

Recycle Your Code

FEATURE ARTICLE

Stephen Bowling

Conversion and Optimization Techniques

Portable code and design reusability have become increasingly important as designers are given shorter design cycles and time-to-market goals. As Stephen explains, the PIC18Cxxx architecture lets you get more mileage out of your programming efforts.



Whether you are a seasoned software developer or a beginner, writing good portable code is essential for the reusability of the design. Design reusability has become increasingly important in reducing product design cycles, while reducing the end product's time to market is still essential. To demonstrate good programming practices that benefit design reusability, the upward migration path towards Microchip's newest generation of microcontrollers, the PIC18Cxxx family, will be examined here.

The PIC18Cxxx architecture was designed with these goals in mind:

- increased program efficiency
- increased program and data memory address spaces
- increased execution speed
- enhanced peripheral functions that maintain compatibility with other architectures
- source code level compatibility with existing PICmicro code

These features make the PIC18Cxxx family an excellent option for the PICmicro designer who has either run out of program or data memory space, needs faster program execution, or requires the functionality of the enhanced peripherals.

We'll look at two code-conversion examples using PIC16C7x assembly code. The first example will show the minimum number of steps required to make the source code compatible with the PIC18Cxxx devices. In the second example, I'll use the source code from the first code conversion and further modify it by taking advantage of some of the PIC18Cxxx architectural enhancements.

PIC18CXXX ARCHITECTURE

Readers familiar with the PICmicro architecture will find some differences in the operation of the new PIC18Cxxx family. So, before looking at any source code, let's briefly discuss some of the features of the enhanced architecture.

The PIC18Cxxx device family features a linear data memory map capable of addressing up to 4 KB. The data memory map is subdivided into 16 banks of 256 bytes. The upper 128 bytes of bank 15 (F80h–FFFh) hold the special function registers (SFR). The SFRs are data memory registers that are used to control the operation of the MCU and its peripherals.

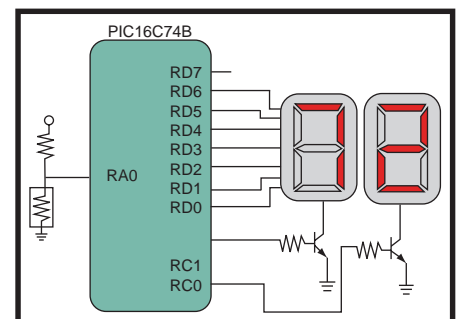


Figure 1—A digital thermometer makes a great code-conversion example. This is the simplified connection diagram of the thermometer.

The data memory may be addressed using a variety of direct or indirect methods. For direct addressing, the bank select register (BSR) is used. The BSR is a SFR that holds the upper four address bits of the data memory location to be accessed. The lower eight bits of the data memory address are encoded into the program instruction.

Direct addressing can also be performed without the BSR. The MOVFF instruction encodes two 12-bit data memory addresses into a two-word instruction and allows a move from any data memory location to any other location regardless of the BSR value.

Three file select registers (FSR) are available for accessing the data memory indirectly. Each FSR consists of two 8-bit registers (FSR#H and FSR#L), which hold a 12-bit address value. To use the FSR, load the desired FSR with the address of the variable using the LFSR instruction. When an operation is performed on the indirect file (INDF) register, the operation is performed on the data memory location pointed to by the FSR.

There are five INDF registers associated with each FSR that control how the FSR value is modified during the operation. These include the INDF, POSTINC, POSTDEC, PREINC, and PLUSW registers. An operation on the INDF register causes no change to the FSR. An operation on the POSTINC or POSTDEC registers will increment or decrement the FSR, respectively, after a data memory access is performed. An operation on the PREINC register will cause the FSR to be incremented before the data memory access. An operation on the PLUSW register uses the value that is presently stored in the working register (WREG) as a signed offset value for the data memory access.

To ensure that commonly used registers in a given application can be accessed in a single instruction cycle regardless of the current BSR value, a special 256-byte data memory region called the Access Bank has been implemented. The Access Bank consists of the first 128 locations in bank 0 (000h–07Fh) and the last 128 data

