AN3401

Using Microchip I²C EERAMs with MPLAB® X and MPLAB® Code Configurator

Author: *Dragos Ciofu*
Microchip Technology Inc.

INTRODUCTION

Most embedded control systems require memory and the most common type used is nonvolatile memory. Serial EEPROMs are a popular choice for nonvolatile storage due to their flexibility and low pin count, power consumption and cost.

However, the drawback of any EEPROM device, regardless of manufacturer, is its limited endurance. Microchip Technology uses refined processes with tried and tested technology to ensure that parameters, such as endurance, are competitive in the memory environment. Microchip's extensive characterization processes ensure that data sheet parameters are met and exceeded.

Endurance is also highly dependent on the actual application being developed. Workarounds such as page-cycling or wear leveling can be implemented in order to extend the lifetime of a part. If you are working with EEPROMs for endurance estimates in a specific application, consult the Total Endurance™ Model which can be obtained on the Microchip website at www.microchip.com.

A newly developed product, Microchip's EERAM, is made of a volatile SRAM array mirrored by a nonvolatile EEPROM array. For more information on EERAM, see https://www.microchip.com/design-centers/memory/serial-eeram

THE I²C PROTOCOL

The Inter-Integrated Circuit (I²C) bus is a widely used industry standard bus used in the transfer of data between integrated circuits, such as between serial EERAMs and microcontrollers. Devices communicate in a master/slave environment in which the master always initiates the communication and the slave device is controlled through addressing. For a broader overview of the I²C protocol, see **Appendix B: "I2C Overview"** at the end of this document.

THE I²C EERAM

To correctly use and deploy I²C EERAMs in a robust project, understanding the following topics is recommended:

- Chip Address Inputs
- Bus Pull-Up Resistors
- Power Supply
- Hardware/Software Store Operations
- External Capacitor Choice

If you are not already familiar with these topics, refer to **Appendix C: "EERAM Characteristics"**, which explains how they impact your project.

MICROCHIP EERAM

To address the inherent issue of endurance in EEPROMS, Microchip Technology has developed EERAM. EERAM is a serial SRAM product that has hidden nonvolatile bits. On any power disruption, the active SRAM array content is moved to the nonvolatile bits. When power is restored, the nonvolatile content is restored to the SRAM array and SRAM operation continues. This model provides unlimited reads and writes to the SRAM cells, using a standard protocol and symmetrical read/write timings, while at the same time providing a transparent mechanism for data retention on power loss or upon request.

MPLAB® CODE CONFIGURATOR

The MPLAB Code Configurator (MCC) is a free, graphical programming environment that generates seamless, easy-to-understand C code that can easily be integrated into a project. Using an intuitive interface, it enables and configures a rich set of peripherals and functions specific to the user's application. It supports AVR® microcontrollers and 8-bit, 16-bit and 32-bit PIC® microcontrollers. MCC is incorporated into both the downloadable MPLAB® X IDE and the cloud-based MPLAB® Xpress IDE.

This application note intends to demonstrate how to interface I²C EERAM devices using MPLAB X 5.05 or later, the XC series of compilers and the MPLAB Code Configurator plugin for the MPLAB X IDE.
The Microchip Explorer 8 and Explorer16/32 Development Boards are used as the hardware development platforms.

The choice of platform or MCU target is not meant as a definite requirement; however, its adoption will lead to a faster time to the completion of the project, using tested and verified code that can fit most applications.

This application note is intended to be a reference for communicating with Microchip's I^2C serial EERAM devices using most of Microchip's PIC microcontrollers, in conjunction with MCC, without the need of extensive knowledge in software writing, peripheral programming or firmware in general. Almost everything related to the topic can be accomplished due to the capabilities of the MPLAB Code Configurator plugin.

MCC OVERVIEW

When starting a new project using PIC16, PIC24 or PIC32 microcontrollers, setup of the configuration and all the peripherals can be time consuming, especially for new projects. The MPLAB Code Configurator is a plugin for MPLAB X IDE that simplifies this down to a series of simple selections from the menus within the MCC. The MCC generates driver code using a Graphical User Interface (GUI). The generated drivers control the peripherals on PIC microcontrollers. The GUI provides an easy means for setting up the configuration of the peripherals.

Additionally, the MCC is used to configure and generate libraries, which allows the user to configure and generate code for software libraries and off-chip peripherals. The generated drivers or libraries can be used in any PIC device application program.

The MCC requires an MPLAB X IDE project to be created or an existing project to be opened before launching the MCC plugin. This is required as MCC needs to know the device used in the project to have access to device-specific information like: registers, bits and configurations and to set up the MCC GUI.

The MCC generates source and header files based on selections made in the GUI. The generated files are added to the active project of MPLAB X IDE.

Prerequisites and Installation of the MCC Plugin

Consult the "*MPLAB® Code Configurator User's Guide*" (DS40001725) for installation instructions, prerequisites and getting familiar with the GUI.

To automatically install the MPLAB Code Configurator Plugin, consider the steps below:

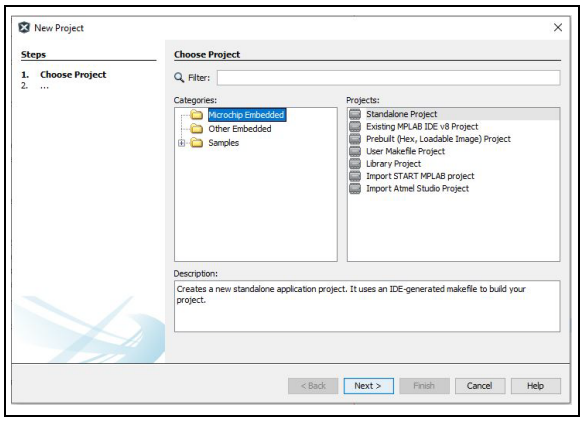
1. In the MPLAB X IDE, select Plugins from the Tools menu
2. Select the **Available Plugins** tab
3. Check the box for the MPLAB Code Configurator v3, and click on **Install**

To manually install the latest available plugin:

1. Access the address https://www.microchip.com/mplab/mplab-code-configurator and click on the **Current Download** tab, then download the .zip file that contains a .nbm file.
2. In the MPLAB X IDE, select Plugins from the Tools menu
3. Click on the **Downloaded** tab, then on **Add Plugins** and browse to a location where the .zip file has been extracted, then select the .nbm file
4. Click **Install**, which will lead you through the rest of the procedure

CONFIGURING YOUR PROJECT

Create a new MPLAB X IDE project or open an existing project. Steps are shown below for creating a project for the PIC16F1719 device.



