INTRODUCTION

This application note is developed based on low latency design. It provides an algorithm, which is designed to use the SPI/I²C™ 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 I²C 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 I²C 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 I²C, 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 I²C modes, the microcontroller can operate in one of the two modes (either in SPI or I²C). 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.

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
- I²C Software Stack – Comprises Application Layer, EEPROM Driver Layer, I²C 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 WP and HOLD pins are tied to Vcc, 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 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 I²C software stack implementation.

**FIGURE 4: I²C™ SOFTWARE STACK**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Layer</td>
<td></td>
</tr>
<tr>
<td>EEPROM Driver Layer</td>
<td></td>
</tr>
<tr>
<td>i²C™ Driver Layer</td>
<td></td>
</tr>
<tr>
<td>Hardware Layer: SCL and SDA Lines</td>
<td></td>
</tr>
</tbody>
</table>

**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.
- **I²C 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 I²C modes, consists of API calls to initialize, write, read and verify the SPI and I²C 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/I²C EEPROM read/write done using the low-level SPI/I²C 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.

- **I²C Driver**
  
  The I²C 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.

- I2C Driver Layer (i2c_drv.c) – It initializes the I2C module (I2C_Init()), disables the module and re-enables it in case of errors (Reset_EE_Chip()), and implements the low-level I2C driver. Like the SPI driver, the low-level I2C driver is a semi-generic state machine implemented as an ISR, which goes through the necessary states to construct an I2C 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 SS pins as serial port pins. These pins are used by the MSSP hardware module during SPI communications.

- I2C Module – Whenever the MSSP module is enabled and configured for I2C 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 I2C 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>
<thead>
<tr>
<th>API</th>
<th>Performance Time (μs)</th>
<th>Performance Time (μs) (Fosc with PLL)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE_Init()</td>
<td>18</td>
<td>4.4</td>
<td>Main API</td>
</tr>
<tr>
<td>EE_Task()</td>
<td>11.2</td>
<td>2.8</td>
<td>Associated API</td>
</tr>
<tr>
<td>EE_Write()</td>
<td>296</td>
<td>74</td>
<td>Main API</td>
</tr>
<tr>
<td>EE_Task()</td>
<td>20</td>
<td>5</td>
<td>Associated API</td>
</tr>
<tr>
<td>EE_Read()</td>
<td>168</td>
<td>42</td>
<td>Main API</td>
</tr>
<tr>
<td>EE_Task()</td>
<td>12.6</td>
<td>3.2</td>
<td>Associated API</td>
</tr>
<tr>
<td>EE_Verify()</td>
<td>180</td>
<td>46</td>
<td>Main API</td>
</tr>
<tr>
<td>EE_Task()</td>
<td>12.8</td>
<td>3.2</td>
<td>Associated API</td>
</tr>
</tbody>
</table>
• **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**

<table>
<thead>
<tr>
<th>API</th>
<th>Performance Time (µs)</th>
<th>Performance Time (µs) (Fosc with PLL)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE_Init()</td>
<td>15.6</td>
<td>3.9</td>
<td>Main API</td>
</tr>
<tr>
<td>EE_Task()</td>
<td>28.8</td>
<td>7.2</td>
<td>Associated API</td>
</tr>
<tr>
<td>EE_Write()</td>
<td>94</td>
<td>23.2</td>
<td>Main API</td>
</tr>
<tr>
<td>EE_Task()</td>
<td>11.6</td>
<td>2.9</td>
<td>Associated API</td>
</tr>
<tr>
<td>EE_Read()</td>
<td>94</td>
<td>23.6</td>
<td>Main API</td>
</tr>
<tr>
<td>EE_Task()</td>
<td>10.8</td>
<td>2.7</td>
<td>Associated API</td>
</tr>
<tr>
<td>EE_Verify()</td>
<td>110</td>
<td>27.2</td>
<td>Main API</td>
</tr>
<tr>
<td>EE_Task()</td>
<td>10.8</td>
<td>2.7</td>
<td>Associated API</td>
</tr>
</tbody>
</table>
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

```c
EE_Result_Type EE_Status (void)
```

Parameter

None

Return Values

Returns current state of EEPROM module.