Listing 1—The following source code is the original digital thermometer application written to operate on a PIC16C74B.

```

; This program uses the A/D converter on the PIC16C74 to measure the
; temperature using a thermistor. The temperature value is displayed
; on two seven segment LEDs. Timer0 times the display updates
; and Timer1 times the A/D conversions.
list p = 16C74

include <p16C74.inc>
; 'Offset' is a correction value that is subtracted from the A/D
; result before finding the temperature in the lookup table.
#define Offset d'58'
; 'TblAddr' is the location of for our lookup tables in program memory.
#define TblAddr 0x800
Temp equ 0x20 ; Temporary storage
TempWREG equ 0x70 ; Context saving for ISR
TempSTATUS equ 0x71 ; Context saving for ISR
org 0x0000 ; Reset vector address.
nop
goto Start
org 0x0004 ; Interrupt vector address.
goto ISR

Start
bcf STATUS,RP1 ; Initialize bank bits
bcf STATUS,RP0 ; to point to bank 0.
movlw 0x20 ; Initialize FSR to first
movwf FSR ; GPR location in bank 0.

ClrRAM
clrf INDF ; Clear memory location.
incf FSR ; Goto next memory location.
btfss FSR,7 ; Are we at address 0x80?
goto ClrRAM ; No, keep clearing RAM.

clrf PORTC ; Set PORTC, PORTD to
movlw 0xff ; known output values.
movwf PORTD ;
bsf STATUS,RP0 ; bank 1
clrf TRISC ; Make PORTC all outputs
clrf TRISD ; Make PORTD all outputs
bcf STATUS,RP0 ; bank 0
bsf STATUS,RP0 ; bank 1
movlw b'00000100' ; Setup A/D for 3 analog
movwf ADCON1 ; channels.
bcf STATUS,RP0 ; bank 0
movlw b'01000001' ; Setup A/D clock for Fosc/8
movwf ADCON0 ; and turn A/D on.
clrf TMRO ; Clear Timer0.
bsf STATUS,RP0 ; bank 1
movlw b'11011111' ; Turn on Timer0.
movwf OPTION_REG ;
bcf STATUS,RP0 ; bank 0
bcf INTCON,TOIF ; Clear Timer0 interrupt flag.
bsf INTCON,TOIE ; Enable Timer0 interrupt.
clrf TMR1H ; Clear Timer1.
clrf TMR1L ;
movlw b'00110001' ; Turn on Timer1.
movwf TICON ;
bcf PIR1,TMR1IF ; Clear Timer1 interrupt flag.
bsf STATUS,RP0 ; bank 1
bsf PIE1,TMR1IE ; Enable Timer1 interrupt.
bcf STATUS,RP0 ; bank 0
bsf INTCON,PEIE ; Enable all peripheral interrupts.
bsf INTCON,GIE ; Enable global interrupts.
Main goto Main ; Loop forever in main program.

ISR
movwf TempWREG ; Save WREG
swaph STATUS,W ; Save STATUS
movwf TempSTATUS ;

btfsc PIR1,TMR1IF ; Is this a Timer1 interrupt?
goto DoConv ; Yes, go do an A/D conversion.
btfss INTCON,TOIF ; Is this a Timer0 interrupt?
goto Restore ; No, exit the ISR.

Display
bcf INTCON,TOIF ; Clear Timer0 interrupt flag.

movlw HIGH(TblAddr) ; Setup PCLATH for lookup table in
movwf PCLATH ; second page of memory.
bsf STATUS,C ;
movlw Offset ; Subtract offset value from

```

Listing 1—continued

```

subwf  ADRES,W          ; A/D result.
btfss  STATUS,C        ; If the result of the subtraction
clr    ; is negative, make the offset '0'.
call   TempTable       ; Call the table and get lookup
                        ; value in W.

bcf    PCLATH,3        ;
btfss  PORTC,1        ; Is one's digit currently displayed?
goto   Ones            ; No, go display one's digit.
                        ; Yes, do the ten's digit.
Tens   movwf Temp      ; Save temperature value in W.
movlw  HIGH(Tb1Addr)  ; Setup PCLATH for lookup table in
movwf  PCLATH         ; second page of memory.
swapf  Temp,W         ; Swap nibbles to get upper BCD digit.
andlw  0x0f           ; Mask BCD temperature value to 4 bits.
call   LEDTable       ; Get the appropriate 7-segment code.
clrf   PORTC          ; Turn off all digits.
bsf    PORTC,0        ; Turn on ten's digit.
movwf  PORTD          ; Write 7-segment code to PORTD.
movlw  HIGH(Restore)  ; Setup PCLATH
movwf  PCLATH         ;
goto   Restore        ; Exit the ISR.

Ones   movwf Temp      ; Save temperature value in W.
movlw  HIGH(Tb1Addr)  ; Setup PCLATH for lookup table in
movwf  PCLATH         ; second page of memory.
movf   Temp,W         ; Retrieve temperature value.
andlw  0x0f           ; Mask BCD temperature value to 4 bits.
call   LEDTable       ; Get the appropriate 7-segment code.
clrf   PORTC          ; Turn off all digits.
bsf    PORTC,1        ; Turn on ones's digit.
movwf  PORTD          ; Write 7-segment code to PORTD.
movlw  HIGH(Restore)  ; Setup PCLATH
movwf  PCLATH         ;
goto   Restore        ; Exit the ISR.

DoConv
bsf    ADCON0,G0      ; Start the A/D conversion.
bcf    PIR1,TMR1IF    ; Clear the interrupt flag while we're
btfss  ADCON0,G0      ; waiting and loop until the conversion
goto   $ - 1          ; is finished.

Restore
swapf  TempSTATUS,W  ; Restore STATUS
movwf  STATUS
swapf  TempWREG,F     ; Restore WREG w/o
swapf  TempWREG,W     ; affecting STATUS
retfie
org    Tb1Addr
; This lookup table takes the offset-adjusted A/D converter result and
; returns a temperature in degrees F. The 'dt' assembler directive
; produces a 'retlw' instruction for each data value listed.
TempTable
addwf  PCL,f
dt     0x32,0x32,0x32,0x32,0x33,0x33,0x34,0x34,0x35,0x35,0x35
dt     0x36,0x36,0x37,0x37,0x37,0x38,0x38,0x39,0x39,0x40,0x41
dt     0x41,0x42,0x43,0x43,0x44,0x44,0x45,0x45,0x46,0x46,0x47
dt     0x48,0x49,0x50,0x51,0x52,0x53,0x54,0x55,0x55,0x56,0x57
dt     0x58,0x59,0x59,0x60,0x61,0x61,0x62,0x63,0x63,0x64
dt     0x65,0x66,0x67,0x68,0x68,0x69,0x70,0x71,0x71,0x72,0x72
dt     0x73,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x80,0x81,0x81
dt     0x82,0x82,0x83,0x84,0x84,0x85,0x86,0x87,0x88,0x89,0x90
dt     0x91,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x99
dt     0x99
; This lookup table holds the codes to form digits on a seven segment
; LED display. These codes are for an active low LED driver scheme, so
; a '0' in a particular bit position turns a segment on.
LEDTable
addwf  PCL,f
retlw  b'11000000'    ; Zero
retlw  b'11111001'    ; One
retlw  b'10100100'    ; Two
retlw  b'10110000'    ; Three
retlw  b'10011001'    ; Four
retlw  b'10010010'    ; Five
retlw  b'10000010'    ; Six
retlw  b'11111000'    ; Seven
retlw  b'10000000'    ; Eight
retlw  b'10010000'    ; Nine

end ; End of the program!

