segment;

int main (void)
{

        eepromInit();


        while (1)
        {
                address = 0x51;
                data = 0x96;
                instruction = READ;
                value = 0xa5;
                segment = NONE;

                switch(instruction)
                {
                        case READ:
                                data = Read(address);
                                break;
                        case WRITE:
                                Write(address,value);
                                break;
                        case WRDI:
                                WriteDisable();
                                break;
                        case WREN:
                                WriteEnable();
                                break;
                        case RDSR:
                                data = ReadStatus();
                                break;
                        case WRSR:
                                WriteProtect(segment);
                                break;
                };

        }
        return (0);
}

void eepromInit(void)
{

        P8.PCR8.BIT.PCR87 = 1;                  /* P8.7 is CS for eeprom                */

        PMR3.PMR.BIT.SO1 = 1;                           /* turn on S0                           */
        PMR3.PMR.BIT.SCK1 = 1;                  /* turn on SCK                          */
        PMR3.PMR.BIT.SI1 = 1;                   /* turn on SI                                   */

        SCI1.SCR1.BIT.SNC = 0;                  /* 8-bit synchronous transfer mode  */
        SCI1.SCR1.BIT.CKS3 = 0;                 /* clock source is prescaler        */
        SCI1.SCR1.BIT.CKS = 4;                  /* phi/16 clock                         */

        P8.PDR8.BIT.P87 = 0;                    /* toggle CS after power-up             */
        P8.PDR8.BIT.P87 = 1;

}


void WriteEnable(void)
{
        P8.PDR8.BIT.P87 = 0;                    /* chip select                          */

        SCI1.SDRL = mirror(WREN);
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer                       */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent         */
        ICR.IRR2.BIT.IRRS1 = 0;

        P8.PDR8.BIT.P87 = 1;

}

void WriteDisable(void)
{
        P8.PDR8.BIT.P87 = 0;

        SCI1.SDRL = mirror(WRDI);
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer                       */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent         */
        ICR.IRR2.BIT.IRRS1 = 0;

        P8.PDR8.BIT.P87 = 1;
```

```
}

BYTE Read(WORD address)
{
        HEADER eeFrame;

        eeFrame.byte = 0;
        eeFrame.fields.AB8 = (address > 0xff);
        eeFrame.fields.Opcode = READ;

        P8.PDR8.BIT.P87 = 0;

        SCI1.SDRL = mirror(eeFrame.byte);
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;
        SCI1.SDRL = mirror((address & 0xff));   /* low order address bits         */
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;
        SCI1.SDRL = 0;                                  /* clock in data              */
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;

        P8.PDR8.BIT.P87 = 1;

        return(mirror(SCI1.SDRL));

}

BYTE ReadStatus()
{
        HEADER eeFrame;

        eeFrame.byte = 0;
        eeFrame.fields.Opcode = RDSR;

        P8.PDR8.BIT.P87 = 0;

        SCI1.SDRL = mirror(eeFrame.byte);
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;
        SCI1.SDRL = 0;                                  /* clock in data              */
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;

        P8.PDR8.BIT.P87 = 1;

        return(mirror(SCI1.SDRL));
```

```
}

void WriteProtect( PROTECTION value)
{
        HEADER eeFrame;
        STATUS eeStatus;

        eeFrame.byte = 0;
        eeFrame.fields.AB8 = (address > 0xff);
        eeFrame.fields.Opcode = WRSR;

        eeStatus.byte = 0;
        eeStatus.bits.BP = value;

        WriteEnable();

        P8.PDR8.BIT.P87 = 0;

        SCI1.SDRL = mirror(eeFrame.byte);
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;
        SCI1.SDRL = mirror(eeStatus.byte);      /* low order addrress bits  */
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;

