# Section 4. Program Memory

## HIGHLIGHTS

This section of the manual contains the following topics:

## 4.1 PROGRAM MEMORY ADDRESS MAP

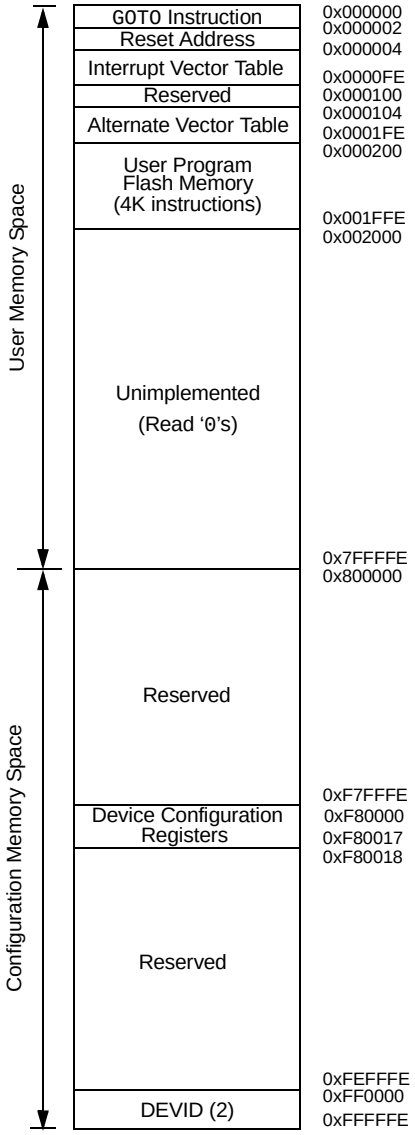
Figure 4-1 shows that the dsPIC33F devices have a 4M x 24-bit program memory address space. Three methods are available for accessing program space.

- Through the 23-bit (Program Counter) PC
- Through table read (TBLRD) and table write (TBLWT) instructions
- By mapping a 32-Kbyte segment of program memory into the data memory address space

The program memory map is divided into the user program space and the user configuration space. The user program space contains the Reset vector, interrupt vector tables, and program memory. The user configuration space contains nonvolatile configuration bits for setting device options and the device ID locations.

| | |
|---|---|
| **Note:** | If the RETURN instruction is placed at the end of the program memory, the Illegal Address Error Trap will be generated by the device during the run-time, which is due to the prefetch operation, which will try to preload the next two instructions from the memory location, which in this case, do not exist. The solution is to leave 2 extra instruction words available after the RETURN instruction so that the compiler can place NOP and RESET instructions at the end of the program memory. |

**Figure 4-1: Example Program Memory Map**




**Note:** The address boundaries for user Flash program memory will depend on the dsPIC33F device variant selected. For further details, refer to the appropriate device data sheet.

## 4.2 CONTROL REGISTER

**Register 4-1: CORCON: Core Control Register**

| U-0 | U-0 | U-0 | R/W-0 | R/W-0 | R-0 | R-0 | R-0 |
|---|---|---|---|---|---|---|---|
| — | — | — | US | EDT | DL<1:0> | | |
| bit 15 | | | | | | | bit 8 |

| R/W-0 | R/W-0 | R/W-1 | R/W-0 | R/C-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| SATA | SATB | SATDW | ACCSAT | IPL3 | PSV | RND | IF |
| bit 7 | | | | | | | bit 0 |

| **Legend:** | | | |
|---|---|---|---|
| R = Readable bit | W = Writable bit | U = Unimplemented bit, read as ‘0’ | |
| -n = Value at POR | ‘1’ = Bit is set | ‘0’ = Bit is cleared | x = Bit is unknown |

| | |
|---|---|
| bit 15-3 | **Not used by the Program Memory**<br>For full description of the CORCON bits, refer to **Section 2. “CPU”** (DS70204). |
| bit 2 | **PSV:** Program Space Visibility in Data Space Enable bit<br>1 = Program space visible in data space<br>0 = Program space not visible in data space |
| bit 1-0 | **Not used by the Program Memory**<br>For full description of the CORCON bits, refer to **Section 2. “CPU”** (DS70204). |

**Register 4-2: PSVPAG: PSV Page Register**

| U-0 | U-0 | U-0 | U-0 | U-0 | U-0 | U-0 | U-0 |
|---|---|---|---|---|---|---|---|
| — | — | — | — | — | — | — | — |
| bit 15 | | | | | | | bit 8 |