```

memory locations (SFRs) in bank 15 (F80h–FFFh). Frequently used variables are located in 000h–07Fh. The use of the Access Bank is selected by a bit in the program instruction. The use of the Access Bank can be manually specified. However, it is best to let the assembler do it for you automatically based on the address of the variable.

The PIC18Cxxx device family has a program memory map, which extends up to 2 MB. The *CALL* and *GOTO* instructions in the architecture allow the entire program memory to be reached without the use of paging.

Program instructions consist of one or two 16-bit words. The program memory for the PIC18Cxxx device family is addressed in bytes to make the architecture more C compiler friendly. To ensure that program instruction words are always accessed with the proper byte alignment, the 21-bit program counter (PC) increments in steps of two with the least significant bit set to 0.

The program memory may be accessed on a byte-by-byte basis using table operations, which is useful for the storage of lookup data and calculating program memory check sums. Three registers in the SFR region (TBLPTRU, TBLPTRH, and TBLPTRL) implement a 21-bit table pointer, which points to the desired program memory address. An 8-bit transfer register (TABLAT) holds the value that is to be transferred to or from program memory. The user transfers the data, using table read (*TBLRD*) and table write (*TBLWT*) instructions. Depending on the syntax used, the *TBLWT* and *TBLRD* instructions can automatically increment or decrement the table pointer.

The PIC18Cxxx device family has an optional two-priority-level interrupt structure with the high-priority interrupt vector at 000008h and the low-priority vector at 000018h. A control bit is available that enables the priority-interrupt scheme, which is disabled by default. This allows interrupts to be compatible with source code written for devices in the PIC16Cxxx family. If priority interrupts are enabled, a high-priority in-

<i>Movlw</i>	<i>Offset</i>	<i>; Table offset value</i>
<i>Call</i>	<i>Table</i>	<i>; Call the table</i>
<i>Movwf</i>	<i>Result</i>	<i>; Get table result</i>
.		
.		<i>; other code</i>
.		
<i>addwf</i>	<i>PCL,F</i>	<i>; Add offset to PC</i>
<i>retlw</i>	<i>Constant2</i>	<i>; Table entry #2</i>
<i>retlw</i>	<i>Constant3</i>	<i>; Table entry #3</i>
<i>retlw</i>	<i>Constant4</i>	<i>; Table entry #4</i>
<i>retlw</i>	<i>Constant5</i>	<i>; Table entry #5</i>

Table 1—An example lookup table.

errupt source may override a low-priority interrupt that is in progress. SFRs are available that set the interrupt priority for each source.

A CODE EXAMPLE

One of the best ways to discuss code conversion and optimization is to use some real source code. The example program that I'll use was written to operate on a PIC16C74B device and implements a simple digital thermometer. It's not a complex program, but it demonstrates many of the code conversion issues that I'll discuss. Figure 1 shows a block diagram of the application. A resistor/thermistor combination is connected to channel 0 of the A/D converter. Two seven-segment LED displays are multiplexed to PORTD of the MCU. Timer0 and Timer1 are used to time the display updates and A/D conversions, respectively, that are handled in an interrupt service routine. The A/D result is used as an offset for a temperature lookup table. Each byte in the lookup table contains a packed BCD temperature result. The resulting BCD values are then used to obtain the proper LED segment data in a second lookup table.

The original code written for the PIC16C74B is given in Listing 1, which is available for downloading. Let me show you how to convert the code so that it will be compatible with a PIC18C452 device. The PIC18C452 is pin-to-pin compatible with the PIC16C74B and has a set of peripherals that are compatible with the PIC16C74B with enhancements.

Let's walk through the code and perform the minimal conversion. The first two lines in the code specify the processor type and the device defini-

tion file. The first thing you need to do is change these for the appropriate processor:

```
LIST    P = 18C452
include <p18c452.inc>
```

The device definition file specifies symbolic names for all of the registers and control bits associated with the device. It's always good practice to use register and bit definitions supplied by the vendor to avoid any naming convention issues.

Definitions for numerical constants and data storage variables are also included at the beginning of the source code. For example, the start address for the data lookup tables has been defined to be 800h. Constants like this one should be defined symbolically at the beginning of the code, because it enhances readability and allows them to be changed without having to sift through many lines of source code.