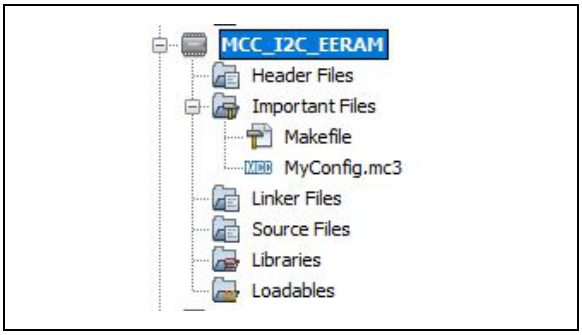
Create a normal stand-alone project.

Select the target device. The MPLAB Code Configurator will only load the supported libraries matching the target device. Internally, the MCC plugin has access to every resource that a particular target has, from pins, system peripherals such as clock and integrated peripherals like MSSP, I^2C, UART, etc. Refer to each target's data sheet to see what capabilities are enabled. In this particular example, the PIC16F1719 has an MSSP peripheral that can be configured to communicate on the I^2C bus specification.

Choose an available debugger, for example PICkit™, an ICD or run the project in Simulator mode.

Having the latest compiler installed ensures the most optimized output for your code if you are looking for size or speed.

Name the project and finish the process.

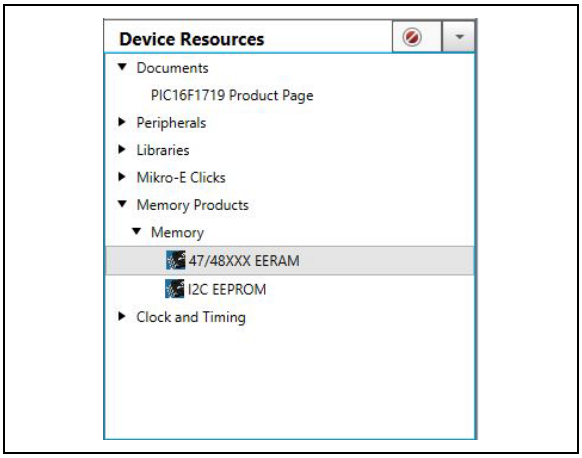


This creates an empty project that contains no files. Clicking the blue MCC icon in the toolbar will launch the MCC plugin within MPLAB X IDE.

SELECTING A PERIPHERAL OR RESOURCE

The MCC Graphical User Interface (GUI) is comprised of multiple panels, out of which the focus is brought to the Project Resources and Device Resources panels.

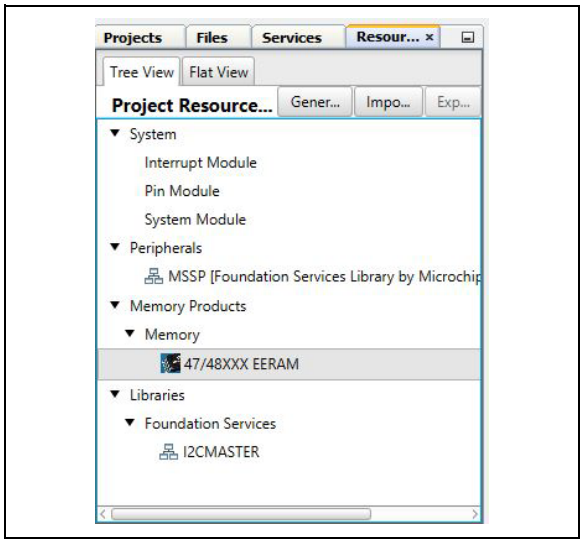
The Device Resources panel lists all the available resources for the selected target. Historically, this panel mainly contained the target microcontroller's peripherals, but has evolved to accommodate other useful resources. Users can take advantage of documentation, libraries for various stack implementations such as CAN and LIN, compatible external products like the Mikro-E line of Click boards™ and, as required for this application note, Memory Products with I^{2}C EEPROM.



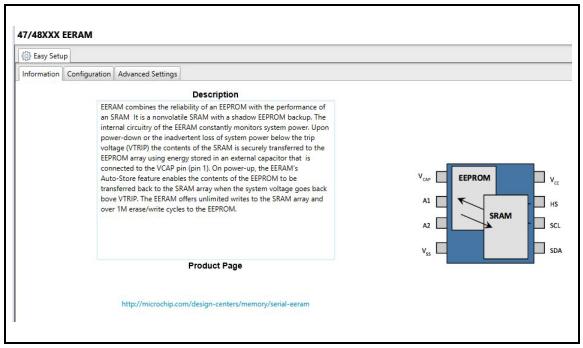
When selecting and double clicking a peripheral, like an ADC, for example, the selected resources will pop out of the Device Resources panel and into the Project Resources panel. This means that the respective peripheral has been selected and prepared for configuration and later code generation.

If selecting one of the newer resources, like I^{2}C EERAM in this case, the MCC plugin will automatically load the required compatible peripheral (in this case MSSP) into the Project Resources panel.

In other cases, for other types of libraries, manual selection of the peripheral to be used may be necessary.



The figure above shows that MSSP has been loaded alongside with the I^{2}C EERAM library.

