PIC1XF1XXX Software Migration

INTRODUCTION

After years of success, the PIC12/16 mid-range microcontroller has been refreshed to be more suited for C and increase the performance for many typical applications.

These changes include:

- Additional instructions
- More memory
- New indirect addressing modes

Every effort was taken to assure the forward compatibility of the legacy code base but in a few cases, there are differences that must be reconciled by adjustments to the software. This PIC1XF1XXX Software Migration document has three sections describing the new design, teaching how to migrate existing software and introducing many software optimizing techniques.

Note:	This device has been designed to perform to the parameters of its data sheet. It has been tested to an electrical specification designed to determine its conformance with these parameters. Due to process differences in the manufacture of this device, this device may have different performance characteristics than its earlier version. These differences may cause this device to perform differently in your application than the earlier version of this device.

Note:	The user should verify that the device oscillator starts and performs as expected. Adjusting the loading capacitor values and/or the oscillator mode may be required.

ARCHITECTURE

New Features

The enhanced PIC12/16 extends the architecture with the following features:

- Program memory extended to 32KW (56KB)
- Data memory extended to 2KB
- 14 new instructions
- Linear mapping
- Simplified core register map
- Enhanced indirect addressing functions
- Automatic interrupt context save

These enhancements can increase the performance of many applications while remaining largely compatible with existing software.

Core Registers

The PIC12/16 has always had a core set of registers that are present in every device. One of the goals for the enhanced version was to collect the core registers into a standard location. The first 12 SFR addresses of every bank are now the core registers, as shown in Table 1.

TABLE 1: SFR ADDRESSES

Address	Register	Description
0x00	INDF0	Indirect File Register 0
0x01	INDF1	Indirect File Register 1
0x02	PCL	Program Counter Low
0x03	STATUS	ALU Status Register
0x04	FSR0L	Indirect Address for INDF0, Low Byte
0x05	FSR0H	Indirect Address for INDF0, High Byte
0x06	FSR1L	Indirect Address for INDF1, Low Byte
0x07	FSR1H	Indirect Address for INDF1, High Byte
0x08	BSR	Bank Select Register
0x09	WREG	W Register
0x0A	PCLATH	Program Counter High Byte Latch
0x0B	INTCON	Interrupt Control Register

The core registers in the enhanced PIC12/16 are the same as the legacy registers with the addition of a second FSR/INDF, the BSR and mapping the W register in WREG. The IRP, RP0 and RP1 bits are no longer present in the STATUS because the BSR is now available and the FSRs are now 16 bits wide.

New Instructions

The new instructions improve arithmetic, simplify paging and banking, and extend the capabilities of the indirect addressing modes.

TABLE 2: NEW INSTRUCTIONS

Mnemonic	Operands	Description	Cycles(1)	Status Affected
ADDFSR	k	Add literal to FSRn	1	—
ADDWFC	f,d	Add W and F with Carry	1	C,DC,Z
SUBWFB	f,d	Subtract W from F with Borrow	1	C,DC,Z
ASRF	f,d	Arithmetic Shift Right	1	C
BRA	k	Branch Relative	2	—
BRW	-	Branch Relative with W	2	—
CALLW	-	Call Absolute with W	2	—
LSRF	f,d	Logical Shift Right	1	C
LSLF/ASRF	f,d	Logical Shift Left	1	C
MOVLB	k	Move Literal to BSR	1	—
MOVLP	k	Move Literal to PCLATH	1	—
MOVIW	*	Move INDFn to W	1	Z
MOVWI	*	Move W to INDFn	1	—
RESET	-	CPU Reset	1	C,DC,Z

Note 1: All cycle counts increase by 1 if the file address points to INDF (0 or 1) and FSR points to program memory.

These instructions were designed to fit inside the “holes” in the existing 14-bit instruction word. By fitting the existing memory design, the cost of the enhanced devices was kept similar to the existing PIC12/16, simplifying the decision to use the new features.

Stack

The legacy 8-level stack has been extended to a 16-level stack. This stack behaves exactly as the legacy stack wrapping on an over or underflow. The stack can be configured to cause a Reset on over or underflow, and the stack is accessible via stack access registers in Bank 31.

STACK OPERATION

The Stack Pointer always points at the last value placed on the stack. If a call or an interrupt occurs, the stack is incremented and then the next Program Counter (PC) value is saved.

FIGURE 1: STACK OPERATION

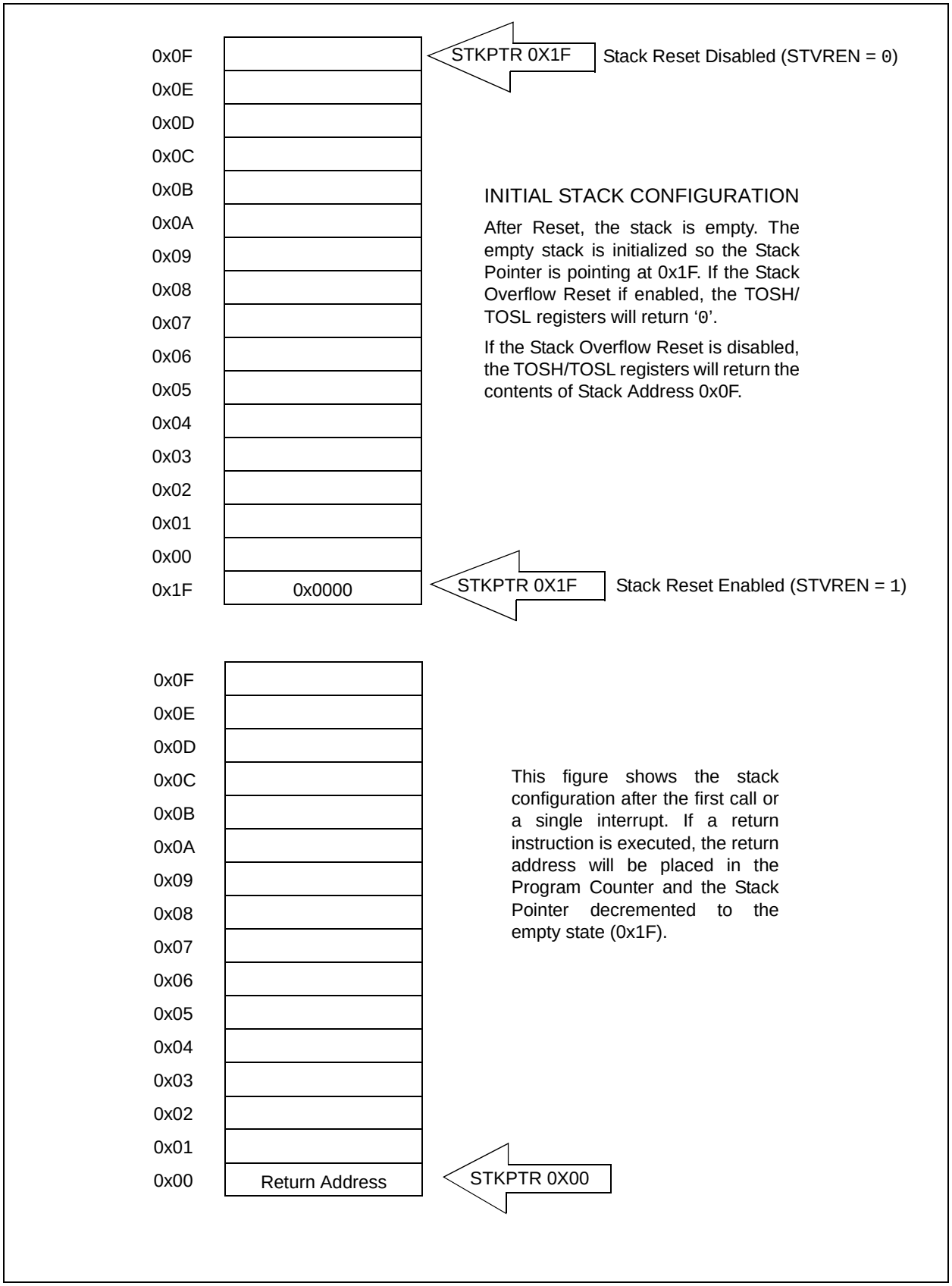
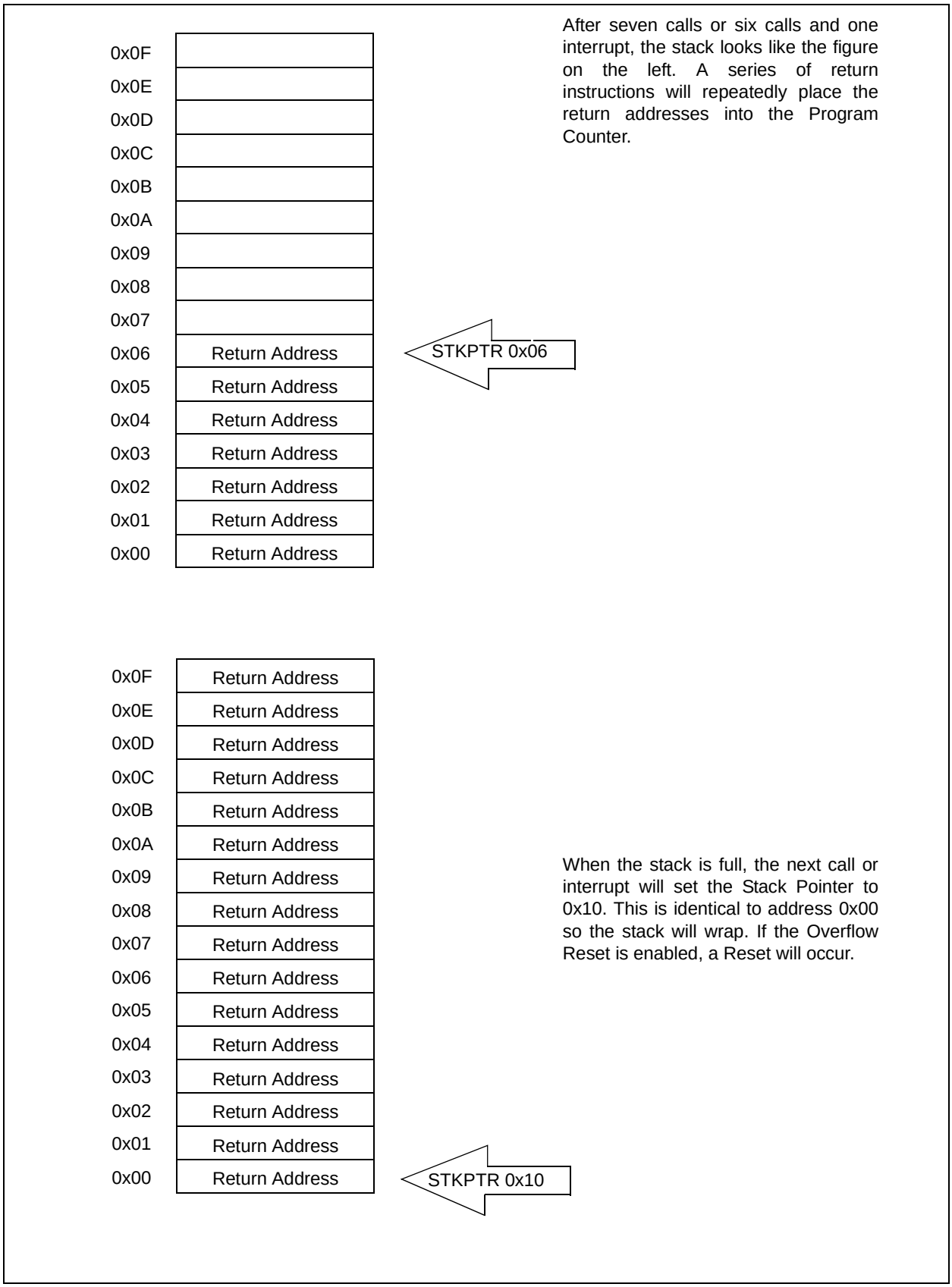


