C H A P T E R 1 3

Writing 32-Bit Applications

This chapter is an introduction to 32-bit programming for the 80386. The guidelines in this chapter also apply to the 80486 processor, which is basically a faster 80386 with the equivalent of a 80387 floating-point processor. Since you are already familiar with 16-bit real-mode programming, this chapter covers the differences between 16-bit programming and 32-bit protected-mode programming.

The 80386 processor (and its successors such as the 80486) can run in real mode, virtual-86 mode, and in protected mode. In real and virtual-86 modes, the 80386 can run 8086/8088 programs. In protected mode, it can run 80286 programs. The 386 also extends the features of protected mode to include 32-bit operations and segments larger than 64K.

The MS-DOS operating system directly supports 8086/8088 programs, which it runs either in real mode or virtual-86 mode. Native 32-bit 80386 programs can be run by using a "DOS extender," by using the WINMEM32.DLL facility of Microsoft Windows 3.x, or by running a native 32-bit operating system, such as Microsoft Windows NT. You can use MASM to generate object code (OMF or COFF) for 32-bit programs. To do this, you will need a software development kit such as the Windows SDK for the target environment. Such kits include the linker and other components specific to your chosen operating environment.

32-Bit Memory Addressing

The 80386 has six segment registers. Four of these are familiar to 8086/8088 programmers: CS (Code Segment), SS (Stack Segment), DS (Data Segment), and ES (Extra Segment). The two additional registers, FS and GS, are used as data segment registers.

Memory addresses on 80x86 machines consist of two parts—a segment and an offset. In real-mode programs, the segment is a 16-bit number and the offset is a 16-bit number. Effective addresses are calculated by multiplying the segment by

16 and adding the offset to it. In protected mode, the segment value is not used directly as a number, but instead is an index to a table of "selectors." Each selector describes a block of memory, including attributes such as the size and location of the block, and the access rights the program has to it (read, write, execute). The effective address is calculated by adding the offset to the base address of the memory block described by the selector.

All segment registers are 16 bits wide. The offset in a 32-bit protected-mode program is itself 32 bits wide, which means that a single segment can address up to 4 gigabytes of memory. Because of this large range, there is little need to use segment registers to extend the range of addresses in 32-bit programs. If all six segment registers are initially set to the same value, then the rest of the program can ignore them and treat the processor as if it used a 32-bit linear address space. This is called 0:32, or flat, addressing. (The full segmented 32-bit addressing mode, in which the segment registers can contain different values, is called 16:32 addressing.) Flat addressing is used by the Windows NT operating system.

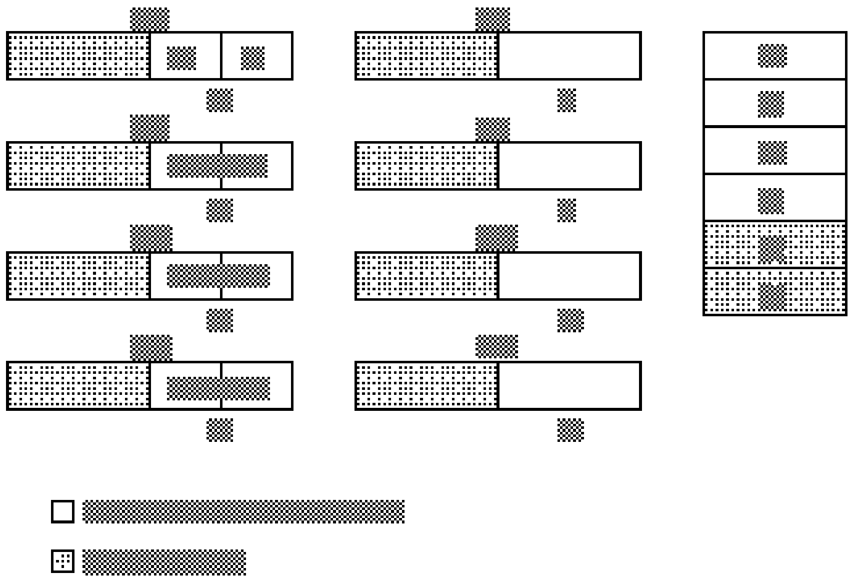

Figure 13.1 32-Bit Register Set

MASM Directives for 32-Bit Programming

If you use the simplified segment directives, a 32-bit program is surprisingly similar to a program for MS-DOS. Here are the differences:

- Supply the **.386** directive, which enables the 32-bit programming features of the 386 and its successors. The **.386** directive must precede the **.MODEL** directive.
- For flat-model programming, use the directive

  ```
  .MODEL  flat, stdcall
  ```

  which tells the assembler to assume flat model (0:32) and to use the Windows NT standard calling convention for subroutine calls.
- Precede your data declarations with the **.DATA** directive.
- Precede your instruction codes with the **.CODE** directive.
- At the end of the source file, place an **END** directive.

