MICROCHIP

AN1243

Low Latency Driver to Access External EEPROM Using PIC18 Family Devices

Authors: *Obul Reddy and Ganesh Krishna S.M*
Microchip Technology Inc.

INTRODUCTION

This application note is developed based on low latency design. It provides an algorithm, which is designed to use the SPI/I^{2}C™ interrupts, to achieve the required communication and enable optimum processor usage. The algorithm is developed based on the PIC18 Master Synchronous Serial Port (MSSP) module with external Serial Peripheral Interface (SPI) EEPROMs and I^{2}C EEPROMs, respectively. The algorithm uses an interrupt driven approach.

OVERVIEW OF LOW LATENCY DESIGN

The low latency design relies on the communication interrupts provided by the PIC® MCUs to extract maximum performance from the microcontroller. This design can be better understood by first investigating the conventional approach and its disadvantages in the following sections.

Existing Conventional Approach

The conventional approach is to write blocking routines that do not relinquish control when they are awaiting an external event. The blocking routines are merely polling for flags to get triggered by the hardware. Therefore, the microcontroller is always busy with execution while waiting for a flag to get triggered.

In SPI mode, the microcontroller is always busy monitoring the Buffer Full (BF) flag/status bit of the MSSP Status (SSPSTAT) register during communications between the PIC MCU and external serial EEPROMs. In I^{2}C mode, the BF status bit gets cleared during transmission, and gets set during reception.

Disadvantages of Conventional Approach

External EEPROM chips, connected via SPI or I^{2}C, tend to consume a lot of microcontroller throughput to communicate. The routines accessing the EEPROM will have to wait until the communication is reliably completed. During this period, the microcontroller remains idle when it can actually be performing other tasks. The applications developed using the conventional approach do not allow the microcontroller to perform other tasks parallely. As this approach requires continuous and dedicated monitoring of the task, it degrades the performance and throughput of the microcontroller by wasting clock cycles.

LOW LATENCY DESIGN

The limitations of the conventional approach can be overcome by following the low latency approach. As the MSSP module comprises both SPI and I^{2}C modes, the microcontroller can operate in one of the two modes (either in SPI or I^{2}C). There is no need to poll the BF status bit continuously as this design uses the interrupt flag (i.e., MSSP Interrupt Flag bit – SSPIF) provided by the MSSP hardware module.

Note:	The MSSP module in PIC18 can be configured to use either the SPI or I^{2}C module.

As soon as the transmission/reception is completed, the SSPIF interrupt flag gets triggered by the hardware and vectors to the Interrupt Service Routine (ISR). To achieve this, the MSSP module interrupt must be enabled along with the global interrupt enable. Thus, communication happens in the background inside the ISR. This, in turn, reduces the load on the microcontroller and enables other tasks to run in a pseudo parallel control flow.

The low latency design is comprised of the following software stacks:

- SPI Software Stack – Comprises Application Layer, EEPROM Driver Layer, SPI Driver Layer and Hardware Layer
- I^{2}C Software Stack – Comprises Application Layer, EEPROM Driver Layer, I^{2}C Driver Layer and Hardware Layer

IMPLEMENTATION

SPI Software Stack

In this implementation, the MSSP module is configured as SPI and is interfaced with Microchip's 25XXX series SPI serial EEPROM device.

Figure 1 displays the layer-wise SPI software stack implementation.

Figure 2 displays the hardware schematic for the interface between the PIC18 MCU and Microchip's 25XXX series devices. The schematic provides the necessary connections between the microcontroller and the tested serial EEPROM; the software is written assuming these connections. The $\overline{WP}$ and $\overline{HOLD}$ pins are tied to V_{CC}, since these are not used in the software stack.