FIGURE 2: STACK OPERATION



OVERFLOW/UNDERFLOW

The new stack has 16 entries but a 5-bit Stack Pointer. This Stack Pointer is visible as the STKPTR register in Bank 31. When the MSb of STKPTR is set after a call or interrupt, the stack is considered in overflow. If the optional Reset is enabled, the Program Counter will go immediately to '0' and the reset condition flag in PCON will indicate a Stack Reset. If the STKPTR reads 31 (all bits set) then the stack is considered empty. A return (RETURN, RETLW, RETFIE) when the stack is empty is an underflow. If the optional Reset is enabled, the Program Counter will go immediately to 0x0000 and the Reset condition flag in PCON will indicate a Stack Reset. If the optional Reset is not enabled, the stack will continue to operate but underflow and overflows will simply wrap around the 16 entry stack. The Stack Overflow and Underflow bits can be checked to determine if a stack is out of bounds.

DEBUGGING ACCESS

The legacy PIC12/16 shared the 8 levels of stack with the in-circuit debugging firmware. This could be very restrictive. The enhanced PIC12/16 has an additional stack reserved for the debugging firmware. This eliminates the possibility of running out of stack when using Debug mode. When debugging, the Stack Reset option becomes a break point to help determine the cause of the stack problem. The stack access mechanism is also used by the debugger to provide the engineer visibility to the call chain that led the software to its present state.

USER ACCESS

The user has full access to the 16 levels of user stack. The debug stack is unavailable. Access to the stack is through three registers; Stack Pointer (STKPTR), Top-of-Stack High (TOSH) and Top-of-Stack Low (TOSL). These registers are located in Bank 31.

MODIFYING THE STACK

Modifying the stack requires that the STKPTR be adjusted to point to the entry that needs updating, and TOSH/TOSL be changed to reflect the modification. The STKPTR register always points at the last entry placed on the stack. When a call or interrupt is executed, the STKPTR register is incremented and the PC is stored at the new TOSH/TOSL location. Make sure that interrupts are disabled before modifying the stack.

Relative Branching

Relative branching is branching to a target address that is based upon the current address. With the legacy PIC12/16, we often wrote code such as the following:

```
Label
    DECFSZ      delay
    GOTO        $-1
```

When we write the $-1 we are requesting a relative branch, except the assembler is converting that into an absolute branch. The payload size of CALL and GOTO created the 2k page size so CALL and GOTO contribute to the paging problem.

The enhanced PIC12/16 adds a BRA and BRW relative branch. The BRA allows you to branch +256 or -255 instructions from the current program counter. The previous code example would look like this.

```
Label
     DECFSZ     delay
     BRA        Label
```

Because Label is an absolute address, the assembler will convert the absolute address to a relative address by subtracting the current Program Counter from the destination address. If the $-1 syntax is used, it will work in the same way. If the constant -1 is supplied to BRA it will not work because -1 specifies an address at the end of the program memory and that will most likely be out of range.

FASTER TABLE READS

The BRW instruction is also a relative branch but the reach is from 0 to 255. The relative address for BRW is contained in the W register. This allows the traditional table read method to execute much faster without regard for table location.

EXAMPLE 1:

```
Table
        MOVLW     3
        BRW
        RETLW     'a'
        RETLW     ' '
        RETLW     's'
        RETLW     't'
        RETLW     'r'
        RETLW     'i'
        RETLW     'n'
        RETLW     'g'
```

In the example above, the letter 't' is returned to the calling function.