Next, you'll need to take care of any register name changes. To begin with, the PIC16Cxxx architecture has one FSR, which is eight bits wide. Because there are three 12-bit FSRs in the PIC18Cxxx architecture, the register names are different. For example, assuming you'll use FSR0, references to the FSR and INDF registers in the PIC16C74B code can be redefined as follows:

```
#define FSR    FSR0L
#define INDF   INDF0
```

The operation of Timer0 is enhanced in the PIC18Cxxx architecture so that it can be used as a 16- or 8-bit timer. Consequently, there are two registers (TMR0H and TMR0L) for Timer0. You can take care of this change with the following #define statement:

```
#define TMR0   TMR0L
```

The PIC18C452 A/D converter provides a 10-bit result in two 8-bit registers, ADRESH and ADRESL. The PIC16C74B has an 8-bit A/D converter and a single

result register, so you'll need the following #define:

```
#define ADRES  ADRESH
```

Finally, it is useful to define constants for the IRP, RPO, and RP1 bank selection bits. These bits are not implemented in the '18Cxxx architecture, nor are they defined in the device definition file.

```
#define RPO    5
#define RP1    6
#define IRP    7
```

For minimal conversion, leave all data memory banking statements in the converted code. Any instructions in the code that operate on the IRP, RPO, or RP1 bits have no effect because the bit locations are not implemented.

The interrupt vector origin needs to be changed at this point. Because byte addressing is used in the PIC18Cxxx architecture, the interrupt vector location is changed from 0004h to 000008h. You should also ensure that there is enough space for the program instructions that are located between the reset vector and interrupt vector. These instructions may need to be modified because *CALL* and *GOTO* instructions are one-word instructions in the PIC16Cxxx architecture but are implemented as two-word instructions in the PIC18Cxxx architecture.

Next, you need to verify that there are no bit location changes in the SFRs that control operation of the peripherals. For example, the values loaded into ADCON0 and ADCON1 should be verified using the device datasheet to ensure that the A/D

<i>Movlw</i>	<i>Offset</i>	<i>; Table offset value</i>
<i>Call</i>	<i>Table</i>	<i>; Call the table</i>
<i>Movwf</i>	<i>Result</i>	<i>; Get table result</i>
.		
.		<i>; other code</i>
.		
<i>addwf</i>	<i>PCL,F</i>	<i>; Add offset to PC</i>
<i>retlw</i>	<i>Constant2</i>	<i>; Table entry #2</i>
<i>retlw</i>	<i>Constant3</i>	<i>; Table entry #3</i>
<i>retlw</i>	<i>Constant4</i>	<i>; Table entry #4</i>
<i>retlw</i>	<i>Constant5</i>	<i>; Table entry #5</i>

Table 2—The lookup table has been modified to operate on the PIC18Cxxx architecture.

Listing 2—The following source code shows the minimal conversion steps required for the application to operate on a PIC18C452. The upper-case comments indicate changes made during the conversion.

```

; This program uses the A/D converter on the PIC16C74 to measure the
; temperature using a thermistor. The temperature value is displayed
; on two seven segment LEDs. Timer0 times the display updates
; and Timer1 times the A/D conversions.
list p = 18c452 ; CHANGE PROCESSOR

include <p18c452.inc> ; CHANGE INCLUDE FILE
; 'Offset' is a correction value that is subtracted from the A/D result
; before finding the temperature in the lookup table.
#define Offset d'58'
; 'TblAddr' is the location of for our lookup tables in program memory.
#define TblAddr 0x800
; HERE ARE SOME DEFINES THAT WE CAN USE TO MAKE THE CONVERSION EASIER
#define FSR FSROL ; WE HAVE 3 12-BIT FSR'S NOW!
#define INDF INDF0 ;
#define TMRO TMR0L ; TMRO IS A 16BIT TIMER NOW
#define RPO 5 ; BANK BITS NOT DEFINED IN
#define RP1 6 ; 18C452 INCLUDE FILE, SO
#define IRP 7 ; DEFINE THEM HERE.
#define ADRES ADRESH ; A/D CONVERTER IS NOW 10 BITS
#define RLF RLCF
#define RRF RRCF
Temp equ 0x20 ; Temporary storage
TempWREG equ 0x70 ; Context saving for ISR
TempSTATUS equ 0x71 ; Context saving for ISR
org 0x0000 ;
nop
goto Start
org 0x0008 ; CHANGE INTERRUPT VECTOR
goto ISR ; TO BYTE ADDRESS

Start
bcf STATUS,RP1 ; NOT REQUIRED
bcf STATUS,RPO ; NOT REQUIRED
movlw 0x20 ; FSR WAS REDEFINED AS FSROL
movwf FSR ; FOR COMPATIBILITY WITH
; 18C !!! CLRAM

clrf INDF ;
incf FSR ;
btfss FSR,7 ;
goto CLRAM ;
clrf PORTC ;
movlw 0xff ;
movwf PORTD ;
bsf STATUS,RPO ; NOT REQUIRED
clrf TRISC ;
clrf TRISD ;
bcf STATUS,RPO ; NOT REQUIRED
bsf STATUS,RPO ; NOT REQUIRED
movlw b'00000100' ;
movwf ADCON1 ; CHECK A/D BITS !!!!
bcf STATUS,RPO ; NOT REQUIRED
movlw b'01000001' ;
movwf ADCON0 ; CHECK A/D BITS !!!!
clrf TMRO ; TMRO REDEFINED AS TMR0L !!!!
movlw b'11011111' ;
bsf STATUS,RPO ; NOT REQUIRED
movwf TOCON ; CHANGE OPTION_REG TO TOCON !!!
bcf STATUS,RPO ; NOT REQUIRED
bcf INTCON,TOIF ;
bsf INTCON,TOIE ;
clrf TMR1H ;
clrf TMR1L ;
movlw b'00110001' ;
movwf T1CON ;
bcf PIR1,TMR1IF ;
bsf STATUS,RPO ; NOT REQUIRED
bsf PIE1,TMR1IE ;
bcf STATUS,RPO ; NOT REQUIRED
bsf INTCON,PEIE ;
bsf INTCON,GIE ;
Main goto Main ; Loop forever in main program.
ISR
movwf TempWREG ; NOT REQUIRED
swapf STATUS,W ; NOT REQUIRED
movwf TempSTATUS ; NOT REQUIRED
btfsc PIR1,TMR1IF ;
goto DoConv ;
btfss INTCON,TOIF ;
goto Restore ;

