MPLAB® XC Compiler Functional Safety Manual

INTRODUCTION

This document is a functional safety manual for the MPLAB XC family of compilers.

An MPLAB XC compiler is available for each family of Microchip PIC® microcontrollers:

- MPLAB XC8 C Compiler for 8-bit microcontrollers
- MPLAB XC16 C Compiler for 16-bit microcontrollers
- MPLAB XC32 C Compiler for 32-bit microcontrollers

MPLAB XC compilers are modern, carefully developed, and thoroughly tested tools. They are compatible for use in functional safety applications requiring ISO 26262 up to ASIL D.

The compilers have features that, if considered during planning and design, could impact functional safety. These considerations are presented throughout this functional safety manual.

Disclaimer: It is the responsibility of system or application designers to ensure that systems and applications meet all applicable safety, regulatory, and system-level performance requirements.

All application and safety-related information in this document (including application descriptions, suggested safety measures, suggested Microchip products, and other materials) are provided for reference only.

You understand and agree that your use of Microchip components in functional safety or safety-critical applications is entirely at your risk, and that you (as buyer) agree to defend, indemnify, and hold harmless Microchip Technology Incorporated from any and all damages, claims, suits, or expenses resulting from such use.

PRODUCT OVERVIEW

The MPLAB XC compiler line consists of full-featured, ANSI-C89-compliant language tool suites that include an optimizing C language compiler, assembler, linker, and librarian. The tool suites include many source and precompiled libraries, including standard C, math, and custom device-specific libraries provided by Microchip.

Use Case

A compiler has a simple use case: turn source code into an executable image, based on instructions to the tool suite. The apparent simplicity of the use case should not mask the complexity of this process and the sensitivity of the compiler output to the source code and instructions.

Compiler Features

This manual examines the use of compiler features in regard to their impact on functional safety.

Many features of the MPLAB XC compilers are architecture-specific, e.g., extended data space pointer types or constructs to set device Configuration bits.

Some of the other features of the compilers are generic qualities that are associated with code generation, e.g., the size of generated code, speed optimizations, or how stack-based objects are accessed.

Targeted Applications

This manual provides guidelines for developing embedded, real-time, and functional safety applications using the MPLAB XC language tool suite. The guidelines also apply to developing applications that must be developed in accordance with functional safety standards. In fact, these guidelines should be used for *all* projects.

Product Safety

Microchip MPLAB XC compilers are qualified under the ISO 26262 functional safety standard; so, developers can use these tools with confidence. The tools have been qualified based on the tool classifications indicated in the next section.

QUALIFICATION FOR AUTOMOTIVE SAFETY INTEGRITY LEVEL (ASIL) DEVELOPMENT

According to the tool requirements in the ISO 26262 functional safety standard, the MPLAB XC compilers have a Tool Impact level of TI 2. This level indicates a *possibility* of the tool to introduce errors or fail to detect errors in the application being developed. There is a medium degree of confidence in the error prevention and detection measures, so these tools have a Tool Error Detection level of TD 2. Based on these two assessments, the MPLAB XC compilers can be classified as Tool Confidence Level TCL 2.

For this confidence level and for applications classified as ASIL D, at least one of the tool qualification methods listed below must be applied:

1. Increased confidence from use
2. Evaluation of the tool development process
3. Validation of the software tool
4. Development in accordance with a safety standard

Microchip has applied its world-class software development process and industry-leading compiler validation methodology (tool qualification methods 2 and 3) to meet the tools' classification requirements as part of the functional safety standard. These are described in the next two section sections, respectively.

TOOL DEVELOPMENT PROCESS

Central to each of the Microchip safety solutions is a focus on quality. From design to manufacturing, Microchip employs ISO TS 16949 and ISO 9001:2008 certified Quality Management Systems, as well as a dedication to zero defects methodologies. These systems help to ensure that our products meet the stringent demands of safety applications and standards in the automotive and industrial markets. Microchip also focuses on continuous improvement, with process evaluation, assessments, audits, and gap analyses to ensure processes are continually optimized.

VALIDATION OF SOFTWARE TOOLS

MPLAB XC compilers are subject to rigorous testing before being released. Testing is carried out using the Microchip regression suite, which is comprehensive and frequently updated to support new compiler features and new devices, and to address any reported issues. The regression suite comprises over 600,000 tests from the industry-standard test suites, as well as in-house tests that are specifically tailored to the Microchip device architecture. The suite is designed to ensure correct code operation on Microchip PIC devices and ANSI C compliance. The regression suite is run using various optimization levels; multiple command-line options; all compiler operating modes; and on a number of platforms, including 32- and 64-bit Windows® (excluding Windows Server), Linux® and Mac OS® X platforms. Users should consult the compiler's release notes to confirm the exact platforms on which that version of the compiler was verified.