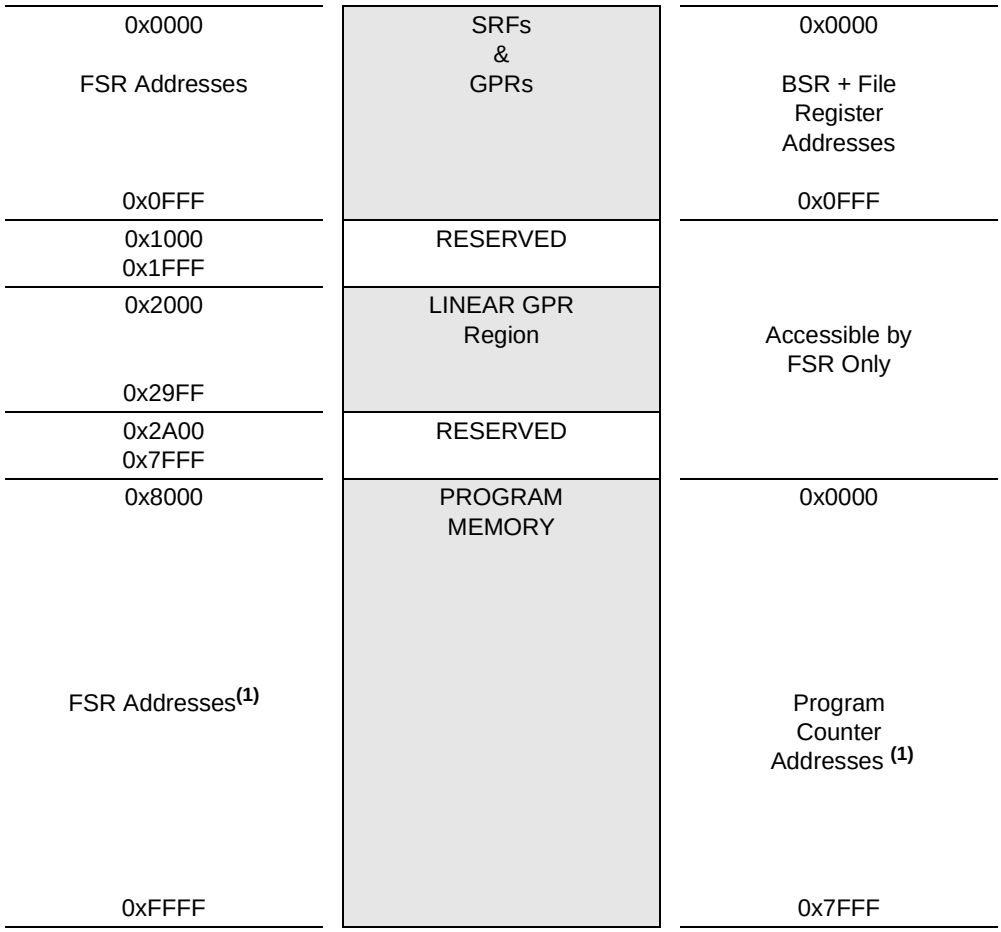
NO PAGING SURPRISES

The primary advantage to using relative branches is to eliminate the possibility of software that works until additional code is added. Sometimes, a simple delay loop will fail because code inserted ahead of the loop pushed the loop across a page boundary. These surprises are eliminated with the use of a relative branch.

Indirect Addressing

Indirect addressing allows an address to be computed at run time and the data modified. Accessing arrays and other memory buffers often requires indirect addressing. The legacy PIC12/16 has very rudimentary support for indirect addressing with a single FSR/INDF. The enhanced PIC12/16 has extensive support for indirect addressing.

FIGURE 3: FSR MEMORY MAP



Note 1: Accessing these addresses via FSR causes the instructions to use 1 additional cycle.

FSR MEMORY MAP

The large addressing space of the new FSRs provides the ability to access additional memory besides the GPR and SFR memory spaces (see Figure 3). The FSR memory map includes the legacy GPR/SFR space. In addition, the GPR memory is mirrored into a linear region to allow large memory blocks to be accessed via the indirect addressing modes. Lastly, the low byte of each program memory address is also mapped into the FSR address space. These two additions to the memory map allow large memory blocks, and pointers that access RAM and Flash. The result of these changes is a large improvement in performance for large data applications.

FSR INSTRUCTION SUPPORT

Indirect addressing on the enhanced PIC12/16 consists of two 16-bit File Select Registers (FSRs) and 2 Indirect File Registers (INDFs). In addition to the extra FSR/INDF there are three new instructions designed to improve the efficiency of indirect operations. The three new instructions are:

1. ADDFSR – Add a literal between -32 and 31 to the specified FSR.
2. MOVIW – Move a value from the specified INDF register into W.
3. MOVWI – Move a value from W into the specified INDF register.

The MOVIW and MOVWI instructions are special because they have the ability to perform pre/post increment/decrement on the FSR. They can also perform relative indirect addressing.

USING MOVIW/MOVWI

MOVIW and MOVWI have the following syntax:

EXAMPLE 2: MOVIW AND MOVWI

```
MOVIW        ++FSR0    ; Preincrement FSR0 then INDF0 -> W
MOVIW        FSR0++    ; INDF0 -> W then postincrement FSR0
MOVIW        --FSR0    ; Predecrement FSR0 then INDF0 -> W
MOVIW        FSR0--    ; INDF0 -> W then postdecrement FSR0
MOVIW        4[FSR0]   ; FSR0+4 INDF0 -> W.  FSR0 unchanged
```

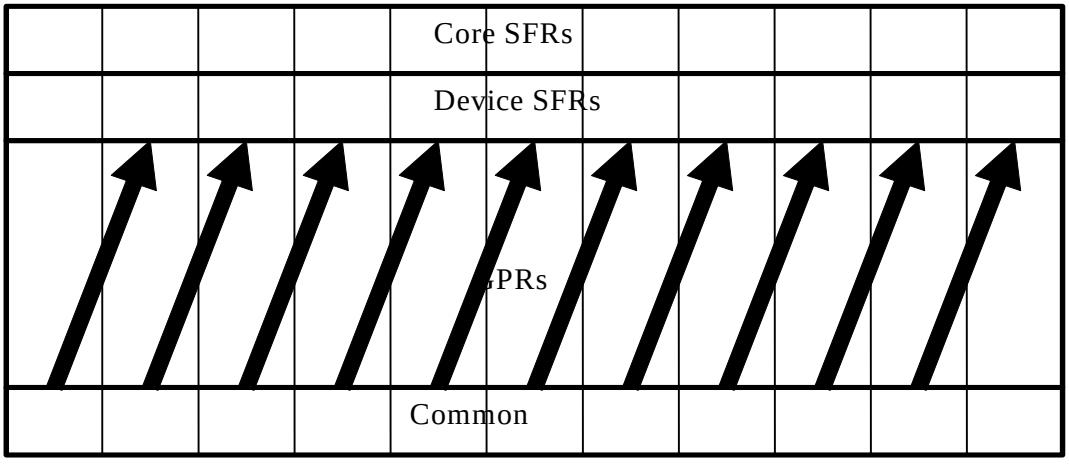
The pre/post, increment/decrement are very simple enhancements. The relative offset is slightly more complex. In the relative offset case, a constant between -32 and +31 is added to the FSR to produce an effective address. The INDF accesses the effective address. After the access is complete (either read or write) the effective address is lost and the FSR remains unchanged.

LINEAR RAM ACCESS

To simplify the use of large memory blocks, the FSRs also provide a method of remapping the data memory into a contiguous block of RAM. The FSR can provide an alternate mapping by taking advantage of the 64-Kbyte address space to locate a different mapping of the GPR into a different memory region.

The GPRs and SFRs are mapped into the first 2 Kbytes of address space in the FSR. Following the GPR/SFR mapping, there is a reserved area (read as '0') and then comes a new view of the GPR data space at address 0x2000, as shown in Figure 4.

FIGURE 4: LINEAR MEMORY ACCESSING



In the new view, the 80-byte blocks of GPR memory are stacked without the common memory or the SFRs to separate them. This allows a 90-byte data structure to span two banks of memory without concern for overwriting unrelated memory locations. To use this view of GPR memory, simply point an FSR at a location after 0x2000.

PROGRAM MEMORY READ VIA FSR

Another useful feature of the indirect addressing system is access to the low 8-bits of the program memory.

Access to the program memory starts at FSR address 0x8000. The following code example shows how to access a memory table with the FSR.

EXAMPLE 3:

```
Code
       MOVLW high table_label
       MOVWF  FSR0H
       MOVLW  low table_label
       MOVWF  FSR0L
       MOVF   FSR0,W
       ...
       Lots More Code
       ...

table_label
       MOVLW  0xAA
       MOVLW  0xAB
.. More Table Data ...
```

The compiler knows when a label references a program memory address. If a program memory address is the target of the high directive, the high directive will return the high part of the address with the MSB set. This will assure that the FSR address is a proper address for accessing the program memory.

When accessing program memory with the FSR, each read of the INDF register will take 2 cycles. The second cycle is needed to fetch the data.

Banking and Paging

Banking and paging have always been a complication of PIC12/16 microcontrollers. To further enhance the new PIC12/16, new instructions were added to simplify paging and banking. These new instructions also allow the number of pages and banks to increase to 32 banks and 16 pages.

BANKING INSTRUCTION

Increasing the number of banks to 32 was impossible with the RP0, RP1 bits in the STATUS register. These bits were removed and replaced with a BSR register. Along with BSR, a new instruction was added to load the BSR from a literal. The MOVLB instruction does in one cycle what the BANKSEL macro currently does in one or two instructions depending upon the device memory size. By making banking one instruction, the penalty for banking was reduced by up to 50%.