Sample Program

The following sample is a 32-bit assembly language subroutine, such as might be called from a 32-bit C program written for the Windows NT operating system. The program illustrates the use of a variety of directives to make assembly language easier to read and maintain. Note that with 32-bit flat model programming, there is no longer any need to refer to segment registers, since these are artifacts of segmented addressing.

```
;* szSearch - An example of 32-bit assembly programming using MASM 6.1
;*
;* Purpose: Search a buffer (rgbSearch) of length cbSearch for the
;*          first occurrence of szTok (null terminated string).
;*
;* Method:  A variation of the Boyer-Moore method
;*          1. Determine length of szTok (n)
;*          2. Set array of flags (rgfInTok) to TRUE for each character
;*                  in szTok
;*          3. Set current position of search to rgbSearch (pbCur)
;*          4. Compare current position to szTok by searching backwards
;*                  from the nth position. When a comparison fails at
;*                  position (m), check to see if the current character
;*                  in rgbSearch is in szTok by using rgfInTok. If not,
;*                  set pbCur to pbCur+(m)+1 and restart compare. If
;*                  pbCur reached, increment pbCur and restart compare.
;*          5. Reset rgfInTok to all 0 for next instantiation of the
;*                  routine.

        .386
        .MODEL  flat, stdcall

FALSE   EQU     0
TRUE    EQU     NOT FALSE
```

```
        .DATA
; Flags buffer - data initialized to FALSE. We will
; set the appropriate flags to TRUE during initialization
; of szSearch and reset them to FALSE before exit.
rgfInTok        BYTE    256 DUP (FALSE);

        .CODE

PBYTE   TYPEDEF PTR BYTE

szSearch PROC PUBLIC USES esi edi,
         rgbSearch:PBYTE,
         cbSearch:DWORD,
         szTok:PBYTE

; Initialize flags buffer. This tells us if a character is in
; the search token - Note how we use EAX as an index
; register. This can be done with all extended registers.
        mov     esi, szTok
        xor     eax, eax
        .REPEAT
        lodsb
        mov     BYTE PTR rgfInTok[eax], TRUE
        .UNTIL  (!AL)

; Save count of szTok bytes in EDX
        mov     edx, esi
        sub     edx, szTok
        dec     edx

; ESI will always point to beginning of szTok
        mov     esi, szTok

; EDI will point to current search position
; and will also contain the return value
        mov     edi, rgbSearch

; Store pointer to end of rgbSearch in EBX
        mov     ebx, edi
        add     ebx, cbSearch
        sub     ebx, edx
```

```
; Initialize ECX with length of szTok
        mov     ecx, edx
        .WHILE  ( ecx != 0 )
        dec     ecx             ; Move index to current
        mov     al, [edi+ecx]   ;   characters to compare

; If the current byte in the buffer doesn't exist in the
; search token, increment buffer pointer to current position
; +1 and start over. This can skip up to 'EDX'
; bytes and reduce search time.
        .IF     !(rgfInTok[eax])
        add     edi, ecx
        inc     edi             ; Initialize ECX with
        mov     ecx, edx        ;   length of szTok
; Otherwise, if the characters match, continue on as if
; we have a matching token
        .ELSEIF (al == [esi+ecx])
        .CONTINUE
; Finally, if we have searched all szTok characters,
; and land here, we have a mismatch and we increment
; our pointer into rgbSearch by one and start over.
        .ELSEIF (!ecx)
        inc     edi
        mov     ecx, edx
        .ENDIF

; Verify that we haven't searched beyond the buffer.
        .IF     (edi > ebx)
        mov     edi, 0          ; Error value
        .BREAK
        .ENDIF
        .ENDW

; Restore flags in rgfInTok to 0 (for next time).
        mov     esi, szTok
        xor     eax, eax
        .REPEAT
        lodsb
        mov     BYTE PTR rgfInTok[eax], FALSE
        .UNTIL  !AL

; Put return value in eax
        mov     eax, edi
        ret
szSearch ENDP

end
```