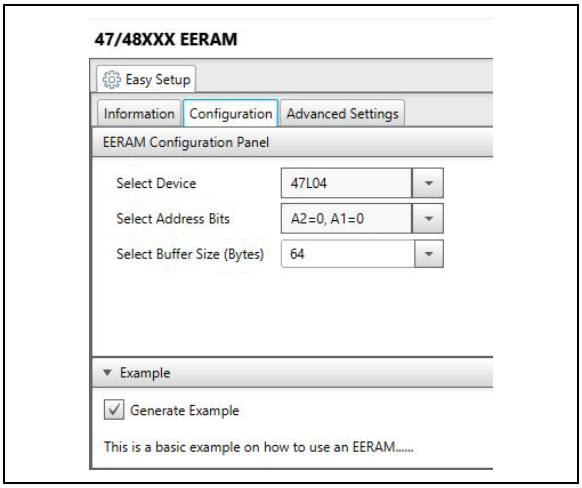


When a resource is selected from the Project Resources panel, the Graphical User Interface corresponding to that resource will be rendered within the main display area of the MCC window. In this case, after loading and selecting the I^{2}C EERAM the user has access to three tabs: **Information**, **Configuration** and **Advanced Settings**.

CHOOSING AND CONFIGURING AN EERAM

The **Configuration** tab contains four drop-down/editable controls:

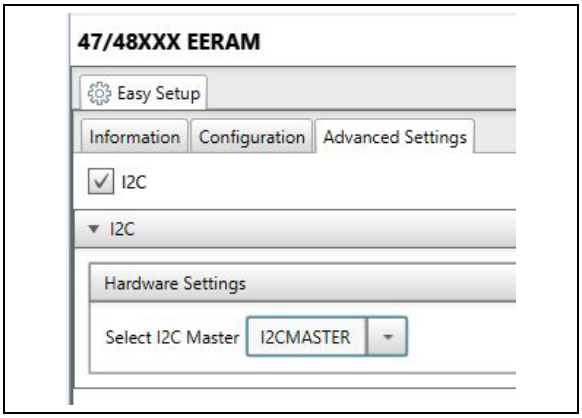
1. Select Device
2. Select Address Bits
3. Select Buffer Size (bytes)



The user can directly select an EERAM device from the drop-down list. Alternatively, a filter by density can be applied and the available device list will narrow accordingly.

The options for the Address bits drop-down list will adjust as well to the device selection and the available physical address pins on the device.

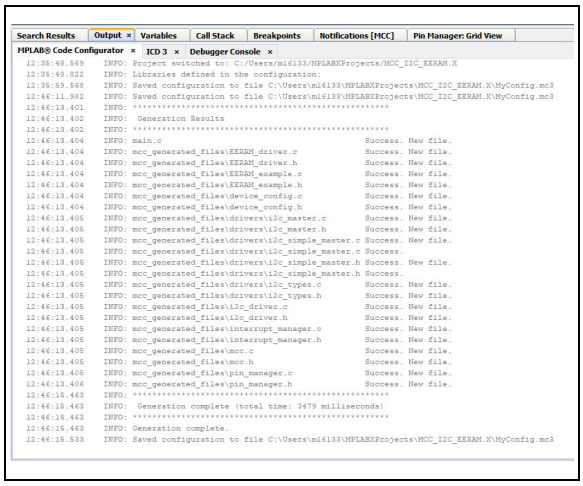
The GUI provides a check box for generating an example. This code example can be ported directly into the main file to quickly test the functionality of the device in real bench top conditions.



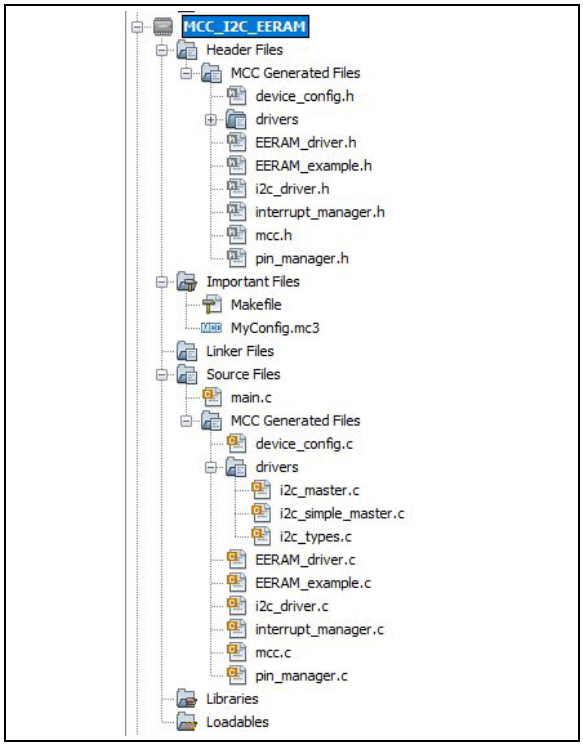
The **Advanced Settings** tab allows for simple configuration of the I^2C bus speed, and the changes applied here will manifest in the automatic altering of the MSSP registers to the correct values. This process is transparent to the user, which makes it far easier to use than manually configuring the registers.

CODE GENERATION

After all changes have been made and the user is satisfied with the selected configuration, by clicking the **Generate** button in the Project Resources panel, MCC will generate all the required files, both headers and .c files.



A report will be generated in the Output section, usually on the bottom of the interface. The user will be prompted to save the current MCC configuration into a .m3c file within the project folder, that can be shared between projects or machines for easier portability.



After all of the steps above have been taken, the user can close the MCC plugin by clicking on the blue MCC icon and can return to the classic project view in MPLAB X.

The project structure can be consulted, and one can observe the presence of several header and source files grouped under the “MCC Generated Files” nodes.

At this point the project can be successfully built.

MCC PROVIDED LAYERS

MCC provides a two-layer implementation to I^2C using MSSP. A third layer is visible to the user and is generated by the EERAM library that was loaded before. For the final user, a final API layer is exposed through the EERAM_driver.* files. In this stack-up, the implementation is comprised of:

- The bottom layer is the driver layer that consists of the i2c_driver.c file and dependences. This layer operates on the register level and manipulates the MSSP peripheral so that is behaves in accordance to the I^2C bus specification.
- The middle layer is the logical layer, consisting of the i2c_master.c source file and dependences. This is where a finite state machine operates on an Interrupt Service Routine (ISR).
- This ISR iterates through the state machine and calls the bottom layer whenever register-level operations are required (i.e., setting buffers, altering flags).
- The upper layer is the user-accessible layer that uses the middle layer for logical operations, directly calling exposed methods. This ensures consistency when propagating calls to the driver layer, as the code becomes very deterministic.
- The user is exposed to an API-like layer, that through the EERAM_driver.* files provides prototypes for single-function call operations.