PAGING INSTRUCTION

The need for paging is due to the CALL and GOTO instructions. These two instructions can reach any address inside of a 2K word window. To leave this window, the PCLATH must be updated before the call or GOTO can take place. To increase the efficiency of editing PCLATH, a new instruction, MOVLP, has been added. MOVLP is the instruction equivalent of the PAGESEL macro. In the enhanced PIC12/16, every CALL or GOTO could be preceded by a MOVLP for a 2-word, 3-cycle operation that can reach the entire program memory.

MATH

Arithmetic on the PIC12/16 has always been very easy but multi-byte operations could be made easier by adding support for carry and arithmetic shifts.

ARITHMETIC WITH CARRY

Add and subtract have been extended to support carry or borrow. Only adds and subtracts with file registers are supported with the Carry or Borrow. The new instructions work exactly as the legacy instructions except the Carry/Borrow flag is included in the operation. The original add and subtract instructions are still present in the enhanced PIC12/16. All existing algorithms will work unchanged but a speed-up is possible by selectively replacing some instruction sequences with the new instructions.

SHIFTS INSTEAD OF ROTATES

The legacy PIC12/16 has a pair of 9-bit rotate instructions. These instructions rotate through the Carry flag so 9 rotates will result in the original value. In addition to the rotates, the new PIC12/16 has 3 shift instructions.

1. ASRF – Arithmetic Right Shift
2. LSRF – Logical Right Shift
3. LSLF – Logical Left Shift
4. ASLF – Arithmetic Left Shift is assembled as LSLF

Shifts are similar to rotates with the exception that the incoming digit is not drawn from the Carry flag but it is either a '0' or a sign extension. An arithmetic shift performs sign extension, while logical shifts bring in zero. Sign extension means the Most Significant bit (MSb) is duplicated. This is a very useful feature because a negative number that is right shifted remains a negative number. If a multi-byte value is being shifted, then the first byte is shifted and the remaining bytes are rotated. That will utilize the Carry flag correctly so all the bits propagate across the bytes. Rotates and Shifts are now more functional because the W register is mapped as the file register WREG. This allows rotates and shifts to directly operate on the W register.

TABLE 3: LOGICAL SHIFT OPERATIONS

Starting Value	Operation	Final Value
0x7F	ASRF	0x3F, C = 1
0x80	ASRF	0xC0, C = 0
0x7F	LSRF	0x3F, C = 1
0x80	LSRF	0x40, C = 0
0x7F	LSLF/ASLF	0xFE, C = 0
0x80	LSLF/ASLF	0x00, C = 1

Table 3 above shows the differences between the different shift operations.

Interrupts

Interrupts on the enhanced PIC12/16 are essentially unchanged but for the addition of a hardware context-save. This reduces the interrupt overhead by eliminating the essential task of saving key registers such as the W, or STATUS at the beginning of the Interrupt Service Routine (ISR) and restoring them at the end.

HARDWARE CONTEXT SAVING

The hardware context-save system consists of a number of registers located in Bank 31. These registers contain the backup copy of the saved context and are the source of the restored context. The registers saved are shown in Table 4.

TABLE 4: HARDWARE CONTEXT SHADOW LOCATIONS

Register	Context-Saved Register	Bank 31 Address
WREG	WREG_SHAD	0xFE5
STATUS	STATUS_SHAD	0xFE4
FSR0L	FSR0L_SHAD	0xFE8
FSR0H	FSR0H_SHAD	0xFE9
FSR1L	FSR1L_SHAD	0xFEA
FSR1H	FSR1H_SHAD	0xFEB
BSR	BSR_SHAD	0xFE6
PCLATH	PCLATH_SHAD	0xFE7

If the context needs to be dynamically adjusted, then the context-save registers are necessary to accomplish this.

Coding Practices for the Enhanced PIC12/16

PAGING MINIMIZATION TECHNIQUES

Minimizing paging can be accomplished by the following:

1. Utilize relative branches whenever possible.
2. Update the RETLW tables to use BRW or the FSR/INDF program memory read functions.

Most applications can minimize paging impact simply by using relative branches.

RECOMMENDED PRACTICES

If your code currently uses the PAGESEL macro, it now produces a MOVLP instruction. If your code currently uses the BANKSEL macro, it will now produce a MOVLB instruction. Use PAGESEL/BANKSEL to produce migrateable code with the least effort.

ROBUST PROGRAMMING

Robust programming is an important issue in many areas. To better support robust practices, the RESET instruction was added, the optional Stack Reset was included, and the ability to examine the stack was added.

SOFTWARE RESET

The RESET instruction performs a complete system Reset. This replaces the previous practice of waiting for the Watchdog Timer to provide a Reset. Reset should be placed around critical code blocks to reduce the possibility of Program Counter upsets from causing improper code execution.

STACK RESET

The Stack Reset allows an application to reset when the stack overflows or underflows. In critical applications such as medical or aerospace, a stack monitor can minimize erroneous behavior.

RESET MONITORING

With so many more ways to Reset, it can be a challenge identifying the Reset source for proper recovery.

The PCON register has been extended to show:

- Stack Overflow
- Stack Underflow
- Reset by Instruction
- Reset by POR
- Reset by $\overline{\text{MCLR}}$

SOFTWARE MIGRATION

Software migration can be a tricky problem. Every effort was made to minimize the complexities of converting legacy code to run correctly on the enhanced PIC12/16.

MOVING FROM PIC1XFXXX TO PIC1XF1XXX

Migrating software from the legacy PIC12/16 devices to the enhanced PIC12/16 consists of changes for paging/banking, adjusting the interrupt context saving and modifying the indirect memory access functions. After the software runs, additional changes can be made to further improve the performance.

Banking

To adjust the banking for the new device, all references to RP0 and RP1 must be replaced with MOVLB instructions. Usually the references to RP0 and RP1 are inside of macros such as BANKSEL or simply BANK0. MPASM™ assembler provides a BANKSEL macro that inserts the correct RP0 and RP1 writes for the legacy PIC12/16. MPASM assembler inserts the MOVLB instruction for BANKSEL on the enhanced PIC12/16. If your software uses a homebuilt BANKx type macro, simply make a new version that uses MOVLB or replace all the BANKx macros with BANKSEL macros. The best general solution is simply to use the built-in BANKSEL macro.

Paging

There are two ways to handle paging issues. The first way is to simply find all the writes to PCLATH and adjust them. Most of them will continue to work correctly but the MOVLP instruction will make them faster. If your code uses the built-in MPASM assembler macro PAGESEL then MOVLP will be used automatically. For further acceleration, the BRA and BRW instructions can be used to eliminate the need for many PAGESEL/GOTO combinations.

Interrupts

Automatic Context Save

Interrupts can remain unchanged but the context time can be dramatically improved by eliminating the context save code. Most Interrupt Service Routines do not need any additional context saved beyond the registers saved by the hardware. Simply removing the context saving code will improve the context switch time by approximately 12 instruction cycles.

Return Without Context Restore

If your application needs to return without restoring the context, the following code can be used:

```
bsf INTCON,GIE
return
```

Your application must handle context. This instruction sequence will work because the instruction after setting the GIE will always execute.

INDIRECT MEMORY ACCESS

Indirect memory accessing is using the FSR/INDF registers to access parts of memory. In the legacy PIC12/16, the FSR is 9 bits wide with the 9th bit stored in STATUS as IRP. The enhanced PIC12/16 has a 16-bit wide FSR with the Most Significant bits stored in the FSRnH register. To access the entire RAM, the Most Significant bits need to be adjusted. This is often done with the BANKISEL macro. The BANKISEL macro would set or clear the IRP bit as required. On the enhanced PIC12/16, the BANKISEL macro will set or clear each bit in FSRnH as required. This will take 8 instruction cycles. If you are willing to write to W, then the update can be done in two cycles by MOVLW and MOVWF or use ADDFSR, or MOVIW/WI with relative offset.

MOVING FROM PIC18FXXX TO PIC1XF1XXX

Moving back from the PIC18 to the enhanced PIC12/16 is different than moving forward from the legacy PIC12/16. The enhanced PIC12/16 lacks some resources:

- 16 stack locations instead of 31
- Different FSR relative addressing mechanism (instructions instead of registers)
- No Multiply instructions
- Shorter CALL/GOTO reach
- 2 FSRs vs 3

Banking

Banking on the PIC18 is often misunderstood. The PIC18 does have banking for the RAM accesses. It also has a special bank that contains nearly all of the SFRs. Each bank on the PIC18 is larger (256 bytes), so the amount of time spent banking is reduced. The enhanced PIC12/16 still has the 128-byte bank size and no Access Bank. Converting the software requires that the BANKSEL macro be used for many SFR accesses and RAM accesses to assure the bank is correctly configured.