FIGURE 1: SPI SOFTWARE STACK

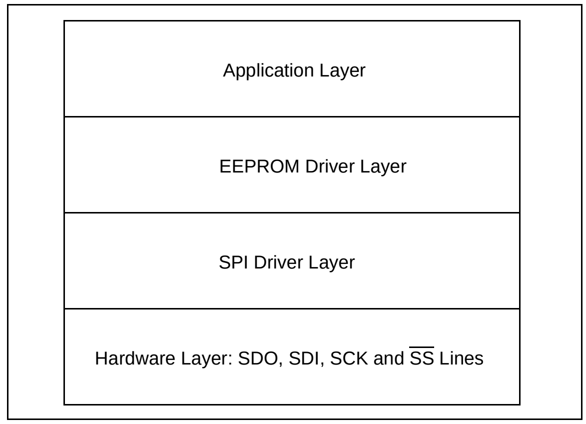
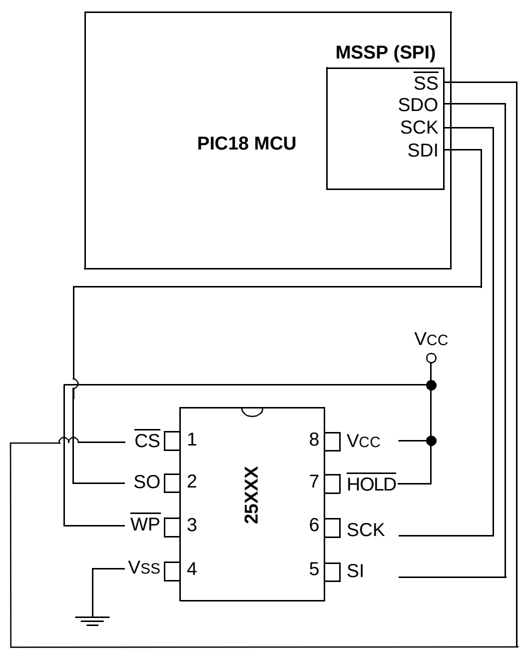


FIGURE 2: CIRCUIT FOR PIC18 MCU AND 25XXX SERIES DEVICE



I²C Software Stack

The MSSP module is configured as I²C and is interfaced with Microchip's 24XXX series' I²C serial EEPROM device.

Figure 3 displays the hardware schematic for the interface between the PIC18 MCU and Microchip's 24XXX series devices. The schematic provides the connections necessary between the microcontroller and the serial EEPROM; the software is developed assuming these connections. As the SDA and SCL pins are open-drain terminals, they require pull-up resistors to VCC (typically, 10 kΩ for 100 kHz and 2 kΩ for 400 kHz and 1 MHz). The $\overline{WP}$ pin is tied to ground as the write-protect feature is not used in the software stack provided.

FIGURE 3: CIRCUIT FOR PIC18 MCU AND 24XXX SERIES DEVICE

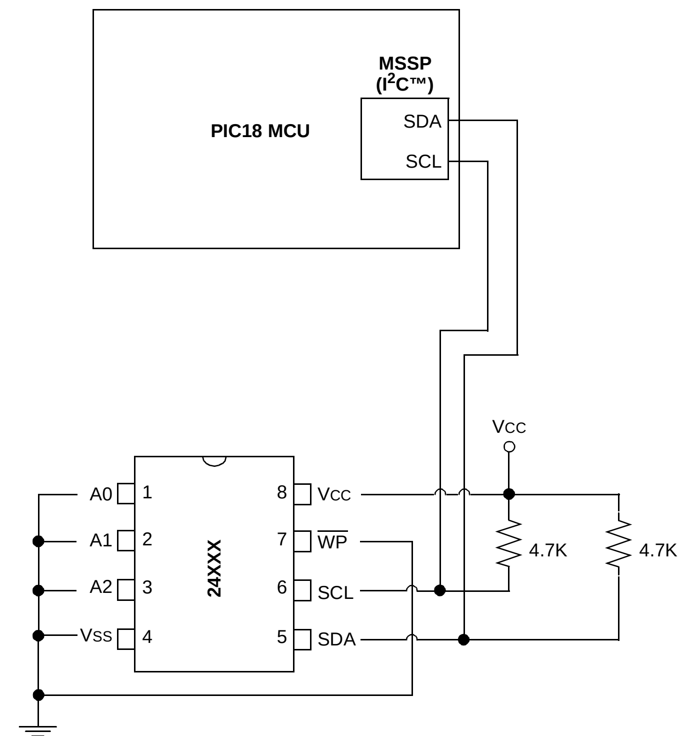
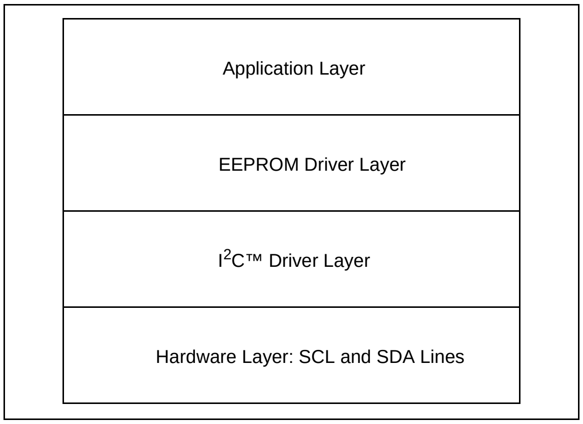


