Application Note

FUJITSU

Communication with SPI device via the Uart or the SIO of the MB90540 series.

This application note describes how a SPI device can be connected to the Fujitsu Microcontroller MB90540 series. Two different possibilities exist to communicate with the SPI device. One is the SIO, the other is the Uart in synchronous mode.

History

05th May 00	Men	V1.0	New Format, new updated version
20th May 00	Men		Updated version
13 th Sep 00	Men	V1.1	Some slide changes

Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH restricts its warranties and its liability for **all products delivered free of charge** (eg. software include or header files, application examples, application Notes, target boards, evaluation boards, engineering samples of IC’s etc.), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling. **Note, all these products are intended and must only be used in an evaluation laboratory environment**.

1. Fujitsu Microelectronics Europe GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials for a period of 90 days form the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Microelectronics Europe GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.

2. Should a Product turn out to be defect, Fujitsu Microelectronics Europe GmbH´s entire liability and the customer´s exclusive remedy shall be, at Fujitsu Microelectronics Europe GmbH´s sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Microelectronics Europe GmbH in original packing and without further defects resulting from the customer´s use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Microelectronics Europe GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Microelectronics Europe GmbH.

3. To the maximum extent permitted by applicable law Fujitsu Microelectronics Europe GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.

4. To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH´s and its suppliers´liability is restricted to intention and gross negligence.

   NO LIABILITY FOR CONSEQUENTIAL DAMAGES

   To the maximum extent permitted by applicable law, in no event shall Fujitsu Microelectronics Europe GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.

Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect.

Contents

1.0 Introduction:

The Serial Peripheral Interface (SPI) circuit is a synchronous serial data link that is standard across many Motorola microprocessors and other peripheral chips. It provides support for a high bandwidth (1 megabaud) network connection amongst CPUs and other devices supporting the SPI.

The SPI is essentially a shift register that serially transmits data bits to other SPI's. During a data transfer, one SPI system acts as the "master" which controls the data flow, while the other system acts as the "slave" which has data shifted into and out of it by the master. Different CPU's can take turn being masters, and one master may simultaneously shift data into multiple slaves. However, only one slave may drive its output to write data back to the master at any given time.

The SPI that is used in this application is the NM25C020. This device is a 2k serial CMOS EEPROM.

2.0 Background:

The serial peripheral interface (SPI), as the name implies, is primarily used to allow the microcontroller unit (MCU) to communicate with peripheral devices. Communication is performed synchronously with the most significant bit (MSB) first. Peripheral devices are simple as an ordinary transistor-transistor logic (TTL) shift register or as complex as a complete subsystem, such as a liquid crystal diode (LCD) display driver or an analog-to-digital (A/D) converter subsystem.

During an SPI transfer, data is simultaneously transmitted (shifted out serially) and received (shifted in serially). A serial clock line synchronizes shifting and sampling of the information on the two serial data lines. A slave select or chip select line allows individual selection of the SPI device. Unselected devices do not interfere with SPI bus activities.

3.0 SPI Clock Phase and Polarity Controls:

The clock phase can be any of two fundamentally different transfer formats. These two transfer modes are described below. The clock phase and polarity should be identical for the master SPI device and the communicating slave device.
The clock polarity can be selected active high or active low clock and has no significant effect on the transfer format.

3.1.0 Clock Phase Equals Zero Transfer Format:

When clock phase equals zero, the chip select (CS low active) line must be negated and reasserted between each successive serial byte. Also, if the slave writes data to the MCU shift register while CS is active low, a write-collision error results.

3.1.1 Clock Phase Equals One Transfer Format:

When clock phase equals one, the CS line may remain active low between successive transfers (can be tied low all the times). This format is sometimes preferred in systems having a single fixed master and a single slave driving the MISO data line.

The diagram 1 shows the timing of the SPI for the two different transfer's formats.