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| PSV Address Page bits | | | | | | | |
| bit 7 | | | | | | | bit 0 |

| **Legend:** | | | |
|---|---|---|---|
| R = Readable bit | W = Writable bit | U = Unimplemented bit, read as ‘0’ | |
| -n = Value at POR | ‘1’ = Bit is set | ‘0’ = Bit is cleared | x = Bit is unknown |

| | |
|---|---|
| bit 15-8 | **Unimplemented:** Read as ‘0’ |
| bit 7-0 | **PSV Address Page bits**<br>The 8-bit PSV Address Page bits are concatenated with the 15 Least Significant bits (LSb) of the W register, to form a 23-bit effective program memory address. |

**Register 4-3: TBLPAG: Table Page Register**

| U-0 | U-0 | U-0 | U-0 | U-0 | U-0 | U-0 | U-0 |
|---|---|---|---|---|---|---|---|
| — | — | — | — | — | — | — | — |
| bit 15 | | | | | | | bit 8 |

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| Table Address Page bits | | | | | | | |
| bit 7 | | | | | | | bit 0 |

| **Legend:** | | | |
|---|---|---|---|
| R = Readable bit | W = Writable bit | U = Unimplemented bit, read as ‘0’ | |
| -n = Value at POR | ‘1’ = Bit is set | ‘0’ = Bit is cleared | x = Bit is unknown |

bit 15-8 **Unimplemented:** Read as ‘0’

bit 7-0 **Table Address Page bits**

The 8-bit Table Address Page bits are concatenated with the W register, to form a 23-bit effective program memory address plus a byte select bit.

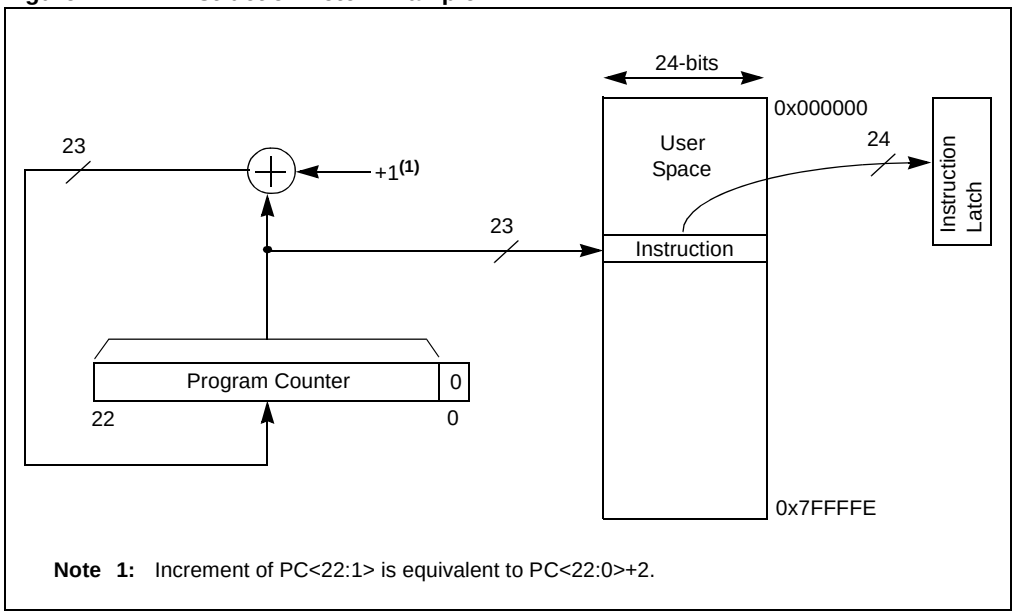
## 4.3 PROGRAM COUNTER

The Program Counter (PC) increments by two with the Least Significant bit (LSb) set to '0' to provide compatibility with data space addressing. Sequential instruction words are addressed in the 4M program memory space by PC<22:1>. Each instruction word is 24 bits wide.

The LSb of the program memory address (PC<0>) is reserved as a byte select bit for program memory accesses from data space that use Program Space Visibility or table instructions. For instruction fetches via the PC, the byte select bit is not required. Therefore, PC<0> is always set to '0'.

Figure 4-2 shows an instruction fetch example. Note that incrementing PC<22:1> by one is equivalent to adding 2 to PC<22:0>.

**Figure 4-2: Instruction Fetch Example**




**Note 1:** Increment of PC<22:1> is equivalent to PC<22:0>+2.

## 4.4 PROGRAM MEMORY ACCESS USING TABLE INSTRUCTIONS

The TBLRDL and TBLWTL instructions offer a direct method of reading or writing the least significant word (lsw) of any address within program space without going through data space, which is preferable for some applications. The TBLRDH and TBLWTH instructions are the only method whereby the upper 8 bits of a program word can be accessed as data.