Figure 4 displays the layer-wise I^{2}C software stack implementation.

FIGURE 4: I^{2}C™ SOFTWARE STACK



FIRMWARE

- SPI Module – The source code consists of three files (`main.c`, `ee_drv.c` and `spi_drv.c`), which fit into the corresponding layers based on file operation.
- I^{2}C Module – The source code consists of three files (`main.c`, `ee_drv.c` and `i2c_drv.c`), which fit into the corresponding layers based on file operation.

APPLICATION LAYER

The application layer (`main.c`), in both SPI and I^{2}C modes, consists of API calls to initialize, write, read and verify the SPI and I^{2}C EEPROM devices. The main API calls are `EE_Init()`, `EE_Write()`, `EE_Read()` and `EE_Verify()`. The other two APIs associated with the main APIs are `EE_Status()` and `EE_Task()`. API `EE_Status()` returns the current status of the EEPROM operation. API `EE_Task()` updates the EEPROM with respect to the operation of the main API and the current status of EEPROM.

EEPROM DRIVER LAYER

The application APIs are defined in the EEPROM driver layer (`ee_drv.c`). API, `EE_Init()`, initializes the EEPROM, `EE_Write()` writes the requested number of bytes to the given EEPROM address, `EE_Read()` reads the requested number of bytes from the given EEPROM address and `EE_Verify()` verifies the number of bytes against the contents of EEPROM at the given address. API, `EE_Status()`, returns the current status of the EEPROM operation and must be called before each read/write to ensure that the driver is free. It must be called after every read to ensure that the data has been successfully copied to the user's space.

API, `EE_Task()`, is implemented as the main (high-level) EEPROM driver. The driver runs through different states to get the SPI/I^{2}C EEPROM read/write done using the low-level SPI/I^{2}C driver.

- SPI Driver

  The SPI driver chops the EEPROM writes into page sizes. The driver waits until the EEPROM chip is ready between consecutive page writes by reading the Write-In-Process (WIP) bit of the status register in the EEPROM. The WIP bit indicates whether the EEPROM is busy with an internal write operation. The driver resets the EEPROM in case of errors.

- I^{2}C Driver

  The I^{2}C driver chops the EEPROM writes into page sizes. The driver waits until the EEPROM chip is ready between consecutive page writes by polling the EEPROM device. The Acknowledgement (ACK) polling between page writes is required to determine whether the external EEPROM device is busy with its internal write operation. The driver also resets the EEPROM device in case of errors.