INSTALLATION

Every modern language tool comes packaged in an installer, which automates the installation process and decreases the chance of incorrect tool operation. Microchip provides a secure, proprietary methodology for installing the compiler and guaranteeing that the compiler is installed *properly*. In addition to verifying the presence and location of every compiler file and the permissions associated with each of them, the installation process provides an executable that checks the integrity of each of these files. Secure Hash Algorithm 1 (SHA-1) checksums are used to verify every application in every directory. Finally, the installer initiates compilation of a small project to confirm that the installation was successful.

DEVELOPMENT INTERFACE AGREEMENTS

A Development Interface Agreement (DIA) is an agreement between a customer and supplier that outlines the management of shared responsibilities in developing a functional safety system. In custom developments, the DIA is a key document executed between the customer and supplier early in the development process. As the MPLAB XC Compiler family is a commercial, off-the-shelf (COTS) product, Microchip has prepared a standard DIA that describes the support Microchip can provide for customer developments. Requests for custom DIAs should be referred to your local Microchip sales office.

SUGGESTED SAFETY MEASURES

Many features of the tool suite, source language, and defensive programming techniques can be employed to enhance the intrinsic efficacy of a functional safety software system. The following sections outline some of the recommendations of the Language Tool development team of Microchip Technology Incorporated. The Safety Integrity Level (SIL) of your application dictates the extent to which you must adhere to these recommendations. It is up to you, the system or application designer, to determine which of these recommendations to use in your design process.

Assertions

Assertions are statements constructed by the programmer that confirm an assumption. If the assumption is not correct at runtime, an error is flagged or the program is aborted. Assertions are a very effective way of finding runtime errors. The MPLAB XC compiler tool suites use the standard assertion macro that is defined in `<assert.h>`.

Beneficial and highly recommended applications of assertions include:

- Function argument range checking (preconditions)
- Function return value checking (postconditions)
- Switch expression range checking
- Index out-of-bounds checking
- Program flow control checking

By taking advantage of conditional compilation, properly constructed assertions leave no footprint in a production image. Programmers can use assertions liberally without having an impact on runtime performance or code size. Using assertions during development and testing can rapidly find oversights and omissions in the programmer's design and understanding. Microchip MPLAB X IDE and hardware debuggers allow an in-place software breakpoint to be used by an assertion to halt program execution without wasting valuable debugging resources.

ANSI Standard and Implementation-defined Behaviors

The programming language specified by the ANSI C standard is not without peculiarities. Examples of code behaviors that often catch unwary developers include the operators' order of precedence, integer promotion, and implicit type coercions.

The Microchip Language Tool development team strongly urges that developers of functional safety, real-time, embedded software systems thoroughly understand the *entire* language specification. Source code that does not conform to the standard, even seemingly innocent language facilities, might cause unexpected and unpredictable results that can adversely affect the safety of a system.

Many aspects of the ANSI C standard relate to behavior that is not fully defined by the standard, and each tool suite implementation may interpret these situations in a way that is best suited to the target system. Inadequate understanding of this "implementation-defined behavior" is often the cause of unpredictable and unsafe application operation.

The implementation-defined behavior of each compiler is described in a dedicated appendix in its users' guide. It is *imperative* that each developer understands when source code has implementation-defined behavior and ensures that they do not make assumptions about the results of such code.

Only Use Documented Options

The MPLAB XC16 C Compiler and MPLAB XC32 C Compiler are based on the GNU Compiler Collection (GCC) open-source code base.

Because of this genealogy, there are GCC compiler switches that remain available in the tool even though they are not particularly pertinent to embedded code generation. The Microchip Language Tool development team strongly urges developers of functional safety, real-time, embedded software systems to refrain from using options that are not documented in the relevant XC compiler's users' guide.

The presence of an option description in generic GCC documentation does not imply that the option is supported by the MPLAB XC16 C Compiler or the MPLAB XC32 C Compiler.

Motor Industry Software Reliability Association (MISRA)

The use of a MISRA rule checker is mandatory in some functional safety industries, and the Microchip Language Tool development team strongly urges all embedded programmers to use a MISRA rule checker on all functional safety projects. Programs that adhere to the MISRA rules in addition to the ANSI standard are more deterministic and less likely to fail. For example, code pruning (discussed in **"Optimization"**) is specifically allowed by the ANSI C standard, but is shown to cause difficulty under different compilation environments.