```

converter is configured properly. In most cases, the bit locations will be the same, but there may be differences between devices. You have to be careful here because the assembler will not catch these types of errors in the program. The only indication of a problem you will receive is when the peripheral controlled by the SFR does not operate as expected.

One special case is the SFR that controls Timer0. In the PIC16Cxxx architecture, Timer0 is configured by the OPTION_REG register. This register is not present in the PIC18Cxxx architecture, so you need to change the name to T0CON. T0CON now controls whether the timer is in 16- or 8-bit mode, therefore, you also need to ensure that the correct set-up value is loaded into the register.

In general, it is best not to use hard-coded bit and register values. For example, the two instructions below are equivalent, but the second is preferred:

```

bsf ADCON0, 0 ;
Don't do this !
bsf ADCON0, ADON ;
Preferred !

```

The code should be checked for any instructions that may be incompatible with the new architecture. When converting from the PIC16Cxxx to the PIC18Cxxx architecture, there are three cases that should be checked. The first instruction is CLRW, which clears the accumulator working register. This instruction is needed in the PIC16Cxxx device family because the working register is not physically addressable. However, the following #define statement makes a simple substitution:

```

#define CLRW CLRWF
WREG

```

The next two instructions that need modification are RLF and RRF, which rotate a register left or right, respectively, through the carry flag. The PIC18Cxxx instructions operate the same but are named differently to indicate usage of the carry flag. Again, two #define statements will make the required substitutions:

```
#define RRF RRCF
#define RLF RLCF
```

The final modifications to the code account for differences in the operation of the program counter. Remember, the program counter increments in steps of two in order to maintain word alignment with program instructions. Because of this, any references to the program counter value must be multiplied by two. In the original PIC16C74B source code, the following instructions poll the GO bit in ADCON0 to determine when an A/D conversion has completed:

```
btfs ADCON0,GO
; Conversion complete!
goto $ - 1
; No, keep checking.
```

The \$ symbol represents the present program counter value. The GOTO instruction decrements the program counter value so that the previous instruction is executed.

To operate correctly on the PIC18Cxxx architecture, the code should be modified to the following:

```
Btfs ADCON0,GO ;
Conversion complete!
goto $ - 2 ;
No, keep checking.
```

It is always good practice to use symbolic address labels. For example, the following code will operate the same on both device families without modification:

```
Test
btfs ADCON0,GO ;
Conversion complete!
goto Test ;
No, keep checking.
```

The lookup tables will require modification. To implement a lookup table, an offset value is placed in WREG and a CALL is made to the table. The table consists of a series of RETLW instructions, which return from the call with a specific value in WREG. The offset value in WREG is added to the program counter and causes a jump to the desired instruc-

Listing 2—continued.

```
Display
    bcf INTCON,T0IF ;
    movlw HIGH(TblAddr) ;
    movwf PCLATH ;
    bsf STATUS,C ;
    movlw Offset ;
    subwf ADRES,W ;
    btfs STATUS,C ;
    clrf WREG ; WREG IS NOW ADDRESSABLE,DON'T HAVE CLRW INSTRUCTION.
    call TempTable ;
    bcf PCLATH,3 ; NOT REQUIRED
    btfs PORTC,1 ;
    goto Ones ;
Tens
    movwf Temp ;
    movlw HIGH(TblAddr) ;
    movwf PCLATH ;
    swapf Temp,W ;
    andlw 0x0f ;
    call LEDTable ;
    clrf PORTC ;
    bsf PORTC,0 ;
    movwf PORTD ;
    movlw HIGH(Restore) ; NOT REQUIRED
    movwf PCLATH ; NOT REQUIRED
    goto Restore ;
Ones
    movwf Temp ;
    movlw HIGH(TblAddr) ;
    movwf PCLATH ;
    movf Temp,W ;
    andlw 0x0f ;
    call LEDTable ;
    clrf PORTC ;
    bsf PORTC,1 ;
    movwf PORTD ;
    movlw HIGH(Restore) ; NOT REQUIRED
    movwf PCLATH ; NOT REQUIRED
    goto Restore ;
DoConv
    bsf ADCON0,GO ;
    bcf PIR1,TMR1IF ;
    btfs ADCON0,GO ;
    goto $ - 2 ; NEED TO CHANGE THIS TO BYTE ADDRESS!
Restore
    swapf TempSTATUS,W ; NOT REQUIRED
    movwf STATUS ; NOT REQUIRED
    swapf TempWREG,F ; NOT REQUIRED
    swapf TempWREG,W ; NOT REQUIRED
    retfie
    org TblAddr
; This lookup table takes the offset-adjusted A/D converter result and
; returns a temperature in degrees F. The 'dt' assembler directive
; produces a 'retlw' instruction for each data value listed.
TempTable
    rlnf WREG ; MULTIPLY OFFSET BY 2!
    addwf PCL,f
    dt 0x32,0x32,0x32,0x32,0x33,0x33,0x34,0x34,0x35,0x35,0x35,0x35
    dt 0x36,0x36,0x37,0x37,0x37,0x38,0x38,0x39,0x39,0x40,0x41
    dt 0x41,0x42,0x43,0x43,0x44,0x44,0x45,0x45,0x46,0x46,0x47
    dt 0x48,0x49,0x50,0x51,0x52,0x53,0x54,0x55,0x55,0x56,0x57
    dt 0x58,0x59,0x59,0x60,0x61,0x61,0x62,0x63,0x63,0x64
    dt 0x65,0x66,0x67,0x68,0x68,0x69,0x70,0x71,0x71,0x72,0x72
    dt 0x73,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x80,0x81,0x81
    dt 0x82,0x82,0x83,0x84,0x84,0x85,0x86,0x87,0x88,0x89,0x90
    dt 0x91,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x99
; This lookup table holds the codes to form digits on a seven segment
; LED display. These codes are for an active low LED driver scheme, so
; a '0' in a particular bit position turns a segment on.
LEDTable
    rlnf WREG ; MULTIPLY OFFSET BY 2!
    addwf PCL,f
    retlw b'11000000' ; Zero
    retlw b'11111001' ; One
    retlw b'10100100' ; Two
    retlw b'10110000' ; Three
    retlw b'10011001' ; Four
    retlw b'10010010' ; Five
    retlw b'10000010' ; Six
    retlw b'11111000' ; Seven
    retlw b'10000000' ; Eight
    retlw b'10010000' ; Nine
    end ; End of the program!
```