LOW-LEVEL DRIVER LAYER

- SPI Driver Layer (`spi_drv.c`) – It initializes the SPI module (`SPI_Init()`), disables the module and re-enables it in case of errors (`Reset_EE_Chip()`), and implements the low-level SPI driver. The low-level SPI driver is a semi-generic state machine implemented as an ISR, which goes through the necessary states to construct an SPI frame. In case the interrupts are shared among different modules, this routine must be called when the root ISR spots that SSPIF is set.
- I^2C Driver Layer (`i2c_drv.c`) – It initializes the I^2C module (`I2C_Init()`), disables the module and re-enables it in case of errors (`Reset_EE_Chip()`), and implements the low-level I^2C driver. Like the SPI driver, the low-level I^2C driver is a semi-generic state machine implemented as an ISR, which goes through the necessary states to construct an I^2C frame. If the interrupts are shared among different modules, this routine must be called when the root ISR spots that SSPIF is set.

HARDWARE LAYER

- SPI Module – Whenever the MSSP module is enabled and configured for SPI mode in the device, it configures the SCK, SDO, SDI and $\overline{SS}$ pins as serial port pins. These pins are used by the MSSP hardware module during SPI communications.
- I^2C Module – Whenever the MSSP module is enabled and configured for I^2C Master mode in the device, it configures the SCL and SDA pins as serial port pins. In Master mode, the SCL and SDA lines are used by the MSSP hardware during I^2C communications.

LATENCY DETAILS

- SPI Driver – Table 1 provides the latency details based on an oscillator frequency of 10 MHz for 1-byte write, read and verify.

TABLE 1: LATENCY DETAILS FOR SPI DRIVER

Fosc = 10 MHz			
API	**Performance Time (μs)**	**Performance Time (μs) (Fosc with PLL)**	**Comments**
`EE_Init()`	18	4.4	Main API
`EE_Task()`	11.2	2.8	Associated API
`EE_Write()`	296	74	Main API
`EE_Task()`	20	5	Associated API
`EE_Read()`	168	42	Main API
`EE_Task()`	12.6	3.2	Associated API
`EE_Verify()`	180	46	Main API
`EE_Task()`	12.8	3.2	Associated API

- I²C Driver – Table 2 provides the latency details based on an oscillator frequency of 10 MHz for 1-byte write, read and verify.

TABLE 2: LATENCY DETAILS FOR I²C™ DRIVER

FOSC = 10 MHz			
API	**Performance Time (μs)**	**Performance Time (μs) (FOSC with PLL)**	**Comments**
`EE_Init()`	15.6	3.9	Main API
`EE_Task()`	28.8	7.2	Associated API
`EE_Write()`	94	23.2	Main API
`EE_Task()`	11.6	2.9	Associated API
`EE_Read()`	94	23.6	Main API
`EE_Task()`	10.8	2.7	Associated API
`EE_Verify()`	110	27.2	Main API
`EE_Task()`	10.8	2.7	Associated API

API DETAILS

EE_Init()

Initializes the MSSP module and the external EEPROM chip.

Syntax

```
void EE_Init (void)
```

Parameters

None

Return Values

None

Example

```
void main(void)
{

    // Function to initialize the MSSP and external EEPROM
    EE_Init();
    while (EE_Status() == EE_BUSY)
    {
        EE_Task();
        // Perform any other task here
    }
}
```

EE_Write()

Writes the requested number of bytes to the given EEPROM address.

Syntax

```
void EE_Write(unsigned int address, unsigned char *data, unsigned int numbytes)
```

Parameter

`address` – Address on EEPROM chip to write to

`data` – Location from where data must be copied

`numbytes` – Number of bytes to be written

Return Values

None

Example

```
unsigned int Address = 0x0000;
unsigned int Length = 6
unsigned char WriteString[6] = {0x1,0x2,0x3,0x4,0x5,0x6};

void main(void)
{
    // Function to write data into EEPROM
    EE_Write(Address, WriteString, Length);
    while (EE_Status() == EE_BUSY)
    {
        EE_Task();
        // Perform any other task here
    }
}
```

EE_Read()

Reads the requested number of bytes from the given EEPROM address.