It is also important to consider that the effectiveness of MISRA checkers will vary from product to product. Exercise due diligence when selecting a MISRA checker. Ensure it is full-featured, proven, and thorough.

RULE SETS

It is incumbent upon the developers of functional safety systems to understand which MISRA rules must be enabled and which may be ignored. In general, enabling more rules increases the safety benefits. Consequently, the Microchip Language Tool development team recommends that all MISRA rules be used.

Turn On All Compiler Messages

Microchip strongly recommends that all compiler warnings are cleared before testing or utilizing any code. A compiler warning indicates that there is a high probability that there is a syntactic or semantic problem with the code identified.

When using MPLAB XC16 and MPLAB XC32, use the following options to enable all warnings:

`-Wall` and `-Wextra`

When using MPLAB XC8, use the following option:

`--WARN=-9`

Note that using compiler-specific settings to disable warning reporting is strongly discouraged.

Microchip also encourages the promotion of warnings to errors. In this way, warnings *must* be dealt with before you can continue development.

When using MPLAB XC16 and XC32, use the following option to promote warnings:

`-Werror`

When using MPLAB XC8 this must be specified with `#pragma warning error`.

Note that the use of compiler-specific settings to disable error reporting is strongly discouraged.

Recursion

The use of recursion is always a design decision. However, no one in the Microchip Language Tool development team has encountered a situation that has actually *required* recursion. The team agrees with the MISRA specification and recommends against the use of recursion in all embedded software designs.

Reentrancy

While every MPLAB XC compiler has the ability to encode a function so that it is reentrant, the programmer must ensure that the source code they place in functions does not explicitly break reentrancy conditions. A function that can be called by an interrupt service routine and mainline code, for example, must be reentrant.

A routine must satisfy the following conditions to be reentrant:

A routine must never modify itself.

That is, under no circumstance should the instructions of the routine be changed at run-time.

Non-reentrant routines must not be called.

This restriction applies to all functions in the call graph of the reentrant function.

Only instances of variables must be modified.

Any variables changed by the routine must be allocated to a particular instance of the function's data memory.

The last point implies that the compiler must allocate auto, parameter, and temporary variables to unique positions in memory – typically on a stack. It also indicates that the programmer must design these routines so they *use* auto variables to store data. Unless the routine uses a way of providing mutual exclusion, global or static variables should never be modified by these routines.

Unused Program Memory and Interrupt Vectors

There are many different strategies for handling unused program memory. The effectiveness of these strategies will depend on the application. Functional strategies include filling unused program memory with a specific instruction, such as the following:

Reset instructions

Reset code may need to detect the fault and allow a gracefully recovery.

Unconditional branches

Jump to a disaster-recovery function that cleans up and may force a Reset.

Halt instructions

Effectively prevent further damage, but do not readily allow for error recovery.

Choosing an appropriate strategy is the responsibility of the development organization, but the Microchip Language Tool development team considers the filling of memory locations that are unused by the application to be essential for safe and predictable application behavior.

Microchip also considers it essential to define interrupt handlers for unused interrupts. There is a wide range of effective strategies for providing default interrupt handlers. Note that some Microchip PIC devices use a table of interrupt vector addresses, and others use fixed locations for interrupt service routines. In both cases, it is recommended to write an interrupt handler to ensure that unexpected interrupts do not cause undue damage to the application's operation.

Cyclomatic Complexity

Short, simple functions are safer functions. The Microchip Language Tool development team strongly recommends using the cyclomatic complexity metric and refactoring any function with a cyclomatic complexity greater than ten (10).

In addition to the direct benefits of easier understandability and maintainability, limiting the cyclomatic complexity of code in a functional safety application restricts the compiler to the most frequently used and extensively tested statement sequences. This substantially reduces the chance of code failure.

Language/Architecture Features

Non-standard compiler extensions that provide access to architecture-specific features allow your application to take full advantage of your chosen device, but these same unconventional features can quickly become liabilities if they are not properly understood.

While no processor architecture feature or compiler extension is *inherently* unsafe, most require knowledge of the context in which they operate, both in terms of the tool suite and the device. The Microchip Language Tool development team strongly urges you to research the purpose and actions of the features or extensions, and be aware of any side effects, before you use them. Used improperly, any feature or extension can cause unpredictable and unsafe results.