### 4.4.1 Table Instruction Summary

A set of table instructions is provided to move byte- or word-sized data between program space and data space. The table read instructions are used to read from the program memory space into data memory space. The table write instructions allow data memory to be written to the program memory space.

| **Note:** | Detailed code examples using table instructions are found in **Section 5. "Flash Programming"** (DS70191). |
|---|---|

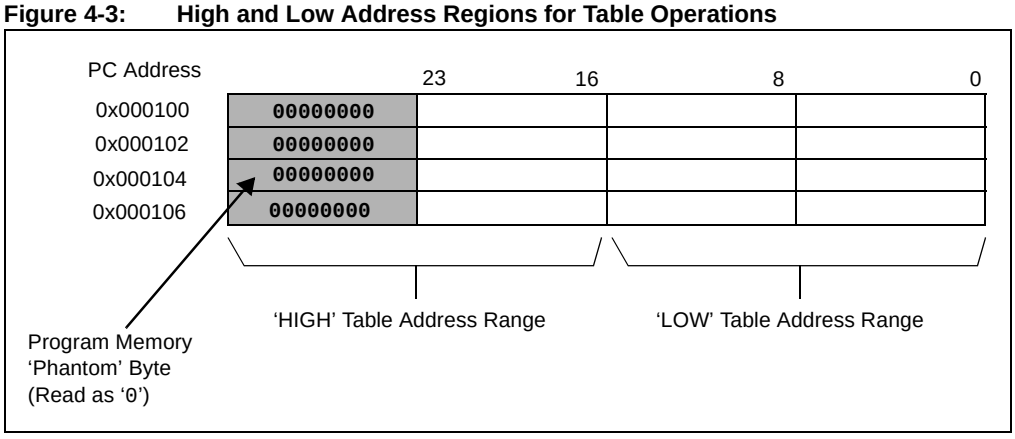
The four available table instructions are:

- TBLRDL: Table Read Low
- TBLWTL: Table Write Low
- TBLRDH: Table Read High
- TBLWTH: Table Write High

For table instructions, program memory can be regarded as two 16-bit, word-wide address spaces residing side by side, each with the same address range as shown in Figure 4-3. This allows program space to be accessed as byte or aligned word addressable, 16-bit wide, 64-Kbyte pages (i.e., same as data space).

TBLRDL and TBLWTL access the least significant data word of the program memory, and TBLRDH and TBLWTH access the upper word. As program memory is only 24 bits wide, the upper byte from this latter space does not exist, although it is addressable. It is, therefore, termed the "phantom" byte.

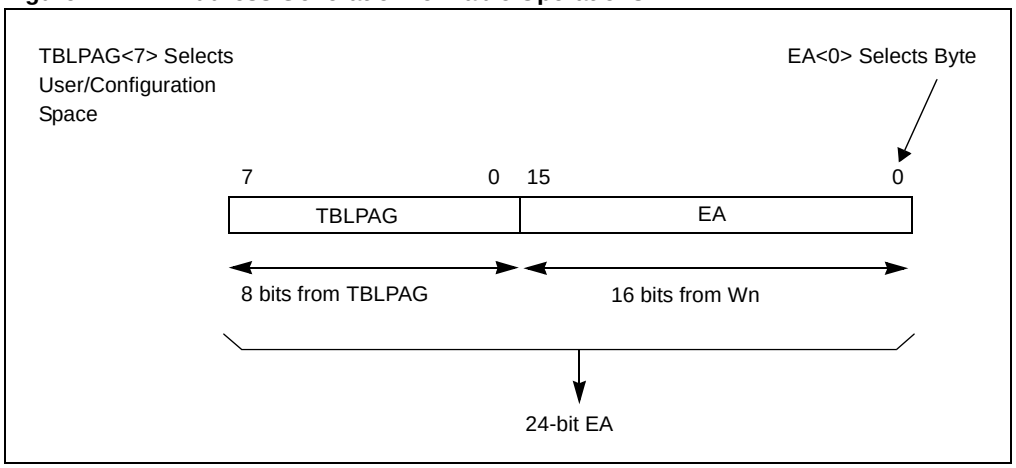
**Figure 4-3: High and Low Address Regions for Table Operations**



### 4.4.2 Table Address Generation

Figure 4-4 shows how for all table instructions, a W register address value is concatenated with the 8-bit Data Table Page (TBLPAG) register, to form a 23-bit effective program space address plus a byte select bit. As there are 15 bits of program space address provided from the W register, the data table page size in program memory is, therefore, 32K words.