Example:

```c
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/i2C EEPROM reads/writes done using the low-level SPI/i2C driver, respectively.

Syntax

```c
void EE_Task (void)
```

Parameter

None

Return Values

None

Example

```c
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

EE_Write() API Flow

SPI_START_COMM

When 'WEL_STATE' is in 'DISABLE_WRITE' Mode

SPI_READ_SP

SPI_WRITE_HEADER

SPI_WRITE_CYCLE

SPI_IDLE_STATE

WEL Enabled?

Yes

No

EE_Read() API Flow

SPI_START_COMM

SPI_WRITE_HEADER

SPI_READ_CYCLE

SPI_IDLE_STATE

EE_Verify() API Flow

SPI_START_COMM

SPI_WRITE_HEADER

SPI_READ_CYCLE

SPI_IDLE_STATE
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**

通过多个 EE_Task() 调用

- EE_CHIP_INIT
- EE_Write()
- EE_POLL
- EE_CHIP_INIT

- EE_READ
- EE_Verify()
- EE_CHIP_INIT

- EE_WRITE
- EE_Init()
- EE_POLL
- Number of bytes left to write > 0
  - Yes
  - Through Multiple Calls to EE_Task()
  - EE_Write()
  - EE_POLL
  - EE_CHIP_INIT
- No
- EE_CLEAN_UP
- EE_CHIP_INIT
FIGURE 8: \textsuperscript{\textsc{I}^{2}\textsc{C}} \texttrademark \textsc{TM} DRIVER CONTROL FLOW

EE\_Write() API Flow

1. \textsc{I}^{2}\textsc{C}\_START\_SENT
2. \textsc{I}^{2}\textsc{C}\_WRITE\_HEADER
3. \textsc{I}^{2}\textsc{C}\_WRITE\_CYCLE
4. \textsc{I}^{2}\textsc{C}\_STOP\_CONDITION\_SENT

EE\_Read() API Flow

1. \textsc{I}^{2}\textsc{C}\_START\_SENT
2. \textsc{I}^{2}\textsc{C}\_WRITE\_HEADER
3. \textsc{I}^{2}\textsc{C}\_RESTART\_SENT
4. \textsc{I}^{2}\textsc{C}\_READ\_CYCLE
5. \textsc{I}^{2}\textsc{C}\_STOP\_CONDITION\_SENT

EE\_Verify() API Flow

1. \textsc{I}^{2}\textsc{C}\_START\_COMM
2. \textsc{I}^{2}\textsc{C}\_WRITE\_HEADER
3. \textsc{I}^{2}\textsc{C}\_READ\_CYCLE
4. \textsc{I}^{2}\textsc{C}\_IDLE\_STATE
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 i2C℠ Serial EEPROMs with PIC18 Devices” – www.microchip.com
**APPENDIX A: LIBRARY DIRECTORY**

**TABLE A-1: LIBRARY DIRECTORY ORGANIZATION**

<table>
<thead>
<tr>
<th>Directory</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low_Lat_DATAEE_soln:</td>
<td>A Low Latency Data EEPROM Solution for ( \text{i}^2\text{C} )™ EEPROM Chips</td>
</tr>
<tr>
<td>I²C_solution</td>
<td>A Low Latency Data EEPROM Solution for ( \text{i}^2\text{C} )™ EEPROM Chips</td>
</tr>
<tr>
<td>SPI_solution</td>
<td>A Low Latency Data EEPROM Solution for SPI EEPROM Chips</td>
</tr>
</tbody>
</table>
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.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KeelLoq, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, rfPIC, SmartShunt and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, PicKit, PICDEM, PICDEM.net, PicTail, PicC™ logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rFAB, Select Mode, Total Endurance, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2008, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.

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.
## 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

### 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

### 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

### WorldWide Sales and Service

01/02/08