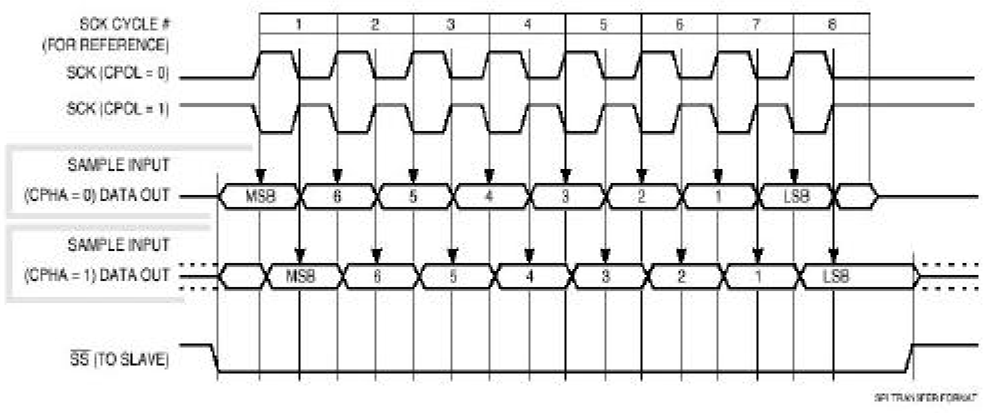


Diagram 1

4.0 SPI via Serial I/O (SIO):

The SIO is quite similar to the SPI, because communication is achieved only synchronously and the MSB is shifted out first. A bit in the serial mode control status register (SMCS) decides the shifting direction. Data is received and transmit simultaneously with the serial shift register (SDR). A serial clock line synchronizes shifting and sampling of the information on the two serial data lines. The bits in the shift register are serially output via the serial output pin (SOT2) at the falling edge of the serial shift clock (external clock or internal clock). The bits are serially input to the shift register via the serial input pin (SIN2) at the rising edge of the serial shift clock. At the end of the serial transfer, this block is stopped or stands by for a read or write of the data register according to the MODE bit of the serial mode status register.

The SIO offers two methods for the conditions to start the transfer operation from the stop state. A MODE bit selects the condition in which the SIO is started. Upon a reset the MODE bit is set low.

<u>MODE = 0:</u>

The first operation method is that to start with a transfer the STRT bit has to be set high and the STOP bit low. During the transfer of data the STRT and BUSY will remain high and the STOP will be low. After transmission is complied STRT and BUSY will change to low and STOP too high. If a next transfer should be started the described sequence has to be repeated.

<u>MODE = 1:</u>

The second operation method is performed by an interrupt. First the device has to be set in standby transition. This is achieved by setting STRT high and STOP low. When BUSY becomes low an interrupt request is issued to CPU and a transmission can be executed. After the transmission is finished the SIR bit has to be cleared. This has to be done to avoid that the program loops inside the interrupt service routine. If the program has finished the service routine and the SIR flag remains high, a now interrupt is requested to CPU and the program will repeat the service routine until a reset occurs.

The transition figure 1 shows the different states for the operation of the SIO.

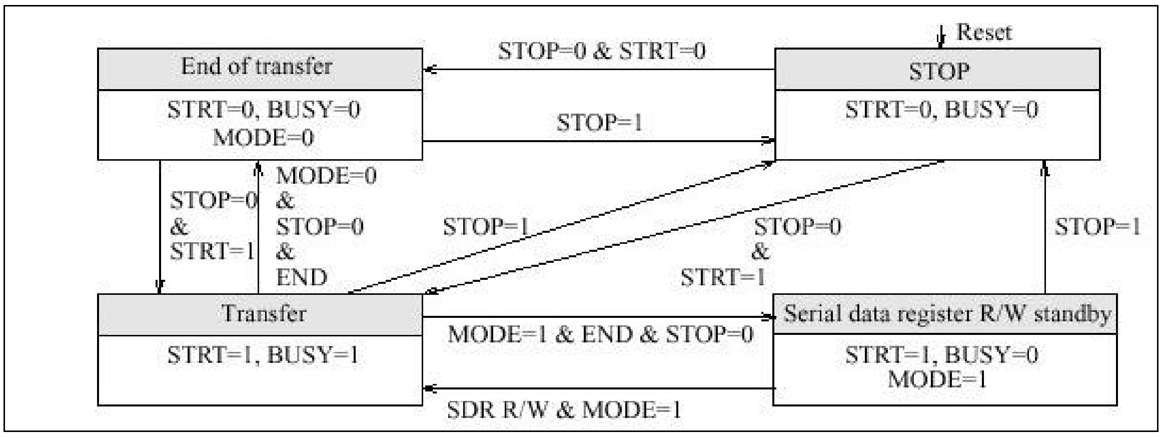


Figure 1

4.1 Configuration of the SIO:

The figure 2 shows the settings for the SIO to communicate with a SPI device.

```
void Init_SIO(void)
{
        SMCS = 0x3202;     /* (01100010xxxx0010) */
        Shift clock = 62.5kHz; interrupt disable; serial output enable
        SCDCR =0x88;       /* (10001000) prescaler */
        Division ration = 8; Prescaler enable
        DDR4 = 0xFF;       /* set all ports to outputs */
        DDR5 = 0x00;       /* set all ports to input */
}
```

Figure 2

5.0 SPI via Uart:

Not all the Fujitsu microcontroller have a SIO macro inside. But it is also possible to use the Uart to communicate with an SPI device.