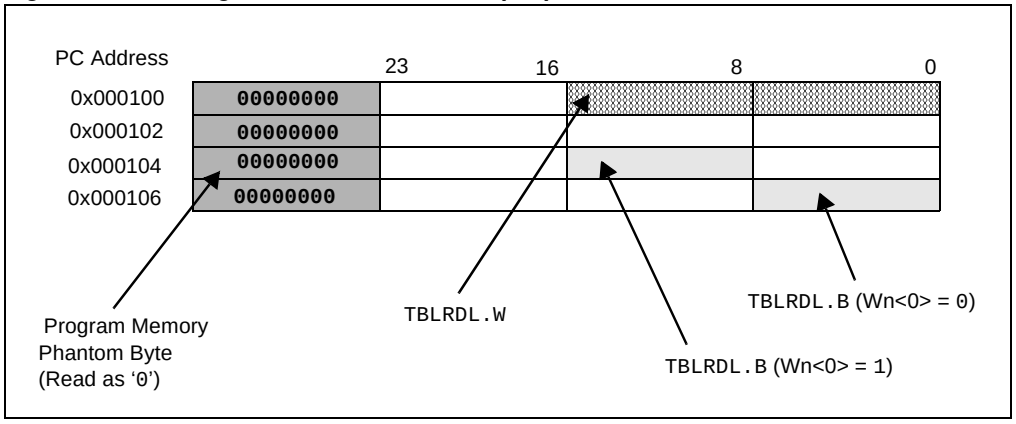
**Figure 4-4: Address Generation for Table Operations**




### 4.4.3 Program Memory Low Word Access

The TBLRDL and TBLWTL instructions are used to access the lower 16 bits of program memory data. The LSb of the W register address is ignored for word-wide table accesses. For byte-wide accesses, the LSb of the W register address determines which byte is read. Figure 4-5 demonstrates the program memory data regions accessed by the TBLRDL and TBLWTL instructions.

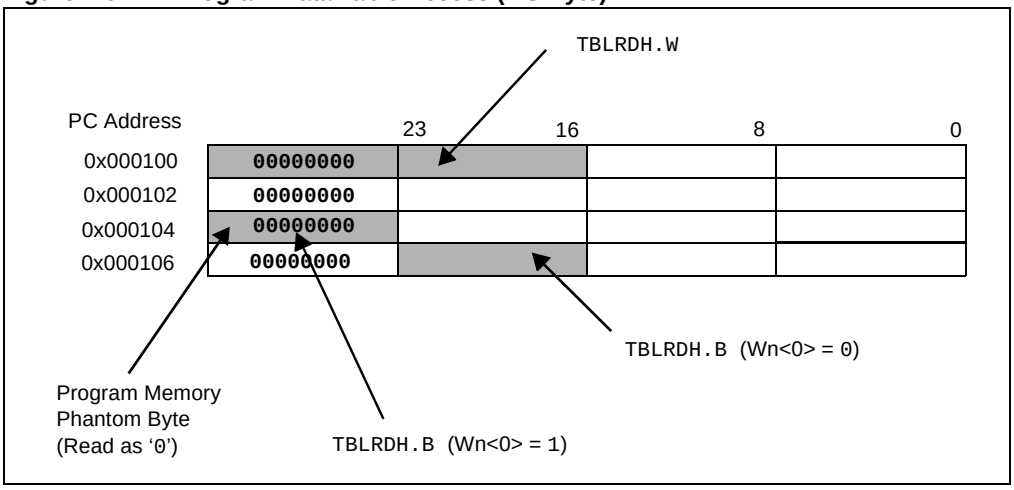
**Figure 4-5: Program Data Table Access (lsw)**



### 4.4.4 Program Memory High Word Access

The TBLRDH and TBLWTH instructions are used to access the upper 8 bits of the program memory data. Figure 4-6 shows how these instructions also support Word or Byte Access modes for orthogonality, but the high byte of the program memory data will always return '0'.

**Figure 4-6: Program Data Table Access (MS Byte)**




### 4.4.5 Data Storage in Program Memory

It is assumed that for most applications, the high byte (PC<23:16>) is not used for data, making the program memory appear 16 bits wide for data storage. It is recommended that the upper byte of program data be programmed either as a NOP instruction, or as an illegal opcode value, to protect the device from accidental execution of stored data. The TBLRDH and TBLWTH instructions are primarily provided for array program/verification purposes and for applications that require compressed data storage.