Syntax

```
void EE_Read(unsigned char *data, unsigned int address, unsigned int numbytes)
```

Parameter

`data` – Location where the read data will be copied
`address` – Address on EEPROM chip to read from
`numbytes` – Number of bytes to read

Return Values

None

Example

```
unsigned int Address = 0x0000;
unsigned int Length = 6
unsigned char ReadString[6]   = {0,0,0,0,0,0};

void main(void)
{
   // Function to read data from EEPROM
   EE_Read(ReadString, Address, Length);
   while (EE_Status() == EE_BUSY)
   {
      EE_Task();
      // Perform any other task here
   }
}
```

EE_Verify()

Verifies contents of a buffer against the contents of the EEPROM.

Syntax

```
void EE_Verify(unsigned char *data, unsigned int address, unsigned int numbytes)
```

Parameter

`data` – Location of data bytes to verify against EEPROM contents
`address` – Address on EEPROM chip to verify from
`numbytes` – Number of bytes to verify

Return Values

None

Example

```
unsigned int Address = 0x0000;
unsigned int Length = 6
unsigned char VerifyString[6] = {0x1,0x2,0x3,0x4,0x5,0x6};

void main(void)
{
   EE_Verify(VerifyString, Address, Length);
   while (EE_Status() == EE_BUSY)
   {
      EE_Task();
      // Perform any other task here
   }
}
```

EE_Status()

Returns the current status of the EEPROM operation.

Syntax

`EE_Result_Type EE_Status (void)`

Parameter

None

Return Values

Returns current state of EEPROM module.

Example:

```
unsigned int Address = 0x0000;
unsigned int Length = 6
unsigned char WriteString[6] = {0x1,0x2,0x3,0x4,0x5,0x6};
typedef enum {EE_BUSY,EE_ERROR,EE_VERIFY_FAIL,EE_FREE}EE_Result_Type;

void main(void)
{
   EE_Write(Address, WriteString, Length);
   while (EE_Status() == EE_BUSY)
   {
      EE_Task();
      // Perform any other task here
   }
}
```

EE_Task()

This API runs through different states to get the SPI/I^{2}C EEPROM reads/writes done using the low-level SPI/I^{2}C driver, respectively.

Syntax

`void EE_Task (void)`

Parameter

None

Return Values

None

Example

```
unsigned int Address = 0x0000;
unsigned int Length = 6
unsigned char WriteString[6] = {0x1,0x2,0x3,0x4,0x5,0x6};

void main(void)
{
   EE_Write(Address, WriteString, Length);
   while (EE_Status() == EE_BUSY)
   {
      EE_Task();
      // Perform any other task here
   }
}
```

SPI Software Stack Control Flow

See Figure 5 for EEPROM driver control flow and Figure 6 for SPI driver control flow.