USING THE GENERATED CODE

If the code generation was successful and the check box for generating an example was enabled, the example main function would have also been generated. Its purpose is to showcase the correct usage of a several generated functions.

Generated Example

```
/*
                         Main application
 */
void main(void)
{
    // initialize the device
    SYSTEM_Initialize();

    // When using interrupts, you need to set the Global and Peripheral Interrupt Enable bits
    // Use the following macros to:

    // Enable the Global Interrupts
    //INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    //INTERRUPT_PeripheralInterruptEnable();

    // Disable the Global Interrupts
    //INTERRUPT_GlobalInterruptDisable();

    // Disable the Peripheral Interrupts
    //INTERRUPT_PeripheralInterruptDisable();

    while (1)
    {
        // Add your application code
    }
}
/**
 End of File
*/
```

```
#include <stdio.h>
#include "mcc.h"
#include "EERAM_example.h"
#include "EERAM_driver.h"

uint8_t dataByte =0xFF;
uint8_t dataBuffer[8] = {0x55, 0xFF, 0xAA, 0xCC, 0x01, 0x02, 0x03, 0x04};


void main_example(void)
{

    EERAM_SetDeviceAddress (0x00);
    //foundation services code.

    EERAM_SetDeviceAddress        (0x00);

    EERAM_WriteOneByte            (0x10, dataByte);

    dataByte = EERAM_ReadOneByte          (0x0010);

    EERAM_SequentialWrite         (0x20, dataBuffer, 4);

    EERAM_RandomAddressRead       (0x22, dataBuffer+2, 2);

    EERAM_CurrentAddressRead      (&dataByte);

    dataByte = EERAM_StatusRegisterGet    ();

    EERAM_AutoStoreEnableBitSet   (0x01);

    EERAM_SoftwareStore           ();

    dataByte = EERAM_ArrayModifiedBitGet  ();

    EERAM_SequentialWrite         (0x20, dataBuffer+4, 4);

    EERAM_SoftwareRecall      ();

    EERAM_RandomAddressRead       (0x22, dataBuffer+2, 2);
}
```

The EERAM library generates an example main.c file that can be altered or imported based on user preferences. If called, it showcases both basic functionality as well as the special features of the EERAM range of products.

Some of the functions called in the example file are:

- Setting the device address
- Writing a byte
- Reading a byte
- Sequential write
- Sequential (or random address) read
  - Getting the STATUS register value
  - Setting a bit in the STATUS register value
  - Using the Software Store function
  - Getting a single bit from the STATUS register (the Array Modified bit)
  - Recalling the values from the EEPROM array within the EERAM

Documentation for these functions can be obtained by consulting the generated documentation or the function prototypes in the corresponding header.

FUNCTION USAGE

The example that was generated (shown in the previous section) contains the basic functions that an EERAM can be used with. Going further, here is how one can employ these functions and how they work.

Set Device Address

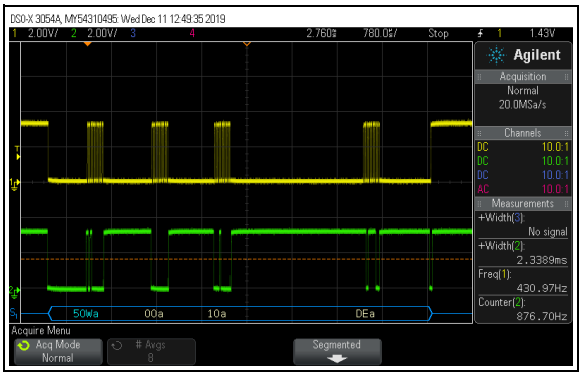
```
EERAM_SetDeviceAddress = (0x00);
```

Sets the device address in firmware to reflect the value corresponding to the Ax address pins of part.

Byte Write

```
EERAM_WriteOneByte(0x0010, 0xDE);
```

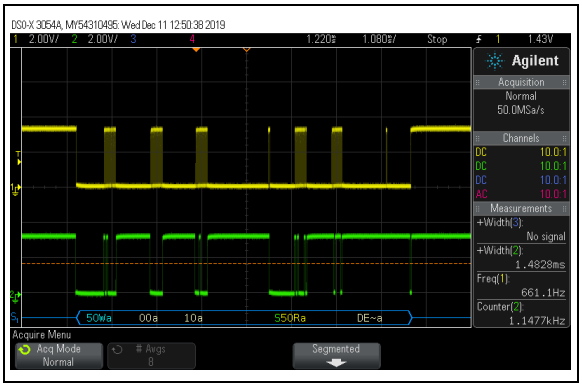
FIGURE 1: BYTE WRITE



Byte Read

```
dataByte = EERAM_ReadOneByte (0x0010);
```

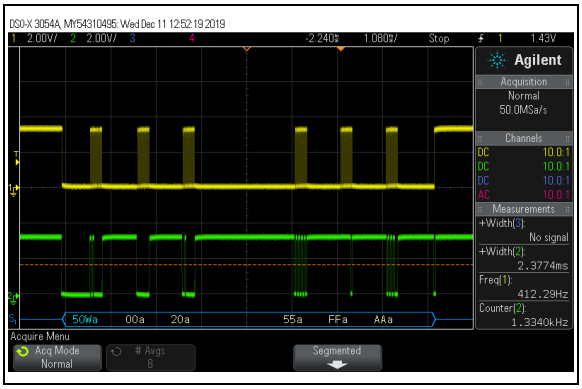
FIGURE 2: BYTE READ



Sequential Write

```
EERAM_SequentialWrite (0x0020, dataBuffer, 3);
```

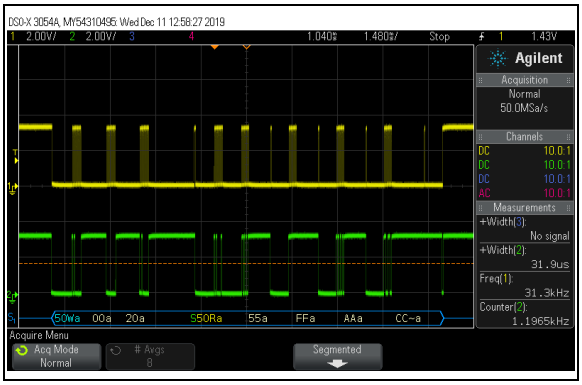
FIGURE 3: SEQUENTIAL WRITE



Random Address Read

```
EERAM_RandomAddressRead = (0x0022, dataBuffer+3, 4);
```