The Fujitsu Uart macro offers two transfer modes asynchronous and synchronous. Not all of the Fujitsu Uart macros can be used for synchronous transfer to communicate via SPI. Only the Uart with serial communications interface (SCI) supports this, because other Uart macro uses in synchronous mode still a start and a stop bit.

Table 3 shows the devices, which contain the Uart macro to communicate with a SPI device.

Device	Uart (SCI)	MSB or LSB	Only LSB
470	Uart0	X	
495	Uart0/1	X	
520	Uart0		X
540/545	Uart1		X
550A	Uart0		X
560	Uart0/1	X	
570	Uart0/1		X
580	Uart0 – 4		X
590			X
595	Uart1		
610A	Uart0 – 2		X
620A	Uart0		X
630A	Uart0/1		X
640A	Uart0/1		X
650A	Uart0		X
660A	Uart0		X
670/675	Uart1		X

Table 3

Nearly all the Fujitsu Uart macros transfer data with the least significant bit (LSB) first. Therefore, the bits have to be switched by software to achieve transmit and receive with the MSB first. A few Uart macros offer a special feature to decide if the MSB or LSB should be transmit or received first. Table 3 shows which devices have this feature inside their Uart.

When an internal clock signal source (dedicated baud rate generator or internal timer) is selected for baudrate generation, a receive clock signal is automatically generated each time data is transmitted. Data length is 8-bit only, and no parity bit may be attached. Also, there is no start/stop bit, so that no error detection is enabled except for overrun errors.

5.1 Configuration of the Uart:

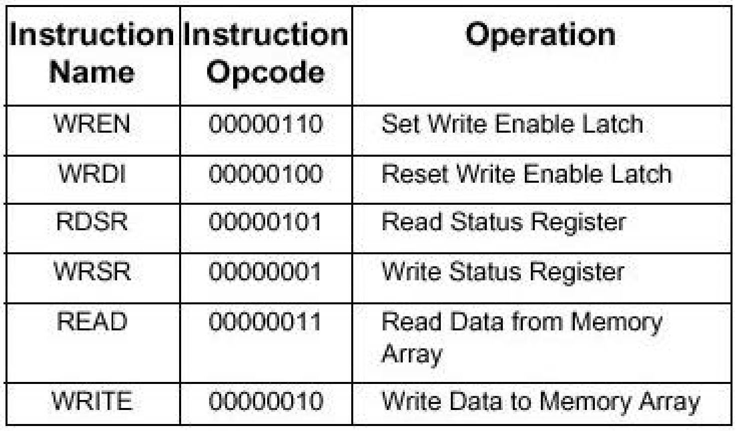
The figure 3 shows the settings for the Uart to communicate with a SPI device.

```
void InitUart_Master(void)
{
        /* initialize UART1 */
        SMR1 = 0xA3;        /* serial mode register (10100011) */
        SCR1 = 0x13;        /* serial control register (00010011) */
        U1CDCR = 0x88;      /* prescaler control register (10001000) */
        DDR4 = 0xE7;        /* port direction register (11100111) */
        WP = TRUE;          /* this pins have to held high */
        HOLD = TRUE;        /* */
}
```

Figure 3

6.0 Operation in general:

In this small circuit the Fujitsu MCU is the master and the EEPROM is acting as the slave. The table 4 shows the small instruction set the SPI device uses. These instructions are needed to communication with this device.



Instruction Name	Instruction Opcode	Operation
WREN	00000110	Set Write Enable Latch
WRDI	00000100	Reset Write Enable Latch
RDSR	00000101	Read Status Register
WRSR	00000001	Write Status Register
READ	00000011	Read Data from Memory Array
WRITE	00000010	Write Data to Memory Array

Table 4

The slave device will be in non-active state from the beginning on. To activate the device the chip select (CS) pin has to hold low. The CS, write protection (WP) and the HOLD pin are all low active.

Hardware data protection is provided by WP pin to protect against inadvertent programming. The HOLD pin allows the serial communication to be suspended the device without resetting the serial sequence.

The figure 4 shows a drawing of the connection between the MCU and the SPI device.