FIGURE 5: EEPROM DRIVER CONTROL FLOW

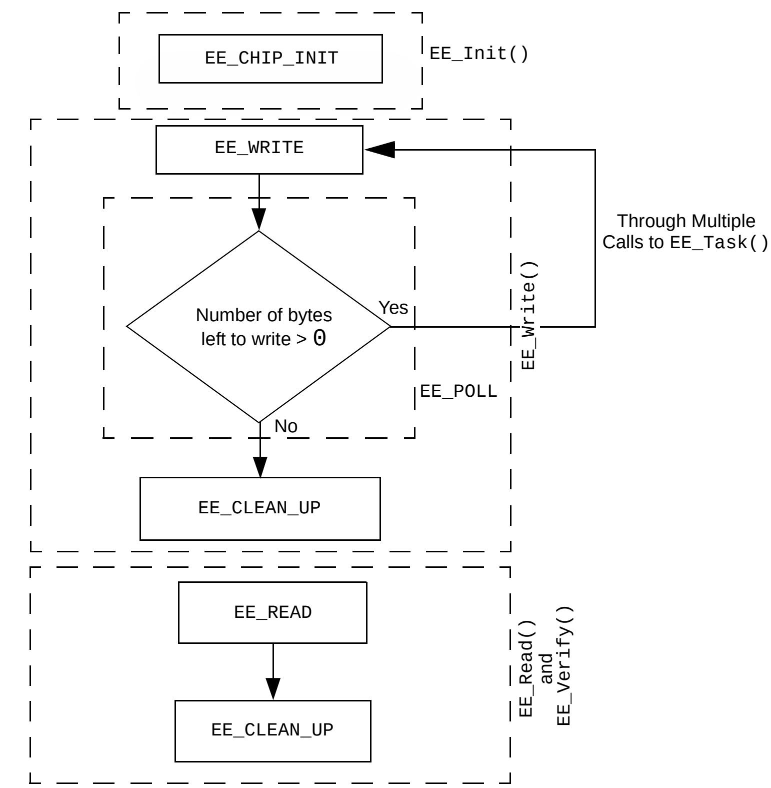
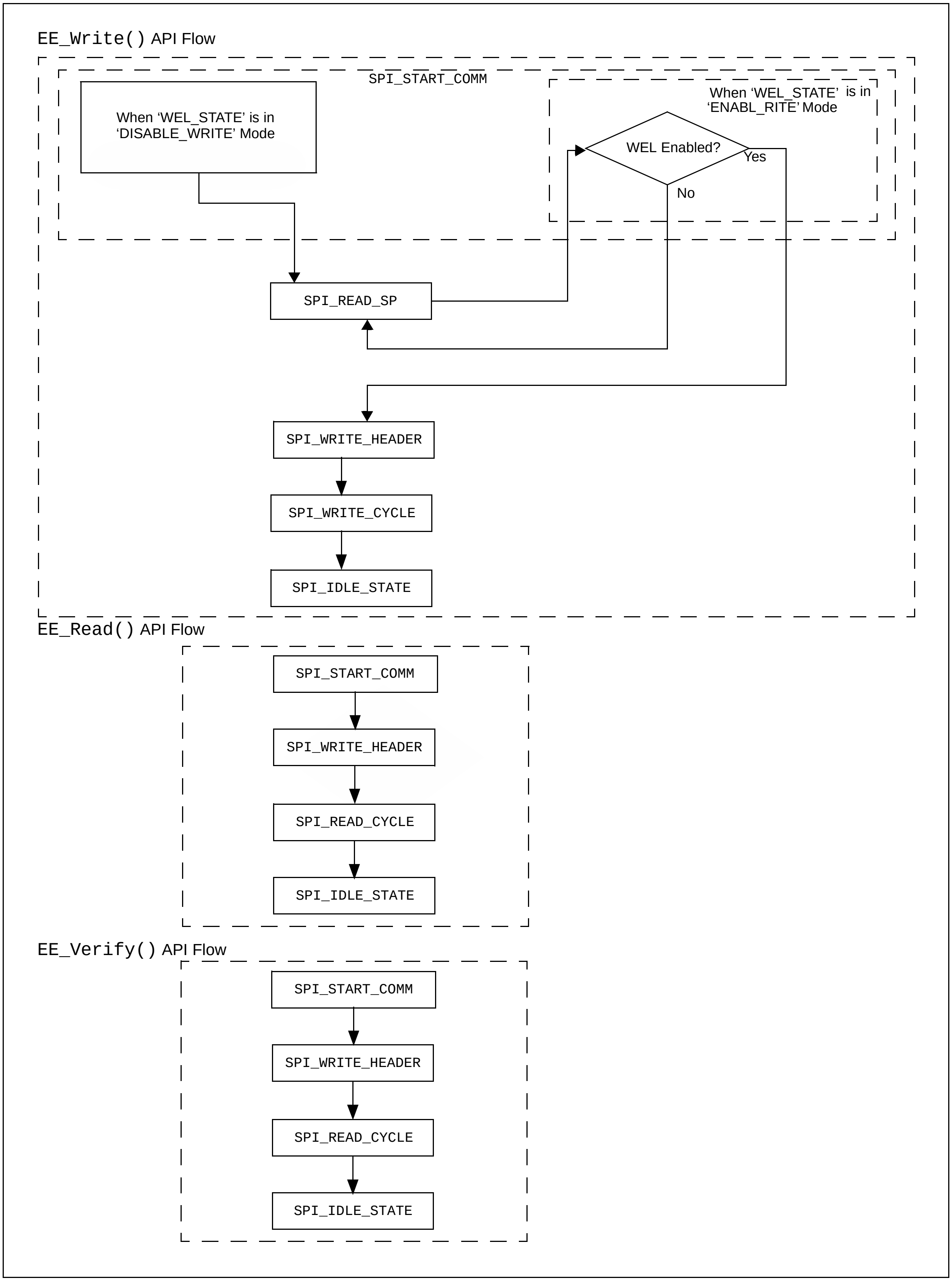