### 4.4.6 Program Memory Access Using Table Instructions Example

Example 4-1 uses table instructions to erase the program memory page starting at the address 0x12000, and programs the values 0x123456 and 0x789ABC into addresses 0x12000 and 0x12002, respectively.

| **Note:** | For more information on the unlocking sequence, refer to **Section 5. "Flash Programming"** (DS70191). |
|---|---|

**Example 4-1: Using Table Instructions to Access Program Memory**

```
#define PM_ROW_ERASE 0x4042
#define PM_ROW_WRITE 0x4001
#define CONFIG_WORD_WRITE0X4000

unsigned long Data;

/* Erase 512 instructions starting at address 0x12000 */
MemWriteLatch(0x1, 0x2000,0x0,0x0);
MemCommand(PM_ROW_ERASE);

/* Write 0x12345 into program address 0x12000 */
MemWriteLatch(0x1, 0x2000,0x0012,0x3456);
MemCommand(PM_ROW_WRITE);

/* Write 0x789ABC into program address 0x12002 */
MemWriteLatch(0x1, 0x2002,0x0078,0x9ABC);
MemCommand(PM_ROW_WRITE);

/* Read program addresses 0x12000 and 0x12002 */
Data = ReadLatch(0x1, 0x2000);
Data = ReadLatch(0x1, 0x2002);

;*******************************************************
;_MemWriteLatch:
;
;W0 = TBLPAG
;W1 = Wn
;W2 = WordHi
;W3 = WordLo
;no return values

_MemWriteLatch:
    mov   W0, TBLPAG
    tblwtl W3, [W1]
    tblwth W2, [W1]

    return

;*******************************************************
; _MemReadLatch:
;
;W0 = TBLPAG
;W1 = Wn
;return: data in W1:W0

_MemReadLatch:
    mov W0,TBLPAG
    tblrdl [W1],W0
    tblrdh [W1],W1

    return

;*******************************************************
; _MemCommand:
;
;W0 = NVMCON
;no return values

_WriteCommand:
    mov W0,NVMCON
    mov #0x55,W0;Unlock sequence
    mov W0,NVMKEY
    mov #0xAA,W0
    mov W0,NVMKEY
    bset  NVMCON,#WR
    nop            ;Required
    nop
Loop:btsc  NVMCON,#WR;Wait for write end
    bra   Loop

    return
```

Example 4-2 uses the `space(prog)` attribute to allocate the buffer in program memory. The MPLAB® C30 built-in functions, such as `builtin_tblpage` and `builtin_tbloffset`, can be used to access the buffer.

**Example 4-2: Using MPLAB® C30 Built-in Functions to Access Program Memory**

```
# include <p33fxxxx.h>

prog_data[10] __attribute__ ((space(prog))) = {0x0000, 0x1111, 0x2222,
                                0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888,
                                0x9999};

main (){
       unsigned int lowWord, highWord;
       unsigned int tbloffset;

       TBLPAG = __builtin_tblpage(&prog_data[3]);
       tbloffset = __builtin_tbloffset(&prog_data[3]);
       lowWord=__builtin_tblrd1(tbloffset);
       highWord=__builtin_tblrdh(tbloffset);

       /* do something */

}
```

## 4.5 PROGRAM SPACE VISIBILITY FROM DATA SPACE

The upper 32 Kbytes of the dsPIC33F data memory address space can optionally be mapped into any 16K word program space page. This mode of operation, called Program Space Visibility (PSV), provides transparent access of stored constant data from X data space without the need to use special instructions (i.e., TBLRD, TBLWT instructions).

### 4.5.1 PSV Configuration

Program Space Visibility is enabled by setting the PSV bit in the Core Control (CORCON<2>) register. A description of the CORCON register is found in **Section 2. "CPU"** (DS70204).

When PSV is enabled, each data space address in the upper half of the data memory map will map directly into a program address (see Figure 4-7). The PSV window allows access to the lower 16 bits of the 24-bit program word. The upper 8 bits of the program memory data should be programmed to force an illegal instruction, or a NOP instruction, to maintain machine robustness. Table instructions provide the only method of reading the upper 8 bits of each program memory word.

Figure 4-8 shows how the PSV address is generated. The 15 LSb of the PSV address are provided by the W register that contains the effective address. The Most Significant bit (MSb) of the W register is not used to form the address. Instead, the MSb specifies whether to perform a PSV access from program space or a normal access from data memory space. If a W register effective address of 0x8000 or greater is used, the data access will occur from program memory space when PSV is enabled. All data access occurs from data memory when the W register effective address is less than 0x8000.