Paging

Paging is eliminated on the PIC18 by making the CALL and GOTO two word instructions. The additional instruction payload gives CALL and GOTO the ability to reach any instruction in memory. Simply adding a PAGESEL before all CALLs and GOTOs will provide the same capability to the enhanced PIC12/16 at the cost of one cycle. The PIC18 relative branch (BRA) is also present on the enhanced PIC12/16. In some cases, the BRA will need to be replaced with a GOTO because the reach of the PIC18 BRA is longer than the reach of the PIC12/16 BRA.

All instruction addresses on a PIC18 are even because the 16-bit instruction memory is byte addressed. This may have an effect on the enhanced PIC12/16 conversion as its program memory is word-addressed, so all instructions are sequentially addressed.

INTERRUPTS

The PIC18 has a 2-level interrupt structure. The interrupts have a fast context-save but the restore is optional, depending upon the use of a fast return from interrupt instruction. All interrupts will need to be adjusted to understand the flat interrupt structure of the enhanced PIC12/16 and all context restores will need to be removed.

INDIRECT MEMORY ACCESS

The PIC18 has 3 FSRs, special registers for addressing, and INDF SFRs for increment and decrement. These can be replaced with the appropriate use of the MOVIW and MOVWI instructions on the enhanced PIC12/16.

PROGRAM MEMORY TABLES

The PIC18 has table read instruction to retrieve 2 bytes from program memory.

The enhanced PIC12/16 can retrieve 1 byte from program memory by using the FSR/INDF to access the program memory.

SOFTWARE OPTIMIZATION

After the software has been migrated to the enhanced PIC12/16, the opportunity exists to increase the performance by taking full advantage of the new features. Software optimization for the enhanced PIC12/16 can be classified as changes to the following areas:

1. Arithmetic
2. ROM Table Access
3. RAM Buffer Access
4. Data Structures

ARITHMETIC

Arithmetic on the PIC12/16 is a very basic function but can be full of small frustrations when working with multi-byte data. For example, the legacy ADD instructions do not include an add with Carry, so multi-byte operations require additional instructions to precondition the next Most Significant Byte (MSB) based upon the results of the previous add. New instructions added to the enhanced PIC12/16 simplify this work, shaving many instructions from common algorithms.

Multi-byte Add

Adding two 16-bit values with a PIC12/16 is usually done as follows:

```
Add16
        MOVF    lsb_a,w
        ADDWF   lsb_b,f
        MOVF    msb_a,w
        BTFSC   STATUS,C
        ADDLW   0x01
        ADDWF   msb_b,f
```

This code works quite well but has 2 extra cycles in the middle, due to the missing add with Carry. This code can be converted to the enhanced PIC12/16 as follows:

```
Add16_enhanced
        MOVF    lsb_a,w
        ADDWF   lsb_b,f
        MOVF    msb_a,w
        ADDWFC  msb_b,f
```

The enhanced version saves two instructions. If 16-bit values are used extensively, the savings will add up.

Multi-byte Subtract

Subtract is similar to add because the Borrow flag must be handled in the following computations. The legacy code is as follows:

```
Sub16
        MOVF    lsb_a,w
        SUBWF   lsb_b,f
        MOVF    msb_a,w
        BTFSS   STATUS,C
        ADDLW   0xFF
        SUBWF   msb_b,f
```

The enhanced version is as follows:

```
Sub16_enhanced
        MOVF    lsb_a,w
        SUBWF   lsb_b,f
        MOVF    msb_a,w
        SUBWFB  msb_b,f
```

The enhanced version of subtract saves two instructions.

Shifting

Shifting is the fastest method to divide or multiply by a power of two, therefore, it is often used instead of rotates. A rotate instruction can be made to operate like a shift simply be preconditioning the Carry flag. The legacy code is as follows:

```
Arithmetic_Right_Shift
       RLF   val,w ;save the MSB
                    in the Carry
                    (corrupt W
       RRF   val,f ;rotate right
                    with sign extend.
```

This code will work but the W register is overwritten by the first rotate. The ASRF instruction simplifies this problem by handling the sign extension in hardware.

```
       ASRF  val,f
```

This is faster by requiring less code. Arithmetic _Left_Shift does not need to perform a sign extension. It simply brings zero into the LSB.

If sign extend is not required, the code is simpler.

```
Logical_Right_Shift
        BCF    STATUS,C
        RRF    val,f

Logical_Left_Shift
        BCF    STATUS,C
        RLF    val,f
```

Simply clearing the Carry flag before performing the rotate will create a logical shift. These examples can be converted to single enhanced PIC12/16 instructions.

```
LSRF   val,f  ;  logical right shift
LSLF   val,f  ;  logical left shift
ASLF   val,f  ;  Arithmetic left shift
              ;  really   logical   left
                 shift)
```

Notice that the Arithmetic_left_shift and the logical_left_shift are identical functions. The enhanced PIC12/16 only provides the logical_left_shift but the assembler will substitute the logical_left_shift if ASLF is used in the assembly language.

TABLE 5: SHIFT OPERATION DIFFERENCES

Starting Value	Operation	Final Value
0x7F	ASRF	0x3F, C = 1
0x80	ASRF	0xC0, C = 0
0x7F	LSRF/ASLF	0x3F, C = 1
0x80	LSRF/ASLF	0x40, C = 0
0x7F	LSRF/ASLF	0xFE, C = 0
0x80	LSRF/ASLF	0x00, C = 1

Table 5 above shows the differences between the different shift operations.

MULTI-BYTE SHIFTS

A multi-byte shift can be created simply by making the first operation the new shift instruction and all subsequent operations rotate. The new instructions shift the outgoing bit into the Carry flag where the rotate instruction will bring it into the next byte.

```
ASRF_16
            ASRF     MSB_f
            RRF      LSB_f
```

ROM Table Access

Accessing data tables is frequently performed in many applications. The traditional method on a PIC12/16 device is to assemble the table into a series of RETLW instructions. By adding a value to the PC a value at a specific offset can be returned to the calling function. The enhanced PIC12/16 has a few improvements.

Fast Tables with FSR

The first trick is to simply use the FSR/INDF to return values saved in program memory. When the MSB is set in the FSRnH register, the value returned by INDF is the Least Significant Byte (LSB) in the program memory address specified by the FSR register. By using the FSR to point to data in the program memory, the additional capabilities of the MOVIW and MOVWI instruction can be used to keep the software short. Generally, the setup to use the FSR takes 4 cycles, while each fetch can take 2 because fetches to program memory take an extra cycle. The primary advantage to using the FSR to fetch data from program memory is the unified pointer. Data can be located in GPR or program memory with no change in the access requirements.

```
code
      MOVLW     HIGH data_table
      MOVWF     FSROH
      MOVLW     LOW data_table
      ADDWF     data_index
      MOVWF     FSR0L
      CLRW
      ADDWFC    FSR0H
      MOVF      INDF0, W

data_table
     RETLW      'm'
     RETLW      'y'
     RETLW      ' '
     RETLW      'd'
     RETLW      'a'
     RETLW      't'
     RETLW      'a'
```

Note: The HIGH directive will automatically set the MSB if the label references an address in program memory.

Fast RETLW Tables with BRW

Using the BRW instruction combined with RETLW will assure that all instruction fetches take 4 cycles plus the initial call. The traditional methods can also run this fast if the data table is located to prevent problems with the program memory page size.

```
code
     MOVLW     data_index
     PAGESEL   data_table
     CALL      data_table

data_table
     BRW
     RETLW     'm'
     RETLW     'y'
     RETLW     ' '
     RETLW     'd'
     RETLW     'a'
     RETLW     't'
     RETLW     'a'
```

If CALL data_table and data_table are not separated by a page boundary, the PAGESEL is not required to assure the CALL can reach. This method never requires the data_table be aligned to any memory address. Eliminating the alignment requirement makes BRW very easy to use. This method of table access requires 6 or 7 cycles, depending on the presence of PAGESEL.

Fast RETLW Tables with CALLW

The CALLW instruction can be used to access data tables anywhere in memory for a fixed 6-cycle time. CALLW is a new instruction intended to make function pointers more efficient. In addition to its intended function, CALLW can also be used to prepare tables of data. CALLW works by concatenating the value in W with the value in PCLATH and "calling" the resulting address.