Listing 3—The following listing shows the optimized source code for the PIC18C452. Comments in upper case indicate the optimizations.

```

; This program uses the A/D converter on the PIC16C74
; to measure the temperature using a thermistor. The
; temperature value is displayed on two seven segment
; LEDs. Timer0 times the display updates and Timer1
; times the A/D conversions.
list p = 18c452 ; CHANGE PROCESSOR

include <p18c452.inc> ; CHANGE INCLUDE FILE
; 'Offset' is a correction value that is subtracted
; from the A/D result before finding the temperature
; in the lookup table.
#define Offset d'58'
; 'TblAddr' is the location of for our lookup tables
; in program memory.
#define TblAddr 0x800
#define RAMTempTable 0x080
; HERE ARE SOME DEFINES THAT WE CAN USE TO MAKE THE
; CONVERSION EASIER
#define FSR FSR0L ; WE HAVE 3 12-BIT FSR'S NOW!
#define INDF INDF0 ; "
#define TMR0 TMR0L ; TMR0 IS A 16BIT TIMER NOW
#define RPO 5 ; BANK BITS NOT DEFINED IN
#define RP1 6 ; 18C452 INCLUDE FILE, SO
#define IRP 7 ; DEFINE THEM HERE.
#define ADRES ADRESH ; A/D CONVERTER IS NOW 10 BITS
Temp equ 0x20 ; Temporary storage

org 0x0000 ;
nop
bra Start ; GOTO CHANGED TO BRA

org 0x0008 ;
bra ISR ; GOTO CHANGED TO BRA

Start
lfsr 0,0 ; SET FSR0 TO 0
ClrRAM c1rf POSTINC0 ; CLEAR RAM AND INCR. FSR
movlw 0x06 ; HAVE WE CLEARED 1536 BYTES?
subwf FSR0H,W ;
bnz ClrRAM ; NO, KEEP GOING....

c1rf TBLPTRU ; INITIALIZE TABLE POINTER
movlw HIGH(TempTable) ; TO ACCESS TEMPERATURE DATA
movwf TBLPTRH ; IN PROGRAM MEMORY
movlw LOW(TempTable) ;
movwf TBLPTRL ;

; THE FOLLOWING CODE LOADS THE TEMPERATURE DATA FROM PROGRAM MEMORY
; INTO DATA MEMORY. WE'LL USE THE 'PLUSW' FEATURE OF INDIRECT
; ADDRESSING TO ACCESS THE DATA TABLE LIKE AN ARRAY.
lfsr 1,RAMTempTable ; LOAD FSR1 WITH THE ADDRESS
movlw d'100' ; FOR OUR DATA TABLE IN RAM.
movwf Temp ; SETUP TO MOVE 100 BYTES OF DATA.
RdTemp tblrd*+ ; GET DATA FROM PROG MEM, INC TABLPTR
movff TABLAT,POSTINC1 ; MOVE RETRIEVED VALUE TO DATA MEM
; AND INCREMENT FSR1.
decfsz Temp ; MOVED 100 BYTES YET?
bra RdTemp ; NO, KEEP GOING.....

lfsr 1,RAMTempTable ; RELOAD TABLE ADDRESS FOR FUTURE DATA
; ACCESSES.

c1rf PORTC ;
setf PORTD ; USE SETF COMMAND TO SET BITS.
c1rf TRISC ;
c1rf TRISD ;

movlw b'00000100' ;
movwf ADCON1 ;
movlw b'01000001' ;
movwf ADCON0 ;

c1rf TMR0L ;
movlw b'11011111' ;
movwf TOCON ; CHANGE OPTION_REG TO TOCON !!!
bcf INTCON,TOIF ;
bsf INTCON,TOIE ;

c1rf TMR1H ;

```

tion. After returning from the call to the lookup table, the WREG value is stored in the desired location. Table 1 is an example lookup table .