Figure 4-8 shows how the remaining address bits are provided by the Program Space Visibility Page Address (PSVPAG<7:0>) register. The PSVPAG bits are concatenated with the 15 LSb of the W register, holding the effective address to form a 23-bit program memory address. PSV can only be used to access values in program memory space. Table instructions must be used to access values in the user configuration space.

The LSb of the W register value is used as a byte select bit, which allows instructions using PSV to operate in Byte or Word mode.

### 4.5.2 PSV Mapping with X and Y Data Spaces

The Y data space is located outside of the upper half of data space for most dsPIC33F variants, such that the PSV area will map into X data space. The X and Y mapping affect the way PSV is used in algorithms.

For example, the PSV mapping can be used to store coefficient data for Finite Impulse Response (FIR) filter algorithms. The FIR filter multiplies each value of a data buffer containing historical filter input data with elements of a data buffer that contains constant filter coefficients. The FIR algorithm is executed using the MAC instruction within a REPEAT loop. Each iteration of the MAC instruction prefetches one historical input value and one coefficient value to be multiplied in the next iteration. One of the prefetched values must be located in X data memory space and the other must be located in Y data memory space.

To satisfy the PSV mapping requirements for the FIR filter algorithm, the user application must locate the historical input data in the Y memory space, and the filter coefficients in X memory space.

**Figure 4-7: Program Space Visibility Operation**

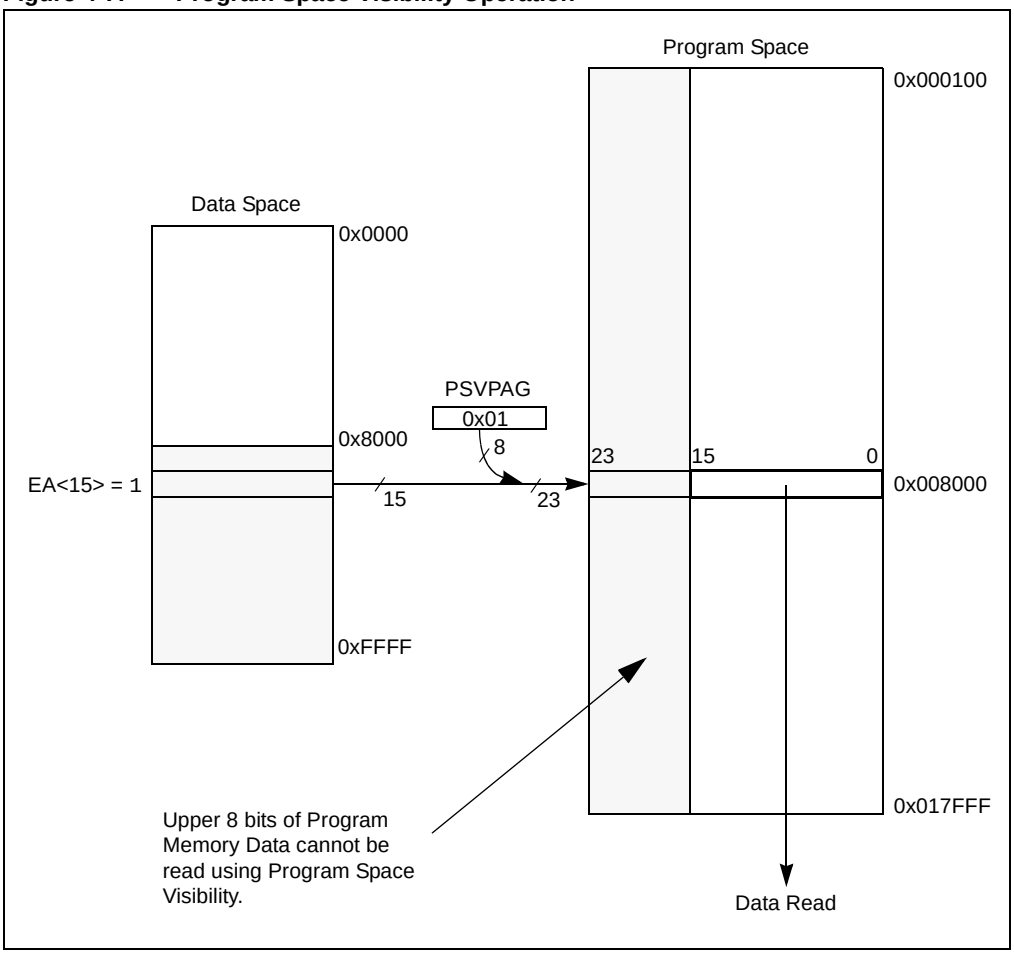
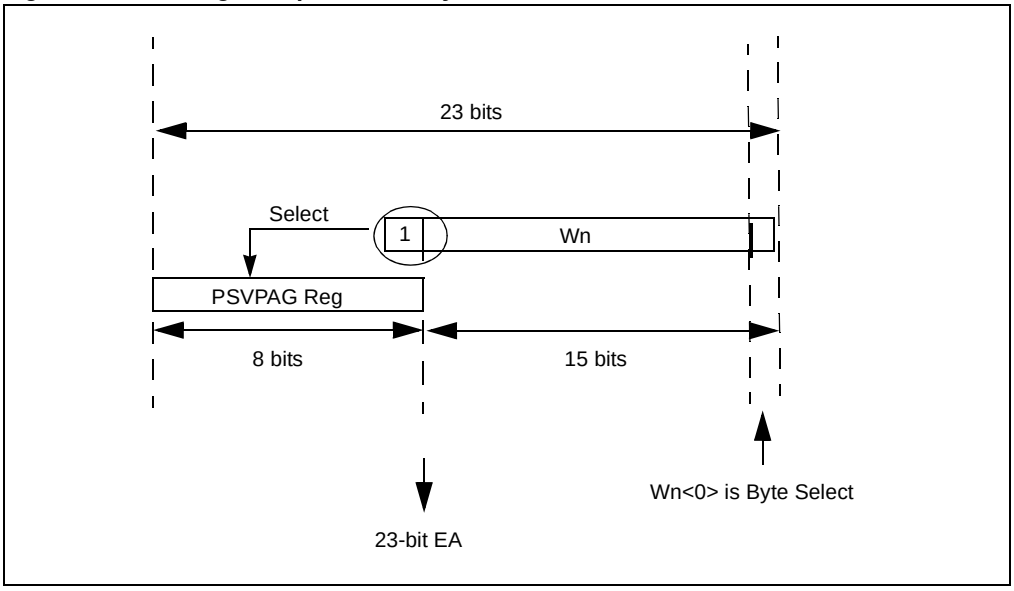