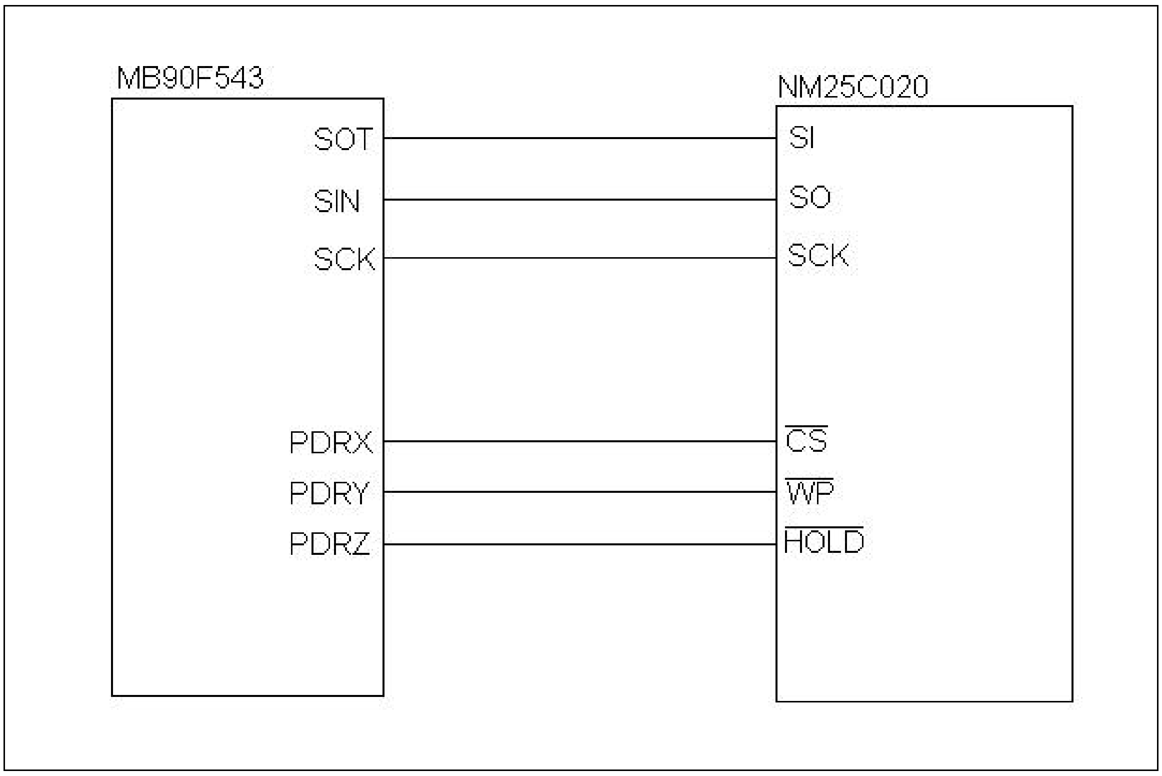


Figure 4

When CS is set to low, transmission can start now. First one of the instructions mentioned above has to be sent to the device. Depending on the op-code additional information must be sent to the EEPROM. To receive data from the SPI device the master must send a dummy out of the SDR. The way the dummy looks like depends on the used SPI device. In the used application the dummy was a simple 8bit of low values.
With this synchronous clock the data of the SPI is clocked in serially to the SDR. After the read or write cycle is finished the CS is pulled back to high level. Now a new device can be selected. For every device it is necessary to provide its own CS line.

<u>7.0 Difference when using the Uart:</u>

When using the Uart instead of the SIO, the general operation is the same. But for the Uart same extra things have to be into account.

First, according to the MCU, which is used, because the Uart starts transmission with the least significant bit (LSB) first the data have to be turn around. Only some Uart’s offer the possibility to determine the order in which the bits are shifted out. If the Uart does not support such a feature, there are two possibilities to tackle this problem.

One possibility is to do the changing of the shifting order by software. This means, all the information and data must be changed before transferred. And all the incoming data must be changed before using it inside the application. Too increase the speed performance, all the op-codes, if some special commands are used to communicate with the connected SPI device, e.g. EEPROM this could be stored in the reverse order. Also transferred data could be defined already in the vice versa order to avoid extra calculation time.

The other solution is to use external hardware. Therefore, a device has to be placed between the master and the slave. This device has to change the order of the bits. But this could be influencing the bandwidth of the SPI transfer.