For the lookup table to function properly on the PIC18Cxxx architecture, the offset value must be multiplied by two. This will cause the program counter to be advanced by the correct number of instructions. The lookup table has been modified to operate on the PIC18Cxxx architecture (Table 2).

The minimal modifications to your PIC16C74B source code have now been completed. The code should now be fully functional on the PIC18C452 device. The complete minimal code conversion is provided in Listing 2.

OPTIMIZING THE CODE CONVERSION

There are still a few modifications that can be done to improve the efficiency of the code. Let's take a look at what can be done. For reference, Listing 3 shows the optimized code.

Consider the variable locations. For the minimal code conversion, you left the three variables in their original locations. There is no advantage to moving these variables because they are all located in the Access Bank region (0–7Fh). In fact, no relocation of variables should be required for an application that uses 256 bytes or less of data memory. In this case, all variables will be located in bank 0 (0–FFh), and no data memory banking will be required. For applications that require more than 256 bytes of data memory, there are benefits to relocating the variables. More frequently used variables should be located in the Access Bank so that they can be accessed in a single instruction cycle.

Certain program instructions can be changed or deleted to decrease program memory usage. All data memory banking instructions that modify the IRP, RPO, and RP1 bits can be deleted. The *CALL* and *GOTO* instructions use two program words in the PIC18Cxxx architecture so that the entire 2-MB address range can be encoded into the instruction. These instructions write to the program counter directly, so you can also delete instructions that write to

PCLATH before *CALL* and *GOTO* instructions. Certain *CALL* and *GOTO* instructions can be further optimized to reduce program memory usage. The *RCALL* (relative call) and *BRA* (unconditional branch) instructions have been implemented in the PIC18Cxxx architecture. These one-word instructions may be used when the destination program counter location is less than 2048 bytes away from the present address.

It should be noted that PCLATH still needs to be set up when using a data lookup table. When the low byte of the program counter (PCL) is the destination for an instruction, the values in PCLATH and PCLATU will be transferred into the program counter.

Lookup tables that contain large amounts of data can be modified to decrease program memory usage significantly. When the data is encoded using *RETLW* instructions, two bytes of program memory are used to store each byte of data. In the PIC18Cxxx architecture, the lookup data may be accessed using table read operations. Because there are 100 bytes of data, it would be beneficial to convert the temperature lookup data in my example application.

Because the data memory is available, you can load the temperature lookup data into data memory at powerup. At the beginning of the program, *TBLPTRU*, *TBLPTRH*, and *TBLPTRL* are initialized to the location of the temperature data table in program memory. You also load *FSR1* with the location for the table in data memory. The data is transferred with minimal code using the automatic post-incrementing modes for the *TBLRD* instructions and *FSRs*. Listing 4 shows how this is accomplished.

Now that the lookup data has been transferred to data memory, it may be easily accessed using the *PLUSW* addressing mode of the *FSRs* as shown:

```
lfsr      1,RAMTempTable
; load FSR1 with the address
```

```
; for our data table in RAM.
Movlw   Offset
; put table offset in WREG
```

Listing 3—continued.

```

    clrf   TMR1L      ;
    movlw  b'00110001' ;
    movwf  T1CON      ;
    bcf    PIR1,TMRIIF ;
    bsf    PIE1,TMRIE  ;

    bsf    INTCON,PEIE ;
    bsf    INTCON,GIE  ;

Main   bra    Main      ; CHANGE GOTO TO BRANCH.

ISR    btfsc  PIR1,TMRIIF ;
       bra    DoConv     ; CHANGE GOTO TO BRANCH.

       btfss  INTCON,TOIF ;
       retfie FAST      ; USE SHADOW REGISTER VALUES.

Display
    bcf    INTCON,TOIF ;
    bsf    STATUS,C ;
    movlw  Offset      ;
    subwf  ADRES,W      ;
    btfss  STATUS,C ;
    clrf   WREG         ;

       btfss  PORTC,1 ;
       bra    Ones      ; CHANGE GOTO TO BRA.
Tens   swapf  PLUSW1,W  ; USE OFFSET IN WREG AND FSR1 TO GET
       andlw  0x0f      ; TEMPERATURE VALUE IN DATA MEMORY ARRAY.
       rcall  LEDTable  ; CHANGE CALL TO RCALL.
       clrf   PORTC    ;
       bsf    PORTC,0  ;
       movwf  PORTD    ;
       retfie FAST     ; USE SHADOW REGISTER VALUES.
Ones   movf   PLUSW1,W  ; USE OFFSET IN WREG AND FSR1 TO GET
       andlw  0x0f      ; TEMPERATURE VALUE IN DATA MEMORY ARRAY.
       rcall  LEDTable  ; CHANGE CALL TO RCALL.
       clrf   PORTC    ;
       bsf    PORTC,1  ;
       movwf  PORTD    ;
       retfie FAST     ; USE SHADOW REGISTER VALUES.

DoConv
    bsf    ADCON0,GO    ;
    bcf    PIR1,TMRIIF ;
    btfsc  ADCON0,GO    ;
    goto   $ - 2        ;
    retfie FAST        ; USE SHADOW REGISTER VALUES.
    org    TblAddr
; THE TEMPERATURE TABLE HAS BEEN MODIFIED TO PLACE A TEMPERATURE
; VALUE IN EVERY BYTE LOCATION. THE 'DW' ASSEMBLER DIRECTIVE PUTS
; THE DATA IN PROGRAM MEMORY TWO BYTES AT A TIME.
TempTable
    Dw     0x3232,0x3232,0x3333,0x3434,0x3535,0x3635
    dw     0x3736,0x3737,0x3838,0x3939,0x4140,0x4241
    dw     0x4343,0x4444,0x4545,0x4646,0x4847,0x5049
    dw     0x5251,0x5453,0x5555,0x5756,0x5958,0x6059
    dw     0x6161,0x6362,0x6363,0x6564,0x6766,0x6868
    dw     0x7069,0x7171,0x7272,0x7373,0x7574,0x7776
    dw     0x7978,0x8180,0x8281,0x8382,0x8484,0x8685
    dw     0x8887,0x9089,0x9191,0x9392,0x9594,0x9796
    dw     0x9998,0x9999

LEDTable
    movwf  Temp      ;
    movlw  HIGH(LEDTable) ; WE STILL NEED TO SETUP PCLATH SINCE
    movwf  PCLATH   ; WE ARE OPERATING DIRECTLY ON PCL.
    movf   Temp,W    ;
    rlnsf  WREG      ;
    addwf  PCL,F     ;
    dt     b'11000000' ; Zero
    dt     b'11111001' ; One
    dt     b'10100100' ; Two
    dt     b'10110000' ; Three
    dt     b'10011001' ; Four
    dt     b'10010010' ; Five
    dt     b'10000010' ; Six
    dt     b'11111000' ; Seven
    dt     b'10000000' ; Eight
    dt     b'10010000' ; Nine

    end              ; End of the program!