Optimization

Like non-standard extensions, no optimization is inherently unsafe, but many engineers have misconceptions about how and when optimizations can change the operation of their code.

C source code is a description of how the final binary image is to run at a *functional level*, i.e., what you, as an external observer, can ‘see’ the program do. The language has no means of conveying the time in which this has to happen or how it is to happen internally to the device. Optimizers work within the functional framework of your code. They can change the internal actions performed by your code, as long as they do not change the external behavior.

The following sections address several misconceptions about optimization and, in most cases, present the appropriate responses on the part of the programmer.

ALGORITHM CHOICE

Usually, the choice of a good algorithm is the biggest improvement that can be made to the performance of an application. Your time is better spent researching better algorithms than trying to optimize one that is inefficient.

Optimizers make only small improvements to performance compared to the improvements possible from looking at the requirements of your application in a different way. Optimizers typically deal with localized parts of your code and can rarely see the overall purpose of your code. They cannot restructure your code; they can only make each small fragment perform at its optimum.

HAND-WRITTEN ASSEMBLY

Many engineers think that they must use hand-written assembly code to avoid real-time performance constraints. While this is sometimes true, software engineers are rarely granted the time to develop, debug and maintain assembly code.

The old adage “first make it work, then make it fast,” is very relevant in this situation. Ensure you do not waste resources optimizing code before its validity has been ascertained.

Initially implementing an algorithm in C may quickly reveal problems in your design. In addition, the time saved by writing C code could be put to use in finding a better algorithm (which has a performance that may negate the need for assembly code entirely). Only consider hand-writing code if C code cannot be made to run at the required speed.

The XC compilers may provide “built-in” functions for common situations that might otherwise require hand-written assembly. Use these functions, as they will be more reliable than assembly code written by hand.

If you must program in assembly code, write separate assembly routines rather than using assembly inline with C code. If you absolutely must use inline assembly code, ensure that you understand and use the “extended” instruction form for MPLAB XC16 and XC32, so you can make your intent clear to the compiler. Wherever possible, use variables defined in C code, not the underlying registers. Inline assembly code is primarily intended to allow the specification of small assembly sequences that cannot otherwise be guaranteed with C code.

CODE WITH NO EFFECT

Optimizers readily remove code that has no effect. Understanding this action requires understanding when code *has* an effect. Recall that the C language is only concerned with the external behavior of a program. Code has no effect in the following situations:

When it cannot be reached.

There are no control flow paths that can lead to the statements.

When it is redundant.

It performs the same functional action as earlier code, and there are no intervening entry points for jumps or loops.

When it doesn’t change anything that is externally observable.

Unless the compiler is told otherwise, memory locations are *not* considered as externally observable.

The last case is a sticking point with many programmers. Consider the following example of a timing loop:

```
int i;
for(i=0; i<10000; i++)
    continue;              // wait one microsecond
```

Functionally, this code does nothing. It assigns values to `i` (eventually stopping at 10,000) but `i` is not something that is externally observable. Given that no code following has external behavior that is dependent on the value of `i`, this entire sequence can be removed. Such action is well within the allowed behavior of a compiler, but if this code was intended to introduce a runtime delay, this *does* affect the operation of your code.

Modifying something externally observable is one manifestation of what is called a side effect. One way of informing the compiler that the above code does have a functional action is to introduce a side effect through the use of the `volatile` specifier. This is described next.

USING VOLATILE TO PREVENT OPTIMIZATIONS

Any access of a `volatile` object denotes a side effect, but making something `volatile` can be interpreted as meaning:

The object might be externally observable.

If it is observable, then code that modifies, or even reads it, can never be deleted.

A change in the object's value may be unknown to the compiler.

The compiler must then ensure the variable is stored in only one permanent location and is read every time its value is needed.

Use of the `volatile` specifier will prevent removal of the timing loop code example shown in the previous section. But this qualifier should be used in other situations, as well. Concurrent operations, like interrupts, are a key part of real-time programming. It is often necessary to share variables between mainline code and ISRs, or even between multiple ISRs. Accessing shared variables can be a problem under optimization. The compiler cannot know when interrupt code will be active, hence it cannot know when concurrent access to shared variables is being made. In this situation, the `volatile` specifier should be used to tell the compiler that the variable's value may change at any time. The compiler will alter the way it stores and reads such variables, and this code will never be removed under optimization.

The `volatile` specifier does not, however, guarantee that an object will be accessed atomically, which is what is required to ensure shared variables are not corrupted. Atomic access is that which uses an assembly instruction sequence that cannot be interrupted (typically a single instruction).