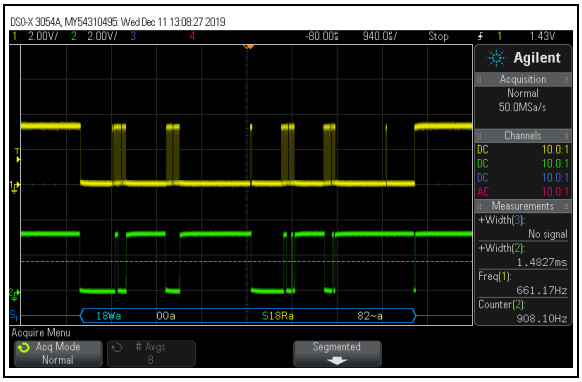
FIGURE 4: RANDOM ADDRESS READ



STATUS Register Get

```
SRegValue_EERAM_StatusRegisterGet();
```

FIGURE 5: STATUS REGISTER GET

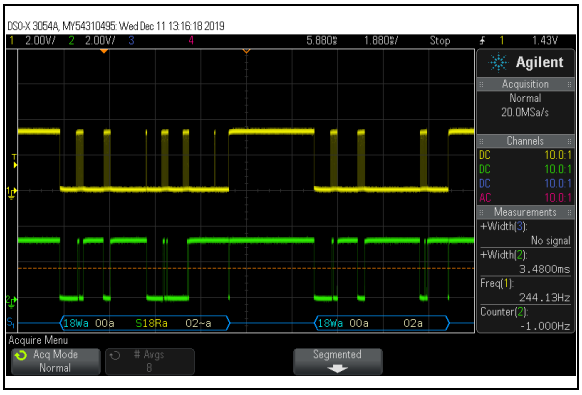


AutoStore Enable

```
EERAM_AutoStoreEnableBitSet = (0x01);
```

In order to set the AutoStoreEnable (ASE) bit, the STATUS register is read, its value is then masked with the ASE bit value, then written back to the part. In the capture shown in Figure 6, the ASE bit has already been set prior to the second attempt to set it, so no effective change is made and the capture only depicts the mechanism.

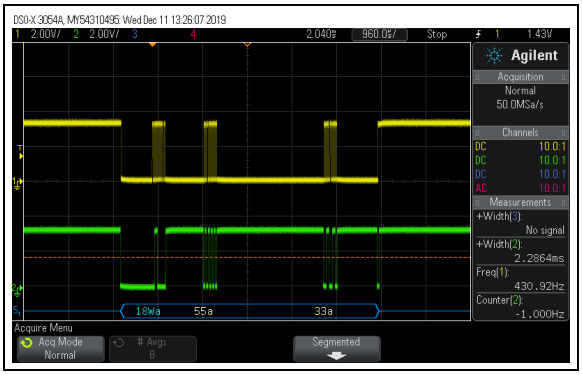
FIGURE 6: AUTOSTORE ENABLE



Software Store

```
EERAM_SoftwareStore = ();
```

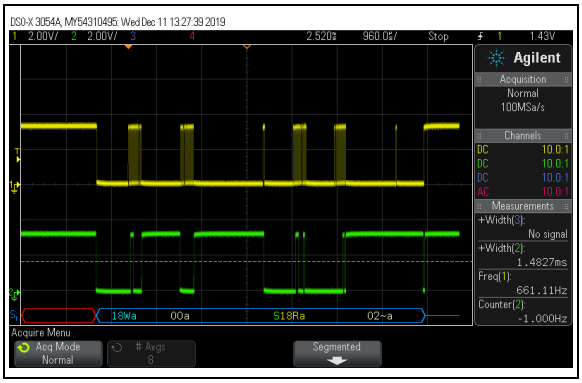
FIGURE 7: SOFTWARE STORE



Get AM Bit

```
AMBitValue = EERAM_ArrayModifiedBitGet = ();
```

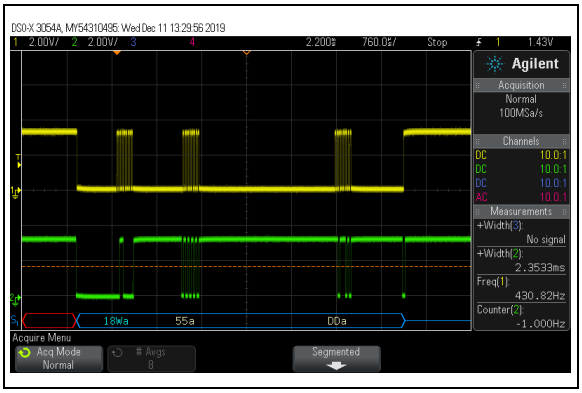
FIGURE 8: GET AM BIT VALUE



Software Recall

```
EERAM_SoftwareRecall();
```

FIGURE 9: SOFTWARE RECALL



COMPARING MCC WITH STANDARD MSSP PERIPHERAL HANDLING

Mid-range Microchip PIC microcontrollers, like the PIC16, might have a generic MSSP peripheral that can be used to implement various serial protocols including the I^2C specification, while PIC24 devices (and also PIC32 devices) have dedicated I^2C peripherals and they do not rely on a MSSP peripheral for serial communications.