```

```
movff    PLUSW1,Result
; move data table value to result
register
```

The automatic post-increment mode of the FSRs has also been used in the optimized code to clear the data memory before program execution ends. The following code segment clears the data memory in the PIC18C452:

```
lfsr      0,0          ; set
fsr0 to 0
ClrRAM   clrf  POSTINC0 ;
clear location and incr. FSR
movlw    0x06         ;
have we cleared 1536 bytes?
subwf    FSR0H,W
bzn      ClrRAM       ; no,
keep going
```

There is one final optimization that you can make to the source code. The interrupt service routine in the PIC16C74B code saves WREG and STATUS in temporary registers. This is not required for this application because the main program loop does not perform any function. However, most applications require context saving, so I included it in the source code. You can remove the context-saving instructions in the code because the PIC18Cxxx devices have three shadow registers for the WREG, STATUS, and BSR registers. When an interrupt occurs, these registers are automatically saved in the shadow registers. To restore the saved regis-

ters at the end of an interrupt service routine, a "fast return from interrupt" is performed as follows:

```
RETFIE   FAST          ; restore
values from shadow registers
```

One thing to remember about the shadow registers is that they are only one level deep. If priority interrupts are enabled and a high-priority interrupt occurs during a low-priority interrupt, the shadow register values will be overwritten. So, if your application uses both high- and low-priority interrupts, be sure that the shadow registers are used only for the high-priority interrupt service routine.

CONCLUSION

Now, you've seen that migrating an existing PICmicro software design to the PIC18Cxxx device family is a relatively simple process. The PIC18Cxxx device family has many features that enhance program efficiency, while maintaining source code and peripheral compatibility with other device families. The thermometer application, as written, consumes 205×14 -bit words of program memory on the PIC16Cxxx architecture. The converted and optimized code uses 145×16 -bit words of program memory on the PIC18Cxxx architecture, providing a 19% reduction in program memory usage. ▣

Stephen Bowling has been an applications engineer at Microchip Technol-

ogy since 1998. He specializes in the PIC16Cxxx and PIC18Cxxx device families. Stephen may be reached via e-mail at stephen.bowling@microchip.com.

REFERENCES

- Microchip Technology, Inc., PIC18Cxx2 Datasheet, DS39026B, Microchip Technology, Inc.
- Microchip Technology, Inc., PIC16C63A/65B/73B/74B Datasheet, DS30605B, Microchip Technology Inc.
- Microchip Technology, Inc., "Migrating Designs from PIC16C74A/74B to PIC18C442," Application note AN716, Microchip Technology Inc.

SOURCE

PIC18Cxxx and PIC16Cxxx microcontrollers

Microchip Technology, Inc.
 (888) 628-6247
 (480) 786-7200
 Fax: (480) 899-9210
www.microchip.com

Listing 4—This is how data is transferred using the automatic post-incrementing modes for the TBLRD instructions and FSRs.

```
clrf    TBLPTRU    ; INITIALIZE TABLE POINTER
movlw   HIGH(TempTable) ; TO ACCESS TEMPERATURE DATA
movwf   TBLPTRH    ; IN PROGRAM

MEMORY
movlw   LOW(TempTable) ;
movwf   TBLPTL     ;
lfsr    1,RAMTempTable ; LOAD FSR1 WITH THE ADDRESS
movlw   d'100'      ; FOR OUR DATA TABLE IN RAM.
movwf   Count       ; SETUP TO MOVE 100 BYTES OF

DATA.
RdTemp
tblrd*+ ; GET DATA FROM PROG MEMORY
; AND INCREMENT TABLPTR.
movff   TABLAT,POSTINC1 ; MOVE RETRIEVED VALUE TO DATA MEM
; AND INCREMENT FSR1.
decfsz Count          ; MOVED 100 BYTES YET?
Bra     RdTemp         ; NO, KEEP GOING.....
```

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitscellar.com or www.circuitcellar.com/subscribe.htm.