Locating the block of RETLW instructions on a 256-word boundary (low address byte as zero) allows a CALLW to retrieve the data. This can be a very fast table lookup.

```
code
     MOVLP     HIGH data_table
     MOVLW     data_index
     CALLW

     ORG       0x??00
data_table
     RETLW     'm'
     RETLW     'y'
     RETLW     ' '
     RETLW     'd'
     RETLW     'a'
     RETLW     't'
     RETLW     'a'
     RETLW      0
```

This method takes 6 cycles to retrieve any data.

Example

A complete code example is shown in **Appendix B: "IIR Filter Conversion Example"**. In this example, an IIR function downloaded from the internet is converted to use the enhanced PIC12/16 features. The resulting code runs nearly twice as fast.

Software License Agreement

The software supplied herewith by Microchip Technology Incorporated (the “Company”) is intended and supplied to you, the Company’s customer, for use solely and exclusively with products manufactured by the Company.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN “AS IS” CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

APPENDIX A: A SIMPLE GETHEX FUNCTION OPTIMIZATION

```
; Original code

     data
hex_temp res 1

     code
gethex
     GLOBAL        gethex
     ANDLW         0x0F
     MOVWF         hex_temp
     MOVLW         high hex
     MOVWF         PCLATH
     MOVLW         low hex
     ADDWF         hex_temp,W
     BTFSC         STATUS,C
     INCF          PCLATH,F
     MOVWF         PCL
hex
     dt            “0123456789ABCDEF”
```

```
; Enhanced Code

     data
; no data needed

     code
gethex
     GLOBAL    gethex
     ANDLW     0x0F
     BRW

hex
     dt        “0123456789ABCDEF”
```

APPENDIX B: IIR FILTER CONVERSION EXAMPLE

The following code is a 24-bit IIR by Tony Kubek extracted from www.piclist.org (http://www.piclist.org/techref/microchip/dsp/iir-24b-256s-tk.htm). It is a good example of a non-trivial algorithm that can be accelerated by taking advantage of the new instructions and FSR features of the enhanced PIC12/16. The original code is presented in **Appendix C: "24-Bit 256 Step IIR Filter (Original Code)"**.

This is quite a lot of code to look over. There are a few notable code blocks that can be noticed right away. First the ADD_FSR_FILTER macros at the beginning need to be adjusted. The original code is using the FSR so it should be updated to use FSR0L. After further study it can be seen that this code is simply a 24-bit add with the source address in the FSR. That opens the possibility of additional optimizations. The MOVIW instruction can be used to eliminate redundant FSR reloads. The ADDWFC instruction can be used to eliminate the Carry propogation.

```
ADD_FSR_FILTER MACRO
  ; 14-17 instructions

  ; add value pointed to by FSR to filter
   MOVF    INDF,W        ; get lowest byte
  ADDWF NewFilter+3,F    ; add to filter sum
                         ;lowest byte

  DECF  FSR,F
  MOVF  INDF,W           ; get next byte
  SKPNC                  ;  if overflow
  INCFSZ INDF,W          ; increase source
  ADDWF NewFilter+2,F    ; and add to dest.
  DECF  FSR,F
  MOVF  INDF,W           ; get next byte
  SKPNC
  INCFSZ INDF,W
  ADDWF NewFilter+1,F

  DECF  FSR,F
  MOVF  INDF,W           ; get msb
  SKPNC
  INCFSZ INDF,W
  ADDWF NewFilter,F
  ENDM
```

```
ADD_FSR0_FILTER MACRO
  ; 8 instructions
  ; add value pointed to by FSR to filter
  moviw   0[INDF0]

ADDWF   NewFilter+3,F   ; add to filter sum
lowest byte
  MOVIW   -1[INDF0]     ; get next byte
  ADDWFC  NewFilter+2,F
  MOVIW   -2[INDF0]     ; get next byte
  ADDWFC  NewFilter+1,F
  MOVIW   -3[INDF0]     ; get msb
  ADDWFC  NewFilter,F
  ENDM
```

Notice an immediate drop in instruction count from 14-17 instructions to 8 instructions. That is a 50% size advantage and a 2x speed-up. The multiply and divide macros are the next interesting block of code in this example.

```
;++++++++++++
;
; DIV_FILTER_BY2 - Divide NewFilter by 2

;
;
DIV_FILTER_BY2 MACRO
  ; 5 instructions
  ; right shift filter value by 1 ( i.e.
divide by 2 )
  BCF STATUS,C  ; clear carry
  ; divide by 2
  RRF NewFilter,F
  RRF NewFilter+1,F
  RRF NewFilter+2,F
  RRF NewFilter+3,F
  ENDM
```

```
;++++++++++++
;
; DIV_FILTER_BY2 - Divide NewFilter by 2

;
;
DIV_FILTER_BY2 MACRO
  ; 4 instructions
  ; right shift filter value by 1 ( i.e.
divide by 2 )
  ; divide by 2
  LSRF  NewFilter,F
  RRF NewFilter+1,F
  RRF NewFilter+2,F
  RRF NewFilter+3,F
  ENDM
```

```
;++++++++++++
;
; MUL_NEWOLD_BY2 - Multiply OldValue and
NewSample with 2
;
;
MUL_NEWOLD_BY2 MACRO
  ; 10 instructions
  ; right shift filter value by 1 ( i.e.
divide by 2 )
  BCF STATUS,C  ; clear carry
  ; multiply old value with 2
  RLF OldFilter+3,F
  RLF OldFilter+2,F
  RLF OldFilter+1,F
  RLF OldFilter,F
  ; multiply new value with 2
  BCF STATUS,C  ; clear carry
  RLF NewSample+3,F
  RLF NewSample+2,F
  RLF NewSample+1,F
  RLF NewSample,F
  ENDM
```

```
;++++++++++++
;
; MUL_NEWOLD_BY2 - Multiply OldValue and
NewSample with 2
;
;
MUL_NEWOLD_BY2 MACRO
  ; 8 instructions
  ; right shift filter value by 1 ( i.e.
divide by 2 )
  ; multiply old value with 2
  LSLF  OldFilter+3,F
  RLF OldFilter+2,F
  RLF OldFilter+1,F
  RLF OldFilter,F
  ; multiply new value with 2
  LSLF  NewSample+3,F
  RLF NewSample+2,F
  RLF NewSample+1,F
  RLF NewSample,F
  ENDM
```

There is not such a dramatic speedup in this case but the clear Carry – shift combination can be eliminated for a 1 instruction savings. These macros are used 8 times per loop in the main TWIST24 function. That 1 instruction savings does help.

After cleaning up the macros, the next interesting code section was the setup of the FSR in the TWIST24 function. Each time through the main loop there are 8 tests of bits to determine if the ADD_FSR_FILTER function should add to the new value or the old value.

The switch is performed by changing the address pointed to by the FSR. This requires a load FSR, test and reload FSR combination 8 times in the main function. A faster method is to conditionally execute an ADDFSR of 4. This will change the FSR to the next value in one instruction rather than 2. In this example, the comments have been removed to make the instructions more clear. The comments are left in the original code.

```
MOVLW   NewSample+3
MOVWF   FSR
MOVLW   OldFilter+3
BTFSC   FilterWeight,0
MOVWF   FSR
ADD_FSR_FILTER    ; add it
MUL_NEWOLD_BY2
```

```
MOVLW    NewSample+3
MOVWF    FSR0L
btfsc    FilterWeight,0
addfsr   .4
ADD_FSR0_FILTER
MUL_NEWOLD_BY2
```

This is another very minor change but it is executed 8*UPDATE_COUNT or 256 times. That is a significant savings.

At the end of the program there is one more code pattern that can be accelerated by the enhanced instruction set.

```
; add one to filter to have proper
; rounding
  MOVLW 0x01
  ADDWF NewFilter+2,F
  SKPNC
  ADDWF NewFilter+1,F
  SKPNC
  ADDWF NewFilter,F
```

```
; add one to filter to have proper
; rounding
 MOVLW   0x01
 ADDWF   NewFilter+2,F
 ADDWFC  NewFilter+1,F
 ADDWFC  NewFilter,F
```

This change eliminates a part of skip if Carry clear instructions that implement add with Carry. This becomes a simple sequence of add with Carry instructions.

The changes made to this program are very simple; yet, the effect is a function that runs nearly twice as fast allowing improved performance or a reduced clock speed and lower power consumption. The enhanced PIC12/16 features can result in very high performance gains for a rather modest effort in porting existing software.