**Figure 4-8: Program Space Visibility Address Generation**



### 4.5.3 PSV Timing

Instructions that use PSV requires two extra instruction cycles to complete execution, except for the following instructions that require only one extra cycle to complete execution:

- The MAC class of instructions with data prefetch operands
- All MOV instructions including the MOV.D instruction

The additional instruction cycles are used to fetch the PSV data on the program memory bus.

#### 4.5.3.1 USING PSV IN A REPEAT LOOP

Instructions that use PSV within a REPEAT loop eliminate the extra instruction cycle(s) required for the data access from program memory, therefore incurring no overhead in execution time. However, the following iterations of the REPEAT loop incur an overhead of two instruction cycles to complete execution:

- The first iteration
- The last iteration
- Instruction execution prior to exiting the loop due to an interrupt
- Instruction execution upon re-entering the loop after an interrupt is serviced

#### 4.5.3.2 PSV AND INSTRUCTION STALLS

For more information about instruction stalls using PSV, refer to **Section 2. "CPU"** (DS70204).

### 4.5.4 PSV Code Examples

Example 4-3 illustrates how to create a buffer and access the buffer in the compiler managed, PSV section. The auto_psv space is the compiler managed PSV section. If the size of this section exceeds 32K, the linker will give an error. The tool chain will arrange for the PSVPAG to be correctly set at program start-up. By default, the compiler places all 'const' qualified variables into the auto_psv space.

**Example 4-3: Compiler Managed PSV Access**

```
#include "p33fxxxx.h"

int  m[5] __attribute__((space(auto_psv))) = { 1, 2, 3, 4, 5};
int  x[5]  = {10, 20, 30, 40, 50};
int sum;

main()
{
// Compiler Managed PSV
    sum=vectordot(m,x);
}


int vectordot(int *m, int *x)
{
    int i,sum=0;

    for(i=0;i<5;i++)
        sum+=*m++ * *x++;

    return(sum);
}
```

Example 4-4 illustrates buffer placement and access in the user managed PSV section. The psv space is the user managed PSV section.