The value assigned to `x` in the simple assignment:

```
volatile long x = 3;
```

(which looks like it is atomic) could be corrupted by an assignment to the same variable in an ISR.

An 8-bit compiler will typically use four 8-bit writes to encode this statement, creating a relatively wide window of time in which an interrupt may occur. If an interrupt routine occurs during this period and that routine modifies `x`, the final value contained in `x` may be *neither* of the values assigned in either statement. This is a common source of mysterious bugs in real-time code.

Data corruption can be avoided by disabling interrupts when modifying shared variables. Or, you can take this one step farther and incorporate this procedure in an access function that modifies these variables. Access functions provide a single point in the program at which these variables can be modified. Provided they are exclusively used to modify these variables, then the chances of corruption are greatly reduced.

CASTING

Casting is an explicit change of data type applied by the programmer to an expression at a point of use in the source code. Rarely do casts need to be added to code as the compiler automatically performs implicit type coercions whenever they are required. It is best to write expressions in which no explicit or implicit type conversions are necessary. A cast should only be added when the type chosen by the compiler is not the type desired by the programmer and when the desired type is contextually legal.

Variables should be declared in an appropriate type for their intended use. If the same variable needs to be used with more than one type, the Microchip Language Tool development team recommends declaring the variable as a `union` type.

One consequence of using an explicit cast is that the compiler will no longer warn of suspicious type conversions, so not only are they usually unnecessary – they may hide potential problems.

The same issues apply when pointer types are explicitly cast, but pointers are less tolerant to any change in type. Some type changes are permissible, but many are not.

Consider only the following pointer type changes:

To and from `void *` pointer types

The generic pointer type can be used to hold any data pointer type, excluding function pointers.

To a `char *` pointer

You can indirectly access any object as individual chars, but not via any other type. Converting *from* `char *` to another pointer type is not acceptable.

To and from any function pointer type

Any function pointer can act as a generic function pointer.

With the exception of `char *` pointers, you must always ensure that the pointer has a type that matches the data or function being indirectly accessed before you dereference them.

The following is an example of code that does not follow the guidelines stated above and is unreliable:

```
float x = 186000;
int exponent = ((*(int *)(&x)) & 0x7F800000) >> 23;
```

Dynamic Memory Allocation

The Microchip Language Tool development team strongly urges that developers refrain from using dynamic memory allocation in functional safety applications. Statistically, many real-time failures can be attributed to faults in memory allocation systems, such as fragmentation, mishandling error conditions, or exhaustion of the available memory resources. Dynamic memory allocation is *never* a requirement and often leads to little-understood costs and inefficiencies. Microchip recommends that you do not use dynamic memory allocation, including the standard C functions, such as `alloc`, `malloc` or `free`; or non-standard functions such as `alloca`; or any other means of dynamically allocating memory.

NEXT STEPS

Microchip support for your safety development does not stop with the delivery of this safety manual. Customers have a wide range of support options, such as:

- Access MPLAB XC compiler documentation, including safety docs and application notes, on the website: www.microchip.com/mplabxc
- Discuss questions and concerns with experts and other developers using the Microchip forums: http://www.microchip.com/forums/f499.aspx
- Attend a training class delivered by Microchip Regional Center instructors

THE MICROCHIP WEBSITE

Microchip provides online support via our www.microchip.com website. This website also provides files and information that can be easily accessed by customers.

Accessible by using your favorite Internet browser, the website contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

CUSTOMER CHANGE NOTIFICATION SERVICE

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip website at www.microchip.com. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the website at:
http://microchip.com/support

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.

Microchip received ISO/TS-16949:2009 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.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
═ ISO/TS 16949 ═

Trademarks

ISBN: 978-1-5224-1053-9

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

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon

Hong Kong
Tel: 852-2943-5100
Fax: 852-2401-3431

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

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

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

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

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

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

China - Hangzhou
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

China - Hong Kong SAR
Tel: 852-2943-5100
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: 86-21-3326-8000
Fax: 86-21-5407-5066

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

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

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

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

ASIA/PACIFIC

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

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

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

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

India - Pune
Tel: 91-20-3019-1500

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

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: 886-3-5778-366
Fax: 886-3-5770-955

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

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

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

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

France - Saint Cloud
Tel: 33-1-30-60-70-00

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-67-3636

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: 47-7289-7561

Poland - Warsaw
Tel: 48-22-3325737

Romaina - 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