The next difference between the SIO and the Uart is that if the Uart uses synchronous receive clock each time data is transmitted. The transmit/receive buffer of the SIO is the same therefore, each time data is written or read the buffer of the SIO, the buffer is cleared.
The Uart has independent buffers. Each time data is transferred, e.g. an op-code to the EEPROM, data is placed in the receive buffer. And as long as the data in the receive buffer remains there, no new data is written to the receive buffer, the buffer is blocked. This means, with every transmit signal the User-Application has to take care, that the receive buffer is clear after every transmission. The Uart has to send a dummy to receive the correct data out of the EEPROM.

<u>8.0 Conclusion:</u>

The SPI via SIO is much easier to implement compared to the Uart, but on the other hand it is possible to use the Uart to communicate with SPI devices. This can be helpful if the chosen device has no SIO macro inside.

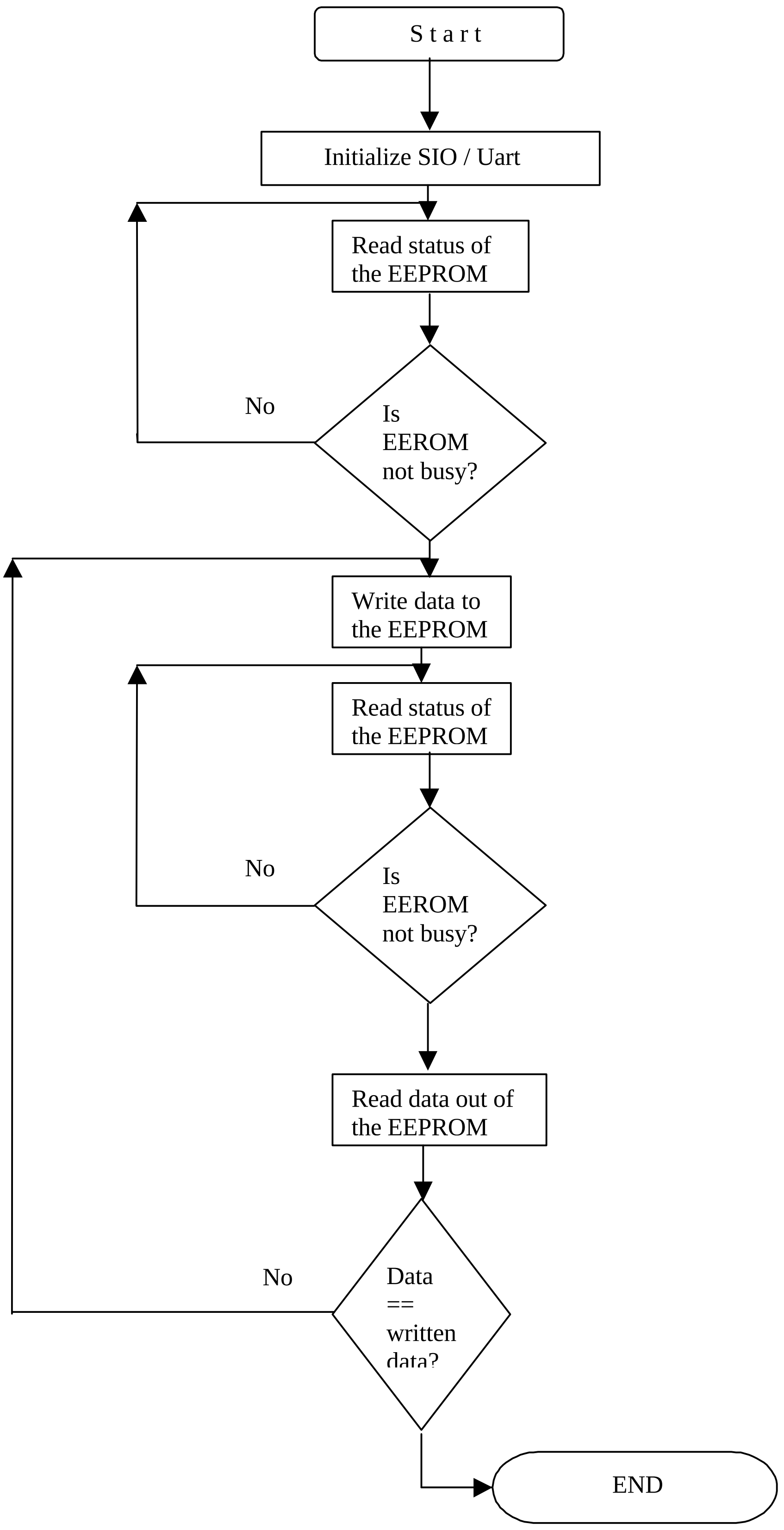
9.0 Appendix:

The Table 6 shows an overview of the Uart and SIO macros of the 16 Bit family.