        P8.PDR8.BIT.P87 = 1;

        while(ReadStatus() & 0x01) ;            /* wait for programming        */
}


void Write(WORD address, BYTE value)
{
        HEADER eeFrame;

        eeFrame.byte = 0;
        eeFrame.fields.AB8 = (address > 0xff);
        eeFrame.fields.Opcode = WRITE;

        WriteEnable();

        P8.PDR8.BIT.P87 = 0;

        SCI1.SDRL = mirror(eeFrame.byte);
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;
        SCI1.SDRL = mirror(address & 0xff);    /* low order addrress bits         */
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer              */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;
        SCI1.SDRL = mirror(value);              /* clock out data              */
        SCI1.SCSR1.BIT.STF = 1;                 /* start transfer               */
        while(!ICR.IRR2.BIT.IRRS1) ;            /* wait for bits to be sent  */
        ICR.IRR2.BIT.IRRS1 = 0;

        P8.PDR8.BIT.P87 = 1;

        while(ReadStatus() & 0x01) ;            /* wait for programming         */
}

unsigned char mirror2(unsigned char value )
{
        unsigned char inMask,outMask,i,reversed;

        inMask =        0x01;
        outMask =       0x80;
        reversed = 0;
        for(i=8;i>0 ;i--)
        {
                if(inMask & value)
                        reversed |= outMask;
                inMask <<= 1;

                outMask >>= 1;
        }
        return(reversed);

}
```

Appendix B Header Files

```
#ifndef __evb3644__
#define __evb3644__

struct st_icr {

                union {                                         /*  IEGR1                    */
                        unsigned char BYTE;             /*  Byte Access    */
                        struct {                                /*  Bit  Access     */
                                unsigned char wk :4;
                                unsigned char IEG3          :1;
                                unsigned char IEG2          :1;
                                unsigned char IEG1          :1;
                                unsigned char IEG0          :1;
                        }           BIT;
                }           IEGR1;

                union {                                         /*  IEGR2                    */
                        unsigned char BYTE;             /*  Byte Access    */
                        struct {                                /*  Bit  Access     */
                                unsigned char INTEG7      :1;
                                unsigned char INTEG6      :1;
                                unsigned char INTEG5      :1;
                                unsigned char INTEG4      :1;
                                unsigned char INTEG3      :1;
                                unsigned char INTEG2      :1;
                                unsigned char INTEG1      :1;
                                unsigned char INTEG0      :1;
                        }           BIT;
                }           IEGR2;

                union {                                         /*  IENR1                    */
                        unsigned char BYTE;             /*  Byte Access    */
                        struct {                                /*  Bit  Access     */
                                unsigned char IENTB1      :1;
                                unsigned char IENTA       :1;
                                unsigned char wk0          :2;
                                unsigned char IEN3          :1;
                                unsigned char IEN2          :1;
                                unsigned char IEN1          :1;
                                unsigned char IEN0          :1;
                        }           BIT;
                }           IENR1;

                union {                                         /*  IENR2                    */
                        unsigned char BYTE;             /*  Byte Access    */
                        struct {                                /*  Bit  Access     */
                                unsigned char IENDT        :1;
                                unsigned char IENAD        :1;
                                unsigned char wk1          :1;
                                unsigned char IENS1        :1;
                unsigned char wk2           :4;
        }       BIT;
}       IENR2;

union {                                             /*  IENR3                   */
        unsigned char BYTE;             /*  Byte Access    */
        struct {                                    /*  Bit  Access     */
                unsigned char INTEN7        :1;
                unsigned char INTEN6        :1;
                unsigned char INTEN5        :1;
                unsigned char INTEN4        :1;
                unsigned char INTEN3        :1;
                unsigned char INTEN2        :1;
                unsigned char INTEN1        :1;
                unsigned char INTEN0        :1;
        }       BIT;
}       IENR3;