The complete source to the modified function is presented in **Appendix D: “24-bit 256 Step IIR filter modified for Enhanced PIC12/16”**.

APPENDIX C: 24-BIT 256 STEP IIR FILTER (ORIGINAL CODE)

```
 ************************ TWIST24 ******************************
;
; Purpose of routine is to evaluate
;
;   A*X0 + B*Y
; X1 =  ---------
;       A + B
;
; Such as A+B = 2^n
;
; Simplified this formula will then become
;
;   A*X0 + ~A*Y + Y
; X1 =  ---------------
;      2^n
;
; Where:
; X1 - is new recursive lowpassed filter value
; X0 - is previous lowpassed filter value
; Y  - is new value
; A,B are weight factors, if A is large relative to B
; then more emphasis is placed on the average ( filtered ) value.
; If B is large then the latest sample are given more weight.
; ~A is A's bitwise complement
;
;
; X0,X1,Y are 24 bit variables
; A+B = 2^8 = 256 for this routine
;
; By Tony Kübek 2000-05-23, based on routine 'twist.asm' by
; Scott Dattalo ( BTW Thanks for sharing :-) )
;
; ************************************************************
;

; NewSample = Y ( 4 byte ram, 3 byte data )
; OldFilter = X0 ( 4 byte ram, 3 byte data )
; NewFilter = X1 ( 4 byte ram, 3 byte data )
; FilterWeight = A ( 1 byte ram, how much weight that is placed on old filtered value )
;
; * variables not really needed but used in example *
; FilterCounter = 1 byte, counter to increase step response.
; UpdateCounter = 1 byte, how many samples between 'global' updates
; AD_NewValue = 3 byte ram, last 24 bit reading from AD/or similar ( copied
; to NewSample, in case AD_NewValue should be used for other purposes )
; AD_DataVal = 3 byte ram, filtered value updated after UpdateCounter samples
; _AD_DataReady = 1 bit, set when AD_DataVal is updated
;
```

```
#define UPDATE_COUNT  0x20      ; 32 samples between global updates
#define FILTER_WEIGHT   0x80    ; filterweight, i.e. 'A' in example above

CBLOCK  0x020
  AD_NewValue:3
  NewSample:4               ; copy of AD_Newvalue, used locally
  OldFilter:4               ; previous value of filtering ( NewFilter from last run )
  NewFilter:4               ; the new filtered value
  FilterWeight:1            ; how much weight that should be posed on old value
  FilterCounter:1           ; to increase step response
  UpdateCounter:1           ; how often we want an global update
  AD_DataVal:3              ; the global update location
  Bitvars:1
  ENDC

#define _AD_DataReady BitVars,0

;++++++++++++
;
; ADD_FSR_FILTER - Adds 32 bit value pointed to by FSR to NewFilter
; FSR must point to LEAST significant byte, FSR-3 is most significant
;

ADD_FSR_FILTER MACRO
  ; 14-17 instructions

  ; add value pointed to by FSR to filter
      MOVF    INDF,W        ; get lowest byte
      ADDWF NewFilter+3,F   ; add to filter sum lowest byte

      DECF  FSR,F
      MOVF  INDF,W          ; get next byte
      SKPNC                 ;  if overflow
      INCFSZ INDF,W         ; increase source
      ADDWF NewFilter+2,F   ; and add to dest.
      DECF  FSR,F
      MOVF  INDF,W          ; get next byte
      SKPNC
      INCFSZ INDF,W
      ADDWF NewFilter+1,F

      DECF  FSR,F
      MOVF  INDF,W          ; get msb
      SKPNC
      INCFSZ INDF,W
      ADDWF NewFilter,F
      ENDM
;++++++++++++
;
; DIV_FILTER_BY2 - Divide NewFilter by 2
;
;
DIV_FILTER_BY2 MACRO
  ; 5 instructions
  ; right shift filter value by 1 ( i.e. divide by 2 )
```

```
  BCF STATUS,C                  ; clear carry
  ; divide by 2
  RRF NewFilter,F
  RRF NewFilter+1,F
  RRF NewFilter+2,F
  RRF NewFilter+3,F
  ENDM
;++++++++++++
;
; MUL_NEWOLD_BY2 - Multiply OldValue and NewSample with 2
;
;
MUL_NEWOLD_BY2 MACRO
  ; 10 instructions
  ; right shift filter value by 1 ( i.e. divide by 2 )
  BCF STATUS,C                  ; clear carry
  ; multiply old value with 2
  RLF OldFilter+3,F
  RLF OldFilter+2,F
  RLF OldFilter+1,F
  RLF OldFilter,F
  ; multiply new value with 2
  BCF STATUS,C                  ; clear carry
  RLF NewSample+3,F
  RLF NewSample+2,F
  RLF NewSample+1,F
  RLF NewSample,F
  ENDM

ORG 0x0000
  GOTO  INIT

; ** interupt routine for data collection
ORG 0x0004
INT
  ; get data from AD
  CALL  GET_AD_DATA             ; not included !

  ; for each sample, copy to AD_NewValue
  ; and call filter once
  CALL  TWIST24

  RETFIE


; cold start vector
INIT
  ; only 'dummy' code here
  BCF _AD_DataReady             ; clear data ready flag


  ; Note, this will inilialize the filter to gradually use
  ; no filtering to 'full' filtering ( increase once for each sample )
  ;
  MOVLW FILTER_WEIGHT
  MOVWF FilterCounter
```

```
  CLRF  FilterWeight

MAIN_LOOP
  ; wait for some data from AD or similar
  BTFSS _AD_DataReady       ; check if data available
  GOTO  MAIN_LOOP           ; nope

  ; filtered data available
  ; do whatever needs to be done..

  BCF _AD_DataReady         ; clear data ready flag

  GOTO  MAIN_LOOP

;++++++++++++
;
; TWIST24 - Variable 24 bit lowpass filter, calculates new lowpassed value in NewFilter,
; by weighing previous filtervalue and NewSample according to FilterWeight
; If FilterWeight is large more emphasis is placed on oldfiltered value
; Maximum value for FilterWeight is 255 ( i.e. 8 bit variable ).
; FilterWeight = 0 -> NewFilter = 0/256 of OldFilter + 256/256 of NewSample, i.e no filtering
; FilterWeight = 255 -> NewFilter = 255/256 of OldFilter + 1/256 of NewSample, i.e. full filtering
; NOTE: Previous filtered value should be kept in NewFilter as it is used for next pass.
;
TWIST24
  ; about 252-285 instructions executed ( with global update and copying of new datavalue )
  ; roufly 57 us at XTAL 20 Mhz

  ; ! uses FSR !

  ; Ramp function, uses an extra ram variable FilterCounter that
  ; increases the FilterWeight until itself zero
  ; Usage: Initialise to FilterWeight, then a speedier time to target will
  ; be accomplished (step response). During run, if for any reason an high ramp
  ; is detected ( or loss of readings ) this could be re-initialized to any
  ; value equal or less than FilterWeight to achive quicker step responce.
  ; NOTE : Filterweight must then ALSO be initialized so that the following
  ; is fulfilled: FilterCounter + FilterWeight = DesiredFilterWeight
  MOVF  FilterCounter,F
  BTFSC STATUS,Z
  GOTO  TWIST_GO

  DECFSZ  FilterCounter,F
  INCF  FilterWeight,F

TWIST_GO
  ; Copy previous filtered value ( note previous value is multiplied by 256
  ; i.e. only copy top three bytes of source to lowest three bytes of dest. )
  MOVF  NewFilter,W
  MOVWF OldFilter+1
  MOVF  NewFilter+1,W
  MOVWF OldFilter+2
  MOVF  NewFilter+2,W
  MOVWF OldFilter+3

  ; copy new value from AD to 'local' variable and add it it
```

```
; to filter as start value
MOVF  AD_NewValue,W          ; get top byte of new reading
MOVWF NewSample+1            ; store in local variable
MOVWF NewFilter+1            ; also add this as start value to new filter
MOVF  AD_NewValue+1,W        ;
MOVWF NewSample+2            ;
MOVWF NewFilter+2            ;
MOVF  AD_NewValue+2,W        ;
MOVWF NewSample+3            ;
MOVWF NewFilter+3            ;
CLRF  NewFilter              ;

CLRF  OldFilter              ; clear top bytes ( we only have a 24 bit filter )
CLRF  NewSample

MOVLW NewSample+3            ; get adress for new value
MOVWF FSR                    ; setup FSR
MOVLW OldFilter+3            ; get adress for old value to W

BTFSC FilterWeight,0         ; check if value that should be added is new or old
MOVWF FSR                    ; adress for old value already in W

ADD_FSR_FILTER               ; add it

MUL_NEWOLD_BY2               ; upshift old and new value, 10 instr.

MOVLW NewSample+3            ; get adress for new value
MOVWF FSR                    ; setup FSR
MOVLW OldFilter+3            ; get adress for old value to W

BTFSC FilterWeight,1         ; check if value that should be added is new or old
MOVWF FSR                    ; old value added to filter, adress in W
ADD_FSR_FILTER               ; add it

MUL_NEWOLD_BY2               ; upshift old and new value, 10 instr.

MOVLW NewSample+3            ; get adress for new value
MOVWF FSR                    ; setup FSR
MOVLW OldFilter+3            ; get adress for old value to W

BTFSC FilterWeight,2         ; check if value that should be added is new or old
MOVWF FSR                    ; old value added to filter, adress in W
ADD_FSR_FILTER               ; add it

MUL_NEWOLD_BY2               ; upshift old and new value, 10 instr.

MOVLW NewSample+3            ; get adress for new value
MOVWF FSR                    ; setup FSR
MOVLW OldFilter+3            ; get adress for old value to W

BTFSC FilterWeight,3         ; check if value that should be added is new or old
MOVWF FSR                    ; old value added to filter, adress in W
ADD_FSR_FILTER               ; add it

MUL_NEWOLD_BY2               ; upshift old and new value, 10 instr.
```

```
  MOVLW NewSample+3         ; get adress for new value
  MOVWF FSR                 ; setup FSR
  MOVLW OldFilter+3         ; get adress for old value to W

  BTFSC FilterWeight,4      ; check if value that should be added is new or old
  MOVWF FSR                 ; old value added to filter, adress in W
  ADD_FSR_FILTER            ; add it

  MUL_NEWOLD_BY2            ; upshift old and new value, 10 instr.

  MOVLW NewSample+3         ; get adress for new value
  MOVWF FSR                 ; setup FSR
  MOVLW OldFilter+3         ; get adress for old value to W

  BTFSC FilterWeight,5      ; check if value that should be added is new or old
  MOVWF FSR                 ; old value added to filter, adress in W
  ADD_FSR_FILTER            ; add it

  MUL_NEWOLD_BY2            ; upshift old and new value, 10 instr.

  MOVLW NewSample+3         ; get adress for new value
  MOVWF FSR                 ; setup FSR
  MOVLW OldFilter+3         ; get adress for old value to W

  BTFSC FilterWeight,6      ; check if value that should be added is new or old
  MOVWF FSR                 ; old value added to filter, adress in W
  ADD_FSR_FILTER            ; add it

  MUL_NEWOLD_BY2            ; upshift old and new value, 10 instr.

  MOVLW NewSample+3         ; get adress for new value
  MOVWF FSR                 ; setup FSR
  MOVLW OldFilter+3         ; get adress for old value to W

  BTFSC FilterWeight,7      ; check if value that should be added is new or old
  MOVWF FSR                 ; old value added to filter, adress in W
  ADD_FSR_FILTER            ; add it

  ; 235-268 instructions to get here

  ; check for rounding
  BTFSS NewFilter+3,7       ; test top bit of lowest byte
  GOTO  TWIST24_EXIT
  ; add one to filter to have proper rounding
  MOVLW 0x01
  ADDWF NewFilter+2,F
  SKPNC
  ADDWF NewFilter+1,F
  SKPNC
  ADDWF NewFilter,F
  ; 245 - 278 instructions to get here

TWIST24_EXIT
  ; check for update
  DECFSZ  UpdateCounter,F
  RETURN
```

```
; update global filter
MOVF  NewFilter+2,W
MOVWF AD_DataVal+2
MOVF  NewFilter+1,W
MOVWF AD_DataVal+1
MOVF  NewFilter,W
MOVWF AD_DataVal
; set data ready flag
BSF _AD_DataReady
; reinitialise update counter
MOVLW UPDATE_COUNT            ; number of samples between global update
MOVWF UpdateCounter

RETURN
```