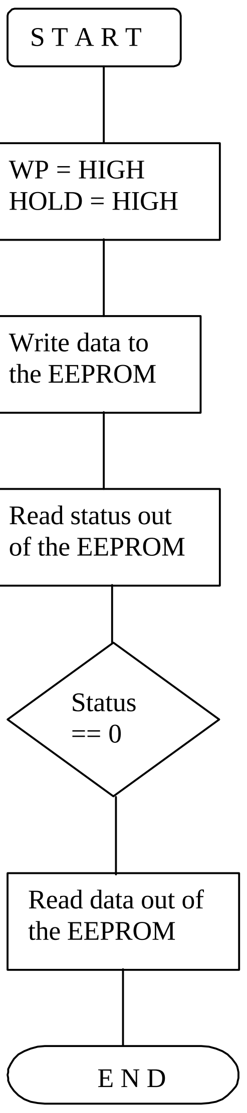
Device	Uart (SCI)	Serial I/O
470	Uart0*	
495	Uart0/1*	
520	Uart0	X
540/545	Uart1	X
550A	Uart0	X
560	Uart0/1*	
570	Uart0/1	X
580	Uart0 – 4	X
590		X
595	Uart1	
610A	Uart0 – 2	X
620A	Uart0	X
630A	Uart0/1	X
640A	Uart0/1	
650A	Uart0	X
660A	Uart0	
670/675	Uart1	

*Uart can select shifting order (MSB or LSB first)

Flowchart of the application:



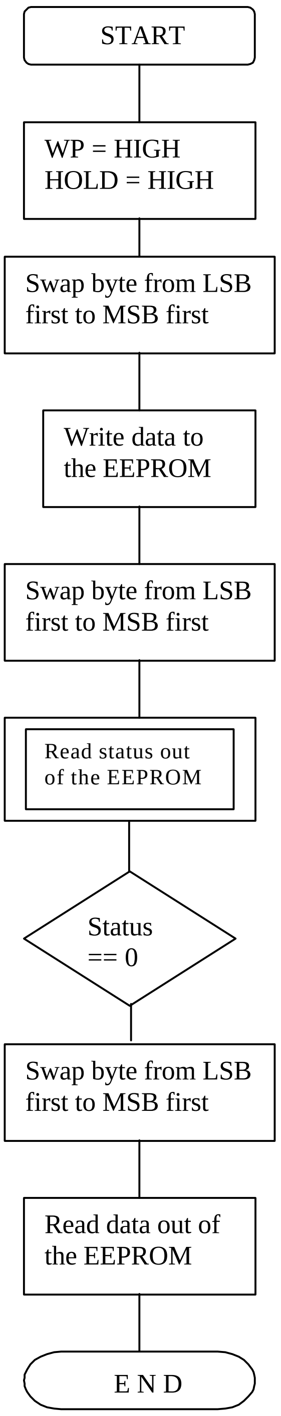
Flowchart of the SPI interface software with Serial I/O:



Program Code for SIO

```
/*-------------------------------------------------------------------
  MAIN.C
  - description
  - See README.TXT for project description and disclaimer.

/*---------------------------------------------------------------------*/

/************************* Includes *********************************/
#include "mb90540.h"
#include "spi.def"

/************************* Constants/Strings ************************/

/*************************** Gloabals ********************************/
uchar rbuf = 0;
uchar status = 0;
uchar tbuf = 0;

/************************ Prototyps *********************************/
void Init_SIO(void);
void read_status(void);
void read(void);
void write(void);
void start_seq(void);
void stop_seq(void);

/*======================= PROCEDURES ==============================*/
void Init_SIO(void)
{
      SMCS = 0x0207;    /* (00000010xxxx0111) 1MHz */
      SCDCR =0x8E;      /* (10001000) prescaler */
      DDR4 = 0xFF;
      DDR5 = 0x00;
}

void start_seq(void)
{
      /*start a synchron transfer */
      SMCS_STOP = FALSE;
      SMCS_STRT = TRUE;
      while(SMCS_BUSY == 1);  /* wait until transmit is complete */
}

void stop_seq(void)
{
      SMCS_STOP = TRUE;
      SMCS_STRT = FALSE;
      SMCS_BUSY = FALSE;
}

void read_status(void)
{
      /* reads status out of the EEPROM */
      CS = FALSE;             /* chipselect (CS) is active low */
      SDR = RDSR; /* write opcode into serial shift data register */
      start_seq();            /* start synchron transfer */
      SDR = DUMMY;/* write dummy into serial shift data register */
      start_seq();            /* start synchron transfer */
      status = SDR;           /* receive status of the EEPROM */
      stop_seq();             /* clear flags */
      CS = TRUE;              /* CS high */
}

void read(void)
{
      /* read data out of the memory */
      CS = FALSE;
      SDR = READ; /* write opcode into serial shift data register */
      start_seq();            /* start synchron transfer */
      SDR = ADDRESS;          /* send memory address */
      start_seq();
      SDR = DUMMY;/* write dummy into serial shift data register */
      start_seq();
      tbuf = SDR;             /* receive status of the EEPROM */
      stop_seq();
      CS = TRUE;
}

void write(void)
{
      CS = FALSE;
      SDR = WREN;             /* write enable*/
      start_seq();            /* start synchron transfer */
      CS = TRUE;              /* CS high */

      CS = FALSE;
      SDR = WRITE;/* write opcode into serial shift data register */
      start_seq();
      SDR = ADDRESS;
      start_seq();
      SDR = DATA;
      start_seq();
      stop_seq();             /* clear flags */
      CS = TRUE;
}

void write_disable(void)
{
      CS = FALSE;
      SDR = WRDI;
      start_seq();
      CS = TRUE;
}
/*===================== END OF PROCEDURES ===========================*/

/********************** M A I N ***********************************/
void main(void)
{
      Init_SIO();
    InitIrqLevels();
    __set_il(7);                /* allow all levels */
    __EI();                      /* globaly enable interrupts */

    HOLD = TRUE;        /* HOLD line */
    WP = TRUE;          /* Write Protected line */

    write_disable();    /* disable write */
    write();            /* write data to a specific memory address */

    while(1)
    {
    /* SPI Master */
      read_status();          /* read status */
      if((status & 1) == 0)   /* if device has finished writing data */
      {
            read();     /* read data from a specific memory address */
      }
    }
}
/************************* end of main *****************************/
```

Flowchart Uart for SPI:


START
WP = HIGH
HOLD = HIGH
Swap byte from LSB first to MSB first
Write data to the EEPROM
Swap byte from LSB first to MSB first
Read status out of the EEPROM
Status == 0
Swap byte from LSB first to MSB first
Read data out of the EEPROM
E N D