**Example 4-4: User Managed PSV Access**

```
#include "p33fxxxx.h"

const int m[5] = { 1, 2, 3, 4, 5};
int  m1[5] __attribute__((space(psv))) = { 1, 2, 3, 4, 5};
int  m2[5] __attribute__((space(psv),address(0xA000))) = { 1, 2, 3, 4, 5};
int  x[5]  = {10, 20, 30, 40, 50};
int sum, sum1, sum2;

main()
{
int temp;

    // User Managed PSV
temp=PSVPAG; // Save auto_psv page

    PSVPAG = __builtin_psvpage(&m1);
    CORCONbits.PSV = 1;
    sum1=vectordot(m1,x);


    PSVPAG = __builtin_psvpage(&m2);
    sum2=vectordot(m2,x);

    PSVPAG=temp; // Restore auto_psv page


    // Compiler Managed PSV
    sum=vectordot(m,x);
}


int vectordot(int *m, int *x)
{
    int i,sum=0;

    for(i=0;i<5;i++)
        sum+=*m++ * *x++;

    return(sum);
}
```

Example 4-5 illustrates constant data placement in program memory and performs accessing of this data through the PSV data window using an assembly program.

**Example 4-5: PSV Code Example in Assembly**

```
        .include “p33fxxxx.inc”
        .section .const,psv
fib_data:
        .word 0, 1, 2, 3, 5, 8, 13

        ;Start of Code section
        .text

        .global __reset
__reset:
        ; Enable Program Space Visibility
        bset.b CORCONL, #PSV

        ; Set PSVPAG to the page that contains the fib_data array
        mov #psvpag(fib_data), w0
        mov w0, __PSVPAG

        ; Make a pointer to fib_data in the PSV data window
        mov #psvoffset(fib_data), w0

        ; Load the first data value
        mov [w0++], w1
```

## 4.6 REGISTER MAPS

A summary of the registers associated with Program Memory is provided in Table 4-1.

**Table 4-1: Program Memory Registers**

| SFR | Addr | Bit 15 | Bit 14 | Bit 13 | Bit 12 | Bit 11 | Bit 10 | Bit 9 | Bit 8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | All Resets |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TBLPAG | 0032 | — | — | — | — | — | — | — | — | Table Page Address Pointer Register | | | | | | | | 0000 |
| PSVPAG | 0034 | — | — | — | — | — | — | — | — | Program Memory Visibility Page Address Pointer Register | | | | | | | | 0000 |
| CORCON | 0044 | — | — | — | US | EDT | DL<2:0> | | | SATA | SATB | SATDW | ACCSAT | IPL3 | PSV | RND | IF | 0000 |

**Legend:** — = unimplemented, read as '0'. Shaded bits are not used in the operation of Program Memory.

## 4.7 PROGRAM MEMORY WRITES

The dsPIC33F family of devices contains internal program Flash memory for executing user code. There are two methods by which the user application can program this memory:

- Run-Time Self Programming (RTSP)
- In-Circuit Serial Programming™ (ICSP™)

RTSP is accomplished using TBLWT instructions. ICSP is accomplished using the SPI interface and integral bootloader software. For further details about RTSP, refer to **Section 5. "Flash Programming"** (DS70191). ICSP specifications can be downloaded from the Microchip Technology web site (www.microchip.com).

## 4.8 RELATED APPLICATION NOTES

This section lists application notes that are related to this section of the manual. These application notes may not be written specifically for the dsPIC33F device family, but the concepts are pertinent and could be used with modification and possible limitations. The current application notes related to the Program Memory module are:

| Title | Application Note # |
|---|---|
| No related application notes at this time. | |

**Note:** For additional Application Notes and code examples for the dsPIC33F family of devices, visit the Microchip web site (www.microchip.com).

## 4.9 REVISION HISTORY

### Revision A (March 2007)

This is the initial released revision of this document.

### Revision B (April 2007)

Minor updates made to document.

### Revision C (July 2008)

This revision incorporates the following updates:

- Rearranged the following sections: (see **4.2 “Control Register”**), (see **4.7 “Program Memory Writes”**)
- Registers:
  - PSVPAG: PSV Page Register (see Register 4-2)
  - TBLPAG: Table Page Register (see Register 4-3)
- Examples:
  - Using MPLAB C30 Built-in Functions to Access Program Memory (see Example 4-2)
  - Compiler Manager PSV Access (see Example 4-3)
  - User Manager PSV Access (see Example 4-4)
  - PSV Code Example in Assembly (see Example 4-5)
- Added Register Maps section (see **4.6 “Register Maps”**)
- Additional minor corrections such as language and formatting updates are incorporated in the entire document.