union {                                             /*  IRR1                    */
        unsigned char BYTE;             /*  Byte Access    */
        struct {                                    /*  Bit  Access     */
                unsigned char IRRTB1        :1;
                unsigned char IRRTA         :1;
                unsigned char wk3           :2;
                unsigned char IRRI3         :1;
                unsigned char IRRI2         :1;
                unsigned char IRRI1         :1;
                unsigned char IRRI0         :1;
        }       BIT;
}       IRR1;

union {                                             /*  IRR2                    */
        unsigned char BYTE;             /*  Byte Access    */
        struct {                                    /*  Bit  Access     */
                unsigned char IRRDT         :1;
                unsigned char IRRAD         :1;
                unsigned char wk4           :1;
                unsigned char IRRS1         :1;
                unsigned char wk5           :4;
        }       BIT;
}       IRR2;

union {                                             /*  IRR3                    */
        unsigned char BYTE;             /*  Byte Access    */
        struct {                                    /*  Bit  Access     */
                unsigned char INTF7         :1;
                unsigned char INTF6         :1;
                unsigned char INTF5         :1;
                unsigned char INTF4         :1;
                unsigned char INTF3         :1;
                unsigned char INTF2         :1;
                unsigned char INTF1         :1;
                unsigned char INTF0         :1;
```

```
                        }         BIT;
                }         IRR3;
};


struct st_sci1 {

                union {                                        /*  SCR1                 */
                        unsigned char BYTE;            /*  Byte Access   */
                        struct {                               /*  Bit  Access    */
                                unsigned char SNC          :2;
                                unsigned char MRKON        :1;
                                unsigned char LTCH         :1;
                                unsigned char CKS3         :1;
                                unsigned char CKS          :3;
                        }         BIT;
                }         SCR1;

                union                                              /*  SCSR1                */
                        unsigned char BYTE;            /*  Byte Access   */
                        struct {                               /*  Bit Access     */
                                unsigned char wk :1;
                                unsigned char SOL          :1;
                                unsigned char ORER         :1;
                                unsigned char wk1          :3;
                                unsigned char MTRF         :1;
                                unsigned char STF          :1;
                        }         BIT
                }         SCSR1

                unsigned char SDRU;
                unsigned char SDRL
};

struct st_pm3 {
        union {                                               /*  PMR                  */
                unsigned char BYTE;                 /*  Byte Access   */
                struct {                                      /*  Bit Access     */
                        unsigned char wk  :5;
                        unsigned char SO1 :1;
                        unsigned char SI1 :1;
                        unsigned char SCK1:1;
                }BIT;
        }         PMR;
};

struct st_p8 {
        union {                                                       /*  PDR8                 */
                unsigned char BYTE;                         /*  Byte Access   */
                struct    {                                               /*  Bit Access     */
                        unsigned char P87:1;
                        unsigned char P86:1;
                        unsigned char P85:1;
                        unsigned char P84:1;
                        unsigned char P83:1;
                        unsigned char P82:1;
                        unsigned char P81:1;
                        unsigned char P80:1;
                }       BIT;
        }       PDR8    ;

        char    wk3[15];

        union {                                         /*   PCR8                   */
                unsigned char BYTE;             /*  Byte Access   */
                struct {                                /*  Bit  Access     */
                        unsigned char PCR87:1;
                        unsigned char PCR86:1;
                        unsigned char PCR85:1;
                        unsigned char PCR84:1;
                        unsigned char PCR83:1;
                        unsigned char PCR82:1;
                        unsigned char PCR81:1;
                        unsigned char PCR80:1;
                }       BIT;
        }       PCR8    ;
};



#define SCI1    (*(volatile struct st_sci1    *)0xFFA0)          /* SCI1 Addr A0  */
#define PMR3  (*(volatile struct st_pm3    *)0xFFFD)          /* PMR3 Address */
#define ICR     (*(volatile struct st_icr      *)0xFFF2)          /* ICR Address    */
#define P8      (*(volatile struct st_p8                *)0xFFDB)         /* P8 Addr DB                   */

#endif
```