Program Code for Uart

```
/*-------------------------------------------------------------------
  MAIN.C
  - description
  - See README.TXT for project description and disclaimer.

/*-----------------------------------------------------------------*/

/************************* Includes ********************************/
#include "mb90540.h"
#include "spi.def"

/********************** Constants/Strings *************************/

/************************* Gloabals ********************************/
uchar rbuf = 0;
uchar status = 0;
uchar tbuf = 0;
const uchar Data[] = {0x51,0xA2,0x3F};
const uchar Address[] = {0x00,0x01,0x02};
int index = 0;

/************************* Prototyps *******************************/
void InitUart_Master(void);
uchar mirror_byte(uchar);
void write(void);
void write_enable(void);
void write_disable(void);
void read(void);
void read_status(void);
void clear_buffer(void);

/*========================= PROCEDURES ============================*/
void InitUart_Master(void)
{
      /* initialize UART1 */
      SMR1 = 0x83;      /* serial mode register (10000011) 1Mbit */
      SCR1 = 0x13;      /* serial control register (00010011) */
      U1CDCR = 0x88;    /* prescaler control register (10001000) */
      DDR4 = 0xE7;      /* port direction register (11100111) */
      WP = TRUE;        /* this pins have to held high */
      HOLD = TRUE;      /* */
}

/* This function is necessay to change the LSB and MSB for the
   the SPI device
               */
uchar mirror_byte(uchar value)
{
      /* mirror a byte */
      uchar temp,temp1;

      temp1 = value & 1;
      temp1 = temp1 << 7;
      temp = temp1;
      temp1 = value & 2;
      temp1 = temp1 << 5;
      temp = temp1 | temp;
      temp1 = value & 4;
      temp1 = temp1 << 3;
      temp = temp1 | temp;
      temp1 = value & 8;
      temp1 = temp1 << 1;
      temp = temp1 | temp;
      temp1 = value & 0x10;
      temp1 = temp1 >> 1;
      temp = temp1 | temp;
      temp1 = value & 0x20;
      temp1 = temp1 >> 3;
      temp = temp1 | temp;
      temp1 = value & 0x40;
      temp1 = temp1 >> 5;
      temp = temp1 | temp;
      temp1 = value & 0x80;
      temp1 = temp1 >> 7;
      temp = temp1 | temp;
      return(temp);
}

void read_status(void)
{
    uchar send_op_code;

    CS = FALSE;                     /* select device */

      send_op_code = mirror_byte(RDSR);   /* mirror byte */

      SODR1 = send_op_code;               /* send opcode */
      while(SSR1_TDRE == 0);  /* wait that tranmission is finished*/

      clear_buffer();               /* clear the receive buffer */

      SODR1 = 0x00;                 /* send a dummy to receive */
                                    /* the data from EEPROM*/
      while(SSR1_RDRF == 0);  /* wait that data is available */
      if(SSR1_ORE || SSR1_FRE)/* look for valid or invalid data */
      {
            rbuf = SIDR1;
            SCR1_REC = 0;                       /* clear flags */
      }
      else
      {
            rbuf = SIDR1;
            status = mirror_byte(rbuf);         /* mirror byte */
      }
      CS = TRUE;                          /* diselecte the device */
}

void write(void)
{
      uchar send_op_code;
```

```
        CS = FALSE;             /* selecte the device */

        send_op_code = mirror_byte(WRITE);  /* mirror byte */
        SODR1 = send_op_code;               /* send opcode */
        while(SSR1_TDRE == 0);  /* wait that tranmission is finished*/

        clear_buffer();               /* clear the receive buffer */

        send_op_code = mirror_byte(Address[index]);     /* mirror byte */
        SODR1 = send_op_code;                       /* send opcode */
        while(SSR1_TDRE == 0);  /* wait that tranmission is finished*/

        clear_buffer();               /* clear the receive buffer */

        send_op_code = mirror_byte(Data[index]);
        SODR1 = send_op_code;
        while(SSR1_TDRE == 0);

        clear_buffer();

        CS = TRUE;
}

void read(void)
{
        uchar send_op_code;

        CS = FALSE;

        send_op_code = mirror_byte(READ);         /* mirror byte */
        SODR1 = send_op_code;         /* send opcode */
        while(SSR1_TDRE == 0);  /* wait that tranmission is finished*/

        clear_buffer();               /* clear the receive buffer */

        send_op_code = mirror_byte(Address[index]);
        SODR1 = send_op_code;
        while(SSR1_TDRE == 0);

        clear_buffer();

        SODR1 = 0x00;     /* send a dummy to receive the answer */

        while(SSR1_RDRF == 0);  /* wait that data is available */
        if(SSR1_ORE || SSR1_FRE)/* look for valid or invalid data */
        {
              rbuf = SIDR1;
              SCR1_REC = 0;                 /* clear flags */
        }
        else
        {
              rbuf = SIDR1;
              tbuf = mirror_byte(rbuf);     /* mirror byte */
        }
        CS = TRUE;
}
```

```
void write_enable(void)
{
      uchar send_op_code;

      CS = FALSE;

      send_op_code = mirror_byte(WREN);         /* mirror byte */
      SODR1 = send_op_code;                     /* send opcode */
      while(SSR1_TDRE == 0);  /* wait that tranmission is finished*/

      clear_buffer();               /* clear the receive buffer */

      CS = TRUE;
}

void write_disable(void)
{
      uchar send_op_code;

      CS = FALSE;

      send_op_code = mirror_byte(WRDI);
      SODR1 = send_op_code;
      while(SSR1_TDRE == 0);

      clear_buffer();

      CS = TRUE;
}

void clear_buffer(void)
{
      while(SSR1_RDRF == 0);
      rbuf = SIDR1;                             /* clear buffer */
      rbuf = 0;
}

/*===================== END OF PROCEDURES ==========================*/

/************************* M A I N *********************************/
void main(void)
{
      InitUart_Master();
    InitIrqLevels();
    __set_il(7);                /* allow all levels */
    __EI();                     /* globaly enable interrupts */


      for(index = 0;index < 3;index++)
      {
            write_enable();         /* enables a write cycle */
            write();                /* writes data to the memory */
            read_status();          /* read status of the device */
            while((status & 1) == 1)/* wait that device become ready */
                  read_status();
      }
      write_disable();              /* disables the writing */

      while(1)
    {
    /* SPI Master */

            if(index >= 3)
                  index = 0;
            read_status();          /* read status */
            read();                 /* read data from the EEPROM */
            index++;
     }
}
/********************** end of main *********************************/
```