USING MSSP AS I^2C ON A PIC16 WITHOUT MCC

Several of the mid-range enhanced core PIC16 microcontroller devices have a Master Synchronous Serial Port (MSSP). The MSSP module can be used to implement either the I^2C or the SPI communications protocol. The following is an overview of the registers involved in the configuration of MSSP to function as an I^2C peripheral and is meant to showcase the breadth and complexity of bit-level manipulation required for correct use. For more information, see AN735 – *"Using the PICmicro MSSP Module for I^2C Communications" (DS00735)*.

MSSP Registers and Functionality

Some key Special Function Registers (SFRs) utilized by the MSSP module are:

- SSP Control Register 1 (SSPCON1)
- SSP Control Register 2 (SSPCON2)
- SSP STATUS Register (SSPSTAT)
- Pin Direction Control Register (TRISC)
- Serial Receive/Transmit Buffer (SSPBUF)
- SSP Shift Register (SSPSR) - Not directly accessible
- SSP Address Register (SSPADD)
- SSP Hardware Event Status (PIR1)
- SSP Interrupt Enable (PIE1)
- SSP Bus Collision Status (PIR2)
- SSP Bus Collision Interrupt Enable (PIE2)

To configure the MSSP module for Master I^2C mode, key SFR registers must be initialized in order to configure the MSSP module for Master I^2C mode.

- SSP Control Register 1 (SSPCON1)
  - I^2C mode configuration
- SSP Address Register (SSPADD)
  - I^2C bit rate
- SSP STATUS Register (SSPSTAT)
  - Slew rate control
  - Input pin threshold levels
- Pin Direction Control (TRISC)
  - SCL/SDA direction

Once the basic functionality of the MSSP module is configured for Master I^2C mode, the remaining steps relate to the implementation and control of I^2C events. The master can initiate any of the following I^2C bus events:

- Start
- Restart
- Stop
- Read (Receive)
- Acknowledge (after a read)
  - Acknowledge
  - Not Acknowledge (NACK)
- Write

The first four events are initiated by asserting high the appropriate control bit in the SSPCON2 register. The Acknowledge bit event consists of first setting the Acknowledge state, ACKDT (SSPCON2) and then asserting high the event control bit, ACKEN (SSPCON2).

Data transfer with Acknowledge is obligatory. The Acknowledge-related clock is generated by the master. The transmitter releases the SDA line (HIGH) during the Acknowledge clock pulse. The receiver must pull down the SDA line during the Acknowledge clock pulse so that it remains stable LOW during the HIGH period of this clock pulse.

When the Slave does not acknowledge the master during this Acknowledge clock pulse (for any reason), the data line must be left HIGH by the slave. This sequence is termed "Not Acknowledge" or "NACK".

For actual data to be transferred, the SSPBUF register must be written with the control byte and the data to be sent. Once the SSPBUF is loaded with data, the MSSP peripheral will clock out the data at the configured rate.

Pin Assignment

Another aspect involved in the functioning of MSSP as I^2C (or any type of supported bus) is the correct pin assignment. After the MSSP configuration has been set, the data sheet must be consulted in order to determine the correct I/O pins that map to the MSSP peripheral.

Depending on the peripheral configuration of the PIC device in use (i.e., number of MSSP peripherals), the most common ports used are RC3 and RC4. These need to be configured as well by setting the correct data direction according to the role the MSSP plays in the I^2C implementation (master or slave).

Byte Write Routine as Master

```
void wait()
{
while ((SSPSTAT & 0x04) || (SSPCON2 & 0x1F));
return                              ;
}

void main(void)
{
SSPCON    = 0X28                    ;// I2C Master, enable SCL and SDA
SSPCON2   = 0                       ;//
SSPADD    = 99                      ;// calculated using FOSC/(4*speed)-1 where speed is 10000
SSPSTAT   = 0                       ;
TRISC     = 0x18                    ;// using RC3 (SCL) and RC4 (SDA)
char data = 0                       ;

while(1)
{
wait()                              ;
SEN       = 1                       ;// I2C start
wait()                              ;
SSPBUF    = 0x30                    ;// control byte (7-bit address + r/nw bit)
wait()                              ;
SSPBUF    = 0x30                    ;// send and increment data char
wait()                              ;
PEN       = 1                       ;// I2C stop
_delay_ms(100)                      ;
}
}
```

The main takeaway of the example with PIC16 is that even the simplest bit level implementation of the peripheral requires a certain expertise with using the actual MSSP peripheral. Certain bits and flags must be precisely manipulated for correct operation, which can be time consuming.

The advantage of this approach is that developing low-level code leads to a better understanding of the device in use and makes debugging easier and more deterministic. The disadvantage is the amount of time spent in order to get a coherent and working example.

CONCLUSION

When deciding between a bare-bones versus a complete implementation, MCC helps with providing the latter. By using MCC instantiated code, a predictable and deterministic implementation can be achieved.
Microchip Technology is continuously adding supported device libraries and is improving the capabilities of MCC to serve both prototyping and industrial-grade code generation. Its ease of use is matched with consistent code, that can be deployed in real-life applications.

The generated code is documented and visible, and other implementations can be easily derived from it to fit every need. For a robust implementation in embedded systems and peripherals, the MPLAB Code Configurator is a great starting point.

REFERENCES

1. AN 734 – *"Using the Mid-Range Enhanced Core PIC16 Devices' MSSP Module for Slave I^2C Communication"* (DS00000734)
2. AN735 – *"Using the PICmicro MSSP Module for I^2C Communications"* (DS00735)
3. AN 2045 – *"Interfacing Serial EEPROMs with 8-Bit PIC® Microcontrollers"* (DS00002045)
4. AN1028 – *"Recommended Usage of Microchip I^2C Serial EEPROM Devices"* (DS01028)
5. *"1024K I^2C Serial EEPROM"* Data Sheet (DS20001941)
6. "*Inter-Integrated Circuit (I^2C)*" Reference Manual (DS70000195)

APPENDIX A: REVISION HISTORY

Revision A (April 2020)

Initial release of this document.