APPENDIX D: 24-BIT 256 STEP IIR FILTER MODIFIED FOR ENHANCED PIC12/16

```
; ************************ TWIST24 ******************************
;
; Purpose of routine is to evaluate
;
;   A*X0 + B*Y
; X1 =  ---------
;       A + B
;
; Such as A+B = 2^n
;
; Simplified this formula will then become
;
;   A*X0 + ~A*Y + Y
; X1 =  ---------------
;      2^n
;
; Where:
; X1 - is new recursive lowpassed filter value
; X0 - is previous lowpassed filter value
; Y  - is new value
; A,B are weight factors, if A is large relative to B
; then more emphasis is placed on the average ( filtered ) value.
; If B is large then the latest sample are given more weight.
; ~A is A's bitwise complement
;
;
; X0,X1,Y are 24 bit variables
; A+B = 2^8 = 256 for this routine
;
; By Tony Kübek 2000-05-23, based on routine 'twist.asm' by
; Scott Dattalo ( BTW Thanks for sharing :-) )
;
; ***************************************************************
;

; NewSample = Y ( 4 byte ram, 3 byte data )
; OldFilter = X0 ( 4 byte ram, 3 byte data )
; NewFilter = X1 ( 4 byte ram, 3 byte data )
; FilterWeight = A ( 1 byte ram, how much weight that is placed on old filtered value )
;
; * variables not really needed but used in example *
; FilterCounter = 1 byte, counter to increase step responce.
; UpdateCounter = 1 byte, how many samples between 'global' updates
; AD_NewValue = 3 byte ram, last 24 bit reading from AD/or similar ( copied
;     to NewSample, in case AD_NewValue should be used for other purposes )
; AD_DataVal = 3 byte ram, filtered value updated after UpdateCounter samples
; _AD_DataReady = 1 bit, set when AD_DataVal is updated
;


#define UPDATE_COUNT  0x20          ; 32 samples between global updates
#define FILTER_WEIGHT   0x80        ; filterweight, i.e. 'A' in example above

CBLOCK  0x020
  AD_NewValue:3
```

```
  NewSample:4                   ; copy of AD_Newvalue, used locally
  OldFilter:4                   ; previous value of filtering ( NewFilter from last run )
  NewFilter:4                   ; the new filtered value
  FilterWeight:1                ; how much weight that should be posed on old value
  FilterCounter:1               ; to increase step responce
  UpdateCounter:1               ; how often we want an global update
  AD_DataVal:3                  ; the global update location
  Bitvars:1
  ENDC

#define _AD_DataReady BitVars,0

;++++++++++++
;
; ADD_FSRx_FILTER - Adds 32 bit value pointed to by FSR to NewFilter
; FSR must point to LEAST significant byte, FSR-3 is most significant
; two versions one for each FSR
;
ADD_FSR0_FILTER MACRO
  ; 8 instructions
  ; add value pointed to by FSR to filter
  moviw   0[INDF0]
  ADDWF   NewFilter+3,F         ; add to filter sum lowest byte
  moviw   -1[INDF0]             ; get next byte
  addwfc  NewFilter+2,F
  moviw   -2[INDF0]             ; get next byte
  addwfc  NewFilter+1,F
  moviw   -3[INDF0]             ; get msb
  addwfc  NewFilter,F
  ENDM

;++++++++++++
;
; DIV_FILTER_BY2 - Divide NewFilter by 2
;
;
DIV_FILTER_BY2 MACRO
  ; 4 instructions
  ; right shift filter value by 1 ( i.e. divide by 2 )
  ; divide by 2
  LSRF  NewFilter,F
  RRF NewFilter+1,F
  RRF NewFilter+2,F
  RRF NewFilter+3,F
  ENDM
;++++++++++++
;
; MUL_NEWOLD_BY2 - Multiply OldValue and NewSample with 2
;
;
MUL_NEWOLD_BY2 MACRO
  ; 8 instructions
  ; right shift filter value by 1 ( i.e. divide by 2 )
  ; multiply old value with 2
  LSLF  OldFilter+3,F
  RLF OldFilter+2,F
```

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.

Trademarks

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

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

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

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

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

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

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

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

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

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

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

ASIA/PACIFIC

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4080

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

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

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

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

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

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

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

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

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

Taiwan - Hsin Chu
Tel: 886-3-6578-300
Fax: 886-3-6578-370

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

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

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

EUROPE

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

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

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

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

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

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

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

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

03/26/09