FIGURE 6: SPI DRIVER CONTROL FLOW



I²C Software Stack Control Flow

See Figure 7 for EEPROM driver control flow and Figure 8 for SPI driver control flow.

FIGURE 7: EEPROM DRIVER CONTROL FLOW

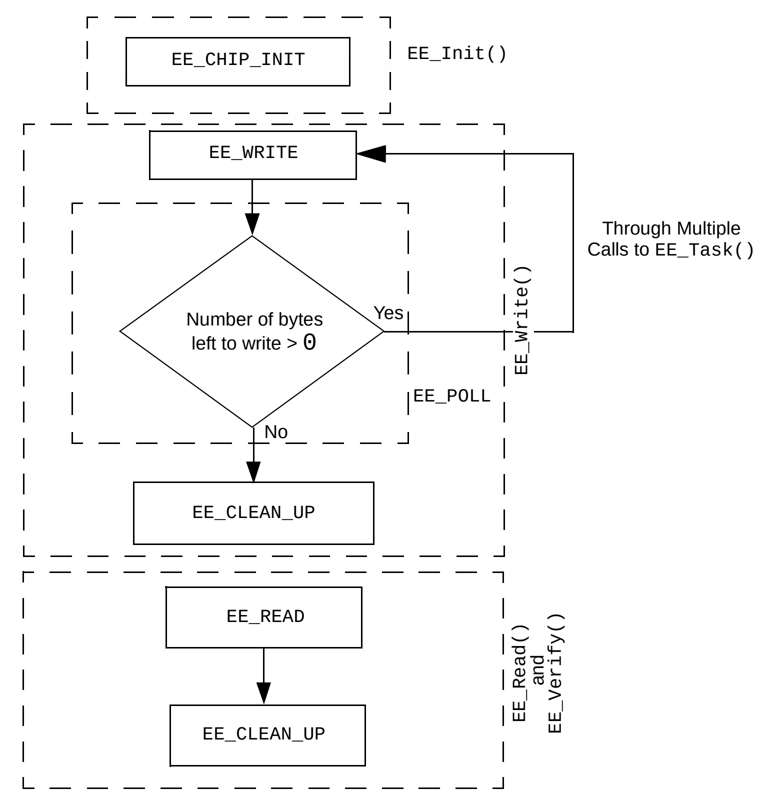
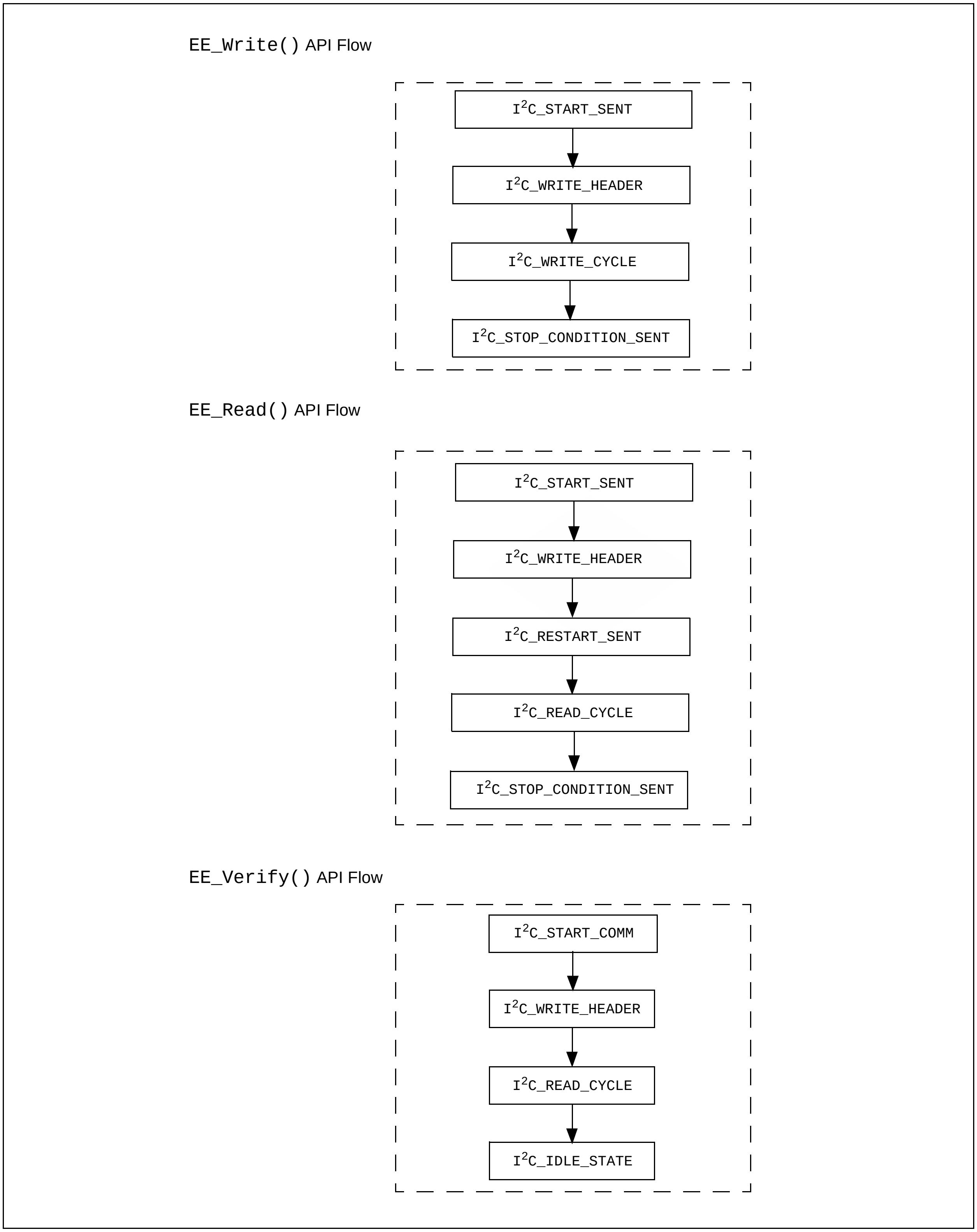


FIGURE 8: I^2C™ DRIVER CONTROL FLOW



CONCLUSION

This application note outlines an algorithm, which uses MSSP module interrupts available in the PIC18 family of devices, to overcome the limitations of the conventional approach by following the low latency design.

REFERENCES

- *AN1000, "Using the MSSP Module to Interface SPI Serial EEPROMs with PIC18 Devices"* – www.microchip.com
- *AN989, "Using the MSSP Module to Interface I^2C™ Serial EEPROMs with PIC18 Devices"* – www.microchip.com

APPENDIX A: LIBRARY DIRECTORY

TABLE A-1: LIBRARY DIRECTORY ORGANIZATION

Directory	Content
Low_Lat_DATAEE_soln:	
I^2C_solution	A Low Latency Data EEPROM Solution for I^2C™ EEPROM Chips
SPI_solution	A Low Latency Data EEPROM Solution for SPI EEPROM Chips

NOTES:

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

MICROCHIP

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/02/08