APPENDIX B: I²C OVERVIEW

An I²C bus can have one or more master devices and one or more slave devices. The master device is the device that initiates a data transfer on the bus and is responsible for generating the serial clock used on the bus. Any addressed device is then considered a slave. Data transfers are performed eight bits at a time, starting with the Most Significant bit (MSb). Each device is recognized by a unique address and can operate as either a transmitter or receiver.

The physical interface of the bus consists of two bidirectional open-drain lines, one line used for the serial clock (SCL) and the other used for serial data (SDA). Each line will require a pull-up resistor to supply voltage to the lines. Pulling the line to ground is considered a logical Low, while letting the line float high is considered a logical High. When the bus is free, both SDA and SCL are logical High.

Data can be transferred at a rate up to 100 Kbits/s in the Standard mode, up to 400 Kbits/s in Fast mode, up to 1 Mbit/s in Fast mode Plus or up to 3.4 Mbits/s in High-Speed mode. Data on the SDA line must be stable during the high period of the clock. Any changes on the SDA line can only occur when the clock signal on the SCL line is low. One clock pulse is generated for each transferred data bit.

The I²C specification defines a Start condition as a transition of the SDA line from a high-to-low state, while the SCL line is high. A Start condition is always generated by the master and signifies the transition of the bus from an Idle to an Active state. The I²C specification states that no bus collision can occur on a Start.

A Stop condition is a transition of the SDA line from a low-to-high state while the SCL line is high. At least one SCL low time must appear before a Stop is valid. A Restart, or Repeated Start, is valid any time that a Stop would be valid. A master can issue a Restart if it wishes to hold the bus after terminating the current transfer. A Restart has the same effect on the slave that a Start would, resetting all slave logic and preparing it to clock in an address. The master may want to address the same or another slave. This can be useful for many I²C peripherals, such as nonvolatile EEPROM memory, in which an I²C write operation and a read operation are done in succession. In this case, the write operation specifies the address to be read and the read operation gets the byte of data. Since the master device does not release the bus after the memory address is written to the device, a Restart sequence is performed to read the contents of the memory address.

The ninth SCL pulse for any transferred byte in I²C is dedicated as an Acknowledge (ACK). It allows receiving devices to respond back to the transmitter by pulling the SDA line low. The transmitter must release control of the line during this time to shift in the response. The ACK is an active-low signal, pulling the SDA line low, indicating to the transmitter that the device has received the transmitted data and is ready to receive more.

FIGURE 10: EXAMPLE DATA TRANSFER SEQUENCE ON I²C

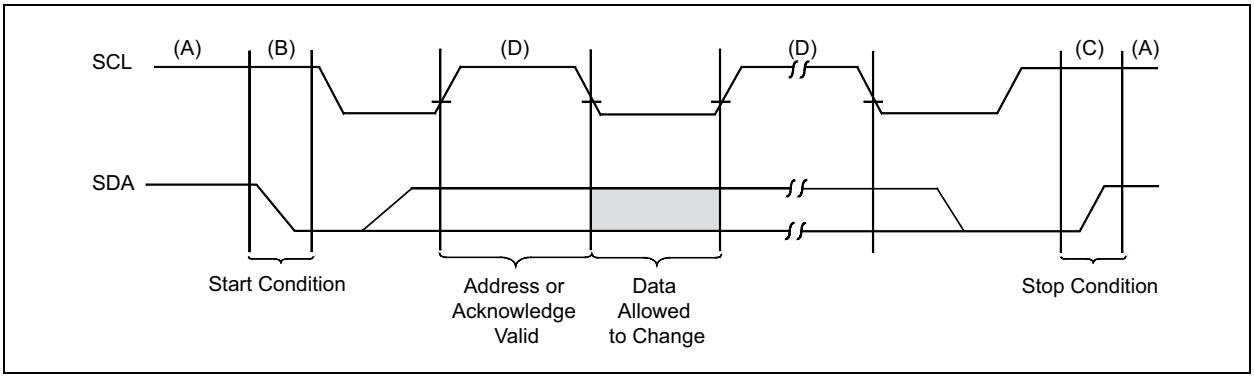
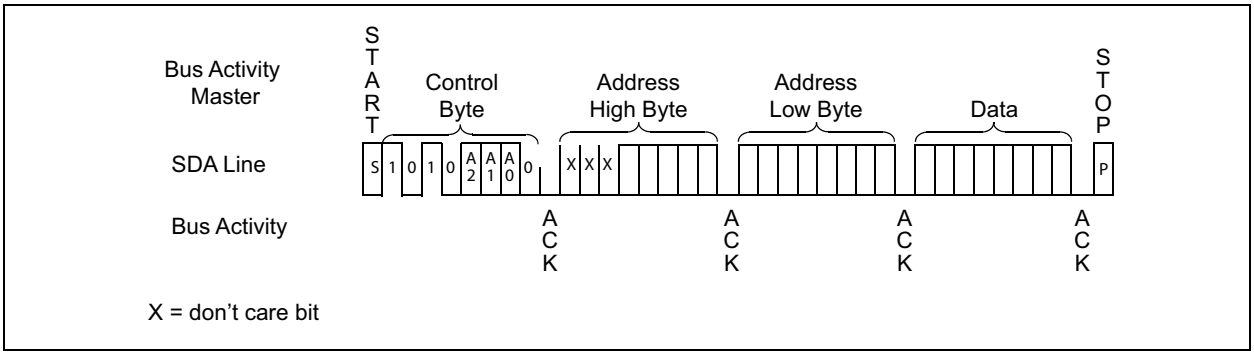


FIGURE 11: EXAMPLE BYTE WRITE ON A 64-KBIT I²C DEVICE



APPENDIX C: EERAM CHARACTERISTICS

Chip Address Inputs

The Chip Address input pins (A0, A1 and A2 or combinations) are used on several devices to support multiple device operation. On devices with this feature, the levels on these inputs are compared with the corresponding bits in the slave address, and the device is selected if the comparison is true. Note that the Chip Address pins are not internally connected on some devices. Also note that some devices like the 47X04, 47X16 and 47L64 do not have an A0 pin, but instead have A1 and A2 pins. Refer to the appropriate device data sheet for more details. For devices with internally connected Chip Address pins, these inputs must be hard-wired to either logic '0' or logic '1'. That is, they cannot be left floating, otherwise the device will not operate correctly.

