Direct & Indirect Addressing

Introduction. This application note describes direct and indirect addressing and shows a method for avoiding the gaps in the PIC16C57's banked memory.

Direct Addressing. PIC microcontrollers have 32 or 80 bytes of RAM, which Microchip refers to as file registers. The first seven registers (eight in the case of the 28-pin PIC's) are mapped to special functions, such as the real-time clock/counter (RTCC), status bits, and input/output (I/O) ports. Figure 1 shows a simplified memory map.

The simplest way to manipulate the contents of a PIC's RAM is to specify a register address in an instruction, like so:

```
mov   010h, #100
```

This instruction, which moves the decimal number 100 into register 10 hexadecimal (h), is an example of *direct addressing*. Most of the instructions in a typical PIC program use this addressing mode.

The TechTools assembler has several helpful features for direct ad-

Address (hex)		Description	
0	indirect	Reads/writes address pointed to by **fsr**.	
1	rtcc	Real-time clock/counter.	
2	pc	Program counterÑ9 to 1 1 bits wide; lower 8 may be read/written.	
3	status	Flag bits for arithmetic operations, sleep and reset, and ROM page selects.	
4	fsr	Pointer; address of data accessed through **indirect**.	
5	ra	I/O port **ra**.	
6	rb	I/O port **rb**.	
7	rc	I/O port **rc** on Õ55 and Õ57; general-purpose register on Õ54 and Õ56.	
8Ð1F		**General-purpose RAM ('54, '55, '56) and RAM bank 0 of '57**	
20Ð2F		RAM warp; reads and writes to 0ÐF.	**RAM Banks 1–3, 16C57**
30Ð3F		RAM bank 1 (Õ57 only)	
40Ð4F		RAM warp; reads and writes to 0ÐF.	
50Ð5F		RAM bank 2 (Õ57 only)	
60Ð6F		RAM warp; reads and writes to 0ÐF.	
70Ð7F		RAM bank 3 (Õ57 only)	

Figure 1. Simplified memory map of PIC's 16C54 through '57.

dressing. The first of these is labeling. Instead of referring to registers by address, you may assign names to them:

```
counter     =       010h            ;Program  header
            .
            .
            .
            mov     counter, #100
```

Labeled memory locations are often called variables. They make a program more understandable and easier to modify. Suppose you needed to change the location in which the counter data was stored. Without the label, you would have to rely on your text editor's search-and-replace function (which might also change other numbers containing "10"). With a label, you could change the *counter = ...* value in the program header only.

You can also define variables without specifying their address by using the *ds* (define space) directive:

```
            org     8               ;Start above special registers.
counter     ds      1               ;One byte labeled "counter."
            .
            .
            .
            mov     counter, #100
```

Using *ds* assigns the label to the next available register. This ensures that no two labels apply to the same register, making variable assignments more portable from one program to another. The only caution in using *ds* is that you must set the origin using the *org* directive twice; once for the starting point of variables in RAM, and again (usually at 0) for the starting point of your program in ROM.

Labels can be assigned to individual bits in two ways. First, if the bit belongs to a labeled byte, add *.x* to the label, where *x* is the bit number (0–7). Or assign the bit its own label:

```
LED         =       ra.3            ;Bit 3 of port ra controls LED.
```

The TechTools assembler has predefined labels for the special-purpose registers, and the bits of the *status* register. See your manual for a list.

Indirect Addressing. The registers used in direct addressing are set forever when the program is burned into the PIC's ROM. They cannot change. However, many powerful programming techniques are based on computing storage locations. Consider a keyboard buffer. If key-strokes can't be processed immediately, they are stored in sequential bytes of memory. *Pointers*—variables containing addresses of other variables—track the locations of data entered and data processed.

The PIC's *indirect addressing* mode allows the use of pointers and the high-level data structures that go with them, such as stacks and queues. Using indirect addressing for the earlier example (writing 100 to register 10h) would look like this:

```
mov     fsr, #010h      ;Set pointer to 10h.
mov     indirect, #100  ;Store 100 to indirect.
```

The value in the file select register (*fsr*; register 04h) is used as the address in any instruction that reads/writes *indirect* (register 00h). So

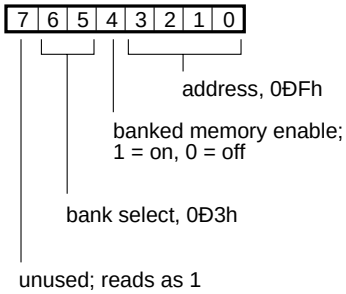

Figure 2. The 16C57 file-select register.

Memory Register banks:		
	0	10h to 1Fh
	1	30h to 3Fh
	2	50h to 5Fh
	3	70h to 7Fh

storing 10h in the *fsr* and then writing 100 to *indirect* is the same as writing 100 to address 10h.

A more practical example would be to store a series of values from an I/O port to sequential registers in memory. All it takes is a loop like this:

```
        mov   pointer, #010h  ;Set start address.
:loop   mov   fsr, pointer    ;Put pointer into fsr.
        mov   indirect, rb    ;Move rb to indirect.
        inc   pointer         ;pointer = pointer + 1.
        cjb   pointer,#01Fh,:loop
```

This fragment assumes that a variable named *pointer* was declared previously (using =, *equ*, or *ds*), and that *rb* is set for input. The loop will rapidly fill registers 10h through 1Fh with data samples from *rb*.

PIC's with 32 bytes of RAM ('54, '55, and '56) have a five-bit-wide *fsr*. Since all registers are eight bits wide, the highest three bits of the *fsr* in these devices are fixed, and always read as 1's. Keep this in mind if you plan to perform comparisons (such as the last line of the example above) directly on the *fsr*. It will always read 224 (11100000b) higher than the actual address it points to.

The 16C57 has 80 bytes of RAM and a seven-bit-wide *fsr*. The highest bit of its *fsr* is fixed and reads as a 1. Seven bits allows for 128 addresses, but only 80 are used. The remaining 48 addresses are accounted for by three 16-byte gaps in the 57's memory map. See the RAM warps in figure 1.

Because these warps map to the lowest file registers of the PIC, they can cause real trouble by altering data in the special-purpose registers. To avoid this problem, consider using a subroutine to straighten out the memory map and avoid the warps. Below is an excerpt from a program that uses the registers from 10h on up as a storage buffer for up to 64 characters of ASCII text. For the purposes of the program, address10h is location 0 in the buffer; 7F is location 63.

When the program needs to write a value representing a position in the buffer to the *fsr*, it puts the value into the *w* register and calls *buf_ptr* (buffer pointer).

```
buf_ptr     mov    temp,w
            mov    fsr, temp
            cjae   temp,#030h,:bank3
            cjae   temp,#020h,:bank2
            cjae   temp,#010h,:bank1
            jmp    :bank0
:bank3      add    fsr,#010h
:bank2      add    fsr,#010h
:bank1      add    fsr,#010h
:bank0      add    fsr,#010h
            ret
```

It may be more useful in some applications to treat these memory locations as register banks, as they are described in the Microchip literature. According to this model, bit 4 of the *fsr* enables bank selection when it is a 1. The 16-byte bank in use is then selected by bits 5 and 6 of the *fsr* as shown in figure 2.

This model explains the warps in the memory map. Each of the three warp addresses (20h, 40h, and 60h) has a 0 in the bit-4 position. This disables banked memory, causing the PIC to disregard all but bits 0 through 3 of the address.