Power Supply

Microchip serial EERAMs feature a high amount of protection from unintentional writes and data corruption while power is within normal operating levels. But certain considerations should be made regarding power-up and power-down conditions to ensure the same level of protection during those times when power is not within normal operating levels. A decoupling capacitor (typically 0.1 µF) should be used to help filter out small ripples on V_{CC}. Consult the most up-to-date data sheet and specification/recommended usage when deploying Microchip EERAMs in sensitive applications.

Bus Pull-Up Resistors

For proper operation, pull-up resistors are required for both SCL and SDA buses. However, the resistor value chosen can have a vast impact on the performance of the system.

Specifically, three limiting actors must be considered when selecting pull-up resistor (RP)

- Supply Voltage (V_{CC})
- Total Bus Capacitance (C_{BUS})
- Total High-Level Input Current (I_{IH})

For an in-depth computation of these factors, consult AN1028 – *"Recommended Usage of Microchip I^2C Serial EEPROM Devices"* (DS01028). Most applications will require the pull-up resistor value to be 2.2 kOhm but this may vary depending on requirements.

External Capacitor

One of the key features of the 47XXX devices is the AutoStore mechanism. To enable this feature, the user sets the ASE (AutoStore Enable) bit in the STATUS register to '1' and installs a capacitor connected between the V_{CAP} pin and ground. However, if the user decides that only manual store operations are required, the ASE bit must be set to '0' and the V_{CAP} pin must be connected to V_{CC}.

See AN2257 for details on choosing the right external capacitor for your application and also consult the data sheet and Microchip application notes for correct usage.

APPENDIX D: UNDERLYING CODE ANALYSIS

This annex helps the reader understand both the functionality of the generated code and the complexity of a robust I^2C implementation.

The entry point for analyzing the code will be the Byte Read function.

```
dataByte = EERAM_ReadOneByte   ;   (0x0010);
```

The function implementation is:

```
uint8_t data                                ;
while (!I2C_open(EERAM_DEVICE_ADDRESS))   ;   //sit here until we get the bus...
I2C_setDataCompleteCallback(readOneByteHandler & data) ;
address = (address&0xFF00)>>8|(address&0x00FF)>>8      ;
I2C_setBuffer(& address,2)                             ;
I2C_setAddressNACKCallback(I2C_restart_write,NULL)     ;  //NACK polling
I2C_masterWrite()                                      ;
while (I2C_BUSY==I2C_close())             ;   //sit here until finished
return data                               ;
}
```

The first important call is:

```
I2C_open(EERAM_DEVICE_ADDRESS)
```

This function operates on a structure that maps a number of flags and data pointers specific to the I^2C implementation on the MSSP peripheral.

```
typedef struct
{
unsigned busy:1                         ;
unsigned inUse:1                        ;
unsigned bufferFree:1                   ;
unsigned addressNACKCheck:1             ;
I2C_address_t address                   ;          ///  The I2C Address
uint8_t *data_ptr                       ;          ///  pointer to a data buffer
size_t data_length                      ;          ///  Bytes in the data buffer
uint16_t time_out                       ;          ///  I2C Timeout Counter between I2C events
uint16_t time_out_value                 ;          ///  Reload value for the timeouts
I2C_fsm_states_t state                  ;          ///  Driver State
I2C_error_t error
I2C_callback callbackTable[6]           ;
void *callbackPayload[6]                ;          ///   each callback can have a payload
} I2C_status_t                          ;
```

This type definition resides in the I2C_master.c file which is generated by MCC.

The I2C_open() function resets the flags and sets the device address within this structure. It also calls I2C_driver_open(). This is also the point where the code eventually reaches the register level:

```
_bit I2C_driver_open(void)
if(!SSP1CON1bits.SSPEN)
{
SSP1STAT            = 0x00          ;
SSP1CON1            = 0x28          ;
SSP1CON2            = 0x00          ;
SSP1ADD             = 0x3           ;
return true                         ;
}
else
return false                        ;
}
```

Going upwards to the user-available API, the EERAM_ReadOneByte() function, after the call to I2C_open() one can observe the assignment of a data complete callback by sending a function pointer as a parameter:

```
I2C_setDataCompleteCallback(readOneByteHandler & data);
```

The readOneByteHandler function is already implemented and basically maps a pointer to the I^2C specific (I2C_status) structure's data pointer.

After the callback has been registered and thus the pointer remapped, the EERAM_ReadOneByte() function calls:

```
I2C_setBuffer(&address,2)  ;
```

In case the slave device (EERAM) does not acknowledge, a restart write is callback is registered through:

```
I2C_setAddressNACKCallback(I2C_restart_write,NULL); //NACK polling
```

The next call is I2C_masterWrite(); which in term alters the I2C_status structure by setting a write flag. Based on these flags, the code eventually iterates through a finite state machine that correctly operates the MSSP peripheral.

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.

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Trademarks

ISBN: 978-1-5224-5987-3

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://www.microchip.com/support
Web Address:
www.microchip.com

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

Austin, TX
Tel: 512-257-3370

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
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

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

China - Beijing
Tel: 86-10-8569-7000

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

China - Chongqing
Tel: 86-23-8980-9588

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115

China - Hong Kong SAR
Tel: 852-2943-5100

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

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

China - Shanghai
Tel: 86-21-3326-8000

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

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

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

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

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444

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

India - Pune
Tel: 91-20-4121-0141

Japan - Osaka
Tel: 81-6-6152-7160

Japan - Tokyo
Tel: 81-3-6880- 3770

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

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

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

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

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600

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

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

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

Denmark - Copenhagen
Tel: 45-4485-5910
Fax: 45-4485-2829

Finland - Espoo
Tel: 358-9-4520-820

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

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-72400

Germany - Karlsruhe
Tel: 49-721-625370

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

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra'anana
Tel: 972-9-744-7705

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

Italy - Padova
Tel: 39-049-7625286

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

Norway - Trondheim
Tel: 47-7288-4388

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

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

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820