Character Translation Methods

One task that assembly language programs handle best is character translation. There are many computer applications for this. The rapidly expanding field of data encryption is one, where data files must be securely stored and transmitted between computers while their contents are kept secret. Another application is data communications, where keyboard and screen codes must be translated in order to emulate various terminals. Often, we need to translate characters from one encoding system to another—from ASCII to EBCDIC, for example. A critical factor in each of these applications is speed, which just happens to be a feature of assembly language.

The XLAT Instruction

The XLAT instruction adds AL to EBX and uses the resulting offset to point to an entry in an 8-bit *translate table*. This table contains values that are substituted for the original value in AL. The byte in the table entry pointed to by EBX + AL is moved to AL. The syntax is

```
XLAT  [tablename]
```

Tablename is optional because the table is assumed to be pointed to by EBX (or BX, in Real-address mode). Therefore, be sure to load BX with the offset of the translate table before invoking XLAT. The flags are not affected by this instruction. The table can have a maximum of 256 entries, the same range of values possible in the 8-bit AL register.

Example Let's store the characters representing all 16 hexadecimal digits in a table:

```
table BYTE '0123456789ABCDEF'
```

The table contains the ASCII code of each hexadecimal digit. If we place 0Ah in AL with the thought of converting it to ASCII, we can set EBX to the table offset and invoke XLAT. The instruction adds EBX and AL, generating an effective address that points to the eleventh entry in the table. It then moves the contents of this table entry to A:

```
mov al,0Ah                    ; index value
mov ebx,OFFSET table          ; point to table
xlat                          ; AL = 41h, or 'A'
```

Character Filtering

One of the best uses of XLAT is to filter out unwanted characters from a stream of text. Suppose we want to input a string of characters from the keyboard and echo only those with ASCII values from 32 to 127. We can set up a translate table, place a zero in each table position corresponding to an invalid character, and place 0FFh in each valid position:

```
validchars  BYTE 32 DUP(0)       ; invalid chars: 0-31
   BYTE 96 DUP(0FFh)             ; valid chars:   32-127
   BYTE 128 DUP(0)               ; invalid chars: 128-255
```

The following *Xlat.asm* program includes code that inputs a series of characters and uses them to look up values in the **validchars** table:

```
TITLE Character Filtering               (Xlat.asm)

; This program filters input from the console
; by screening out all ASCII codes less than
; 32 or greater than 127. Uses INT 16h for
; direct keyboard input.

INCLUDE Irvine32.inc

INPUT_LENGTH = 20
.data
validchars  BYTE 32 DUP(0)       ; invalid chars: 0-31
   BYTE 96 DUP(0FFh)             ; valid chars:   32-127
   BYTE 128 DUP(0)               ; invalid chars: 128-255

.code
main PROC
   mov ebx,offset validchars
   mov ecx,INPUT_LENGTH

getchar:
   call ReadChar                 ; read character into AL
   mov  dl,al                    ; save copy in DL
   xlat validchars               ; look up char in AL
   or   al,al                    ; invalid char?
   jz   getchar                  ; yes: get another
   mov  al,dl
   call WriteChar                ; output character in AL
   loop getchar

   call Crlf
   exit
main ENDP
END main
```

If XLAT returns a value of zero in AL, we skip the character and jump back to the top of the loop. (When AL is ORed with itself, the Zero flag is set if AL equals 0). If the character is valid, 0FFh is returned in AL, and we call **WriteChar** to display the character in DL.

Character Encoding

The XLAT instruction provides a simple way to encode data so it cannot be read by unauthorized persons. When messages are transferred across telephone lines, for instance, encoding can be a way of preventing others from reading them. Imagine a table in which all the possible digits and letters have been rearranged. A program could read each character from standard input, use XLAT to look it up in a table, and write its encoded value to standard output. A sample table is as follows:

```
codetable LABEL BYTE
   BYTE 48 DUP(0)               ; no translation
   BYTE '4590821367'            ; ASCII codes 48-57
   BYTE 7 DUP (0)               ; no translation
   BYTE 'GVHZUSOBMIKPJCADLFTYEQNWXR'
   BYTE 6 DUP (0)               ; no translation
   BYTE 'gvhzusobmikpjcadlftyeqnwxr'
   BYTE 133 DUP(0)              ; no translation
```

Certain ranges in the table are set to zeros; characters in these ranges are not translated. The $ character (ASCII 36), for example, is not translated because position 36 in the table contains the value 0.

Sample Program Let's take a look at a character encoding program that encodes each character read from an input file. When running the program, one can redirect standard input from a file, using the < symbol. For example:

```
encode < infile
```

The program output appears on the screen. Output can also be redirect to a file:

```
encode < infile > outfile
```

The following example shows a line from an input file that has been encoded:

```
This is a SECRET Message         (read from input file)
Ybmt mt g TUHFUY Juttgou         (encoded output)
```

The program cannot use the keyboard as the standard input device, because it quits looking for input as soon as the keyboard buffer is empty. The user would have to type fast enough to keep the keyboard buffer full.

Given some time and effort, a simple encoding scheme like this can be broken. The easiest way to break the code is to gain access to the program itself and use it to encode a known message. But if you can prevent others from running the program, breaking the code takes more time. Another way to discourage code breaking is to constantly change the code. In World War

II, for example, pilots carried a code book for translating messages, so that message encryption could be varied on a day-to-day basis.

```
TITLE Character Encoding Program    (Encode.asm)

; This program reads an input file and encodes
; the output using the XLAT instruction.
; To run it, redirect input at the Command prompt:
;
;            encode < input.txt
;
; Implemented as a 16-bit application because we can
; use INT 21h function 6 to input characters without
; waiting. See Section 13.2.3 for details.
; Last update: 06/01/02

INCLUDE Irvine16.inc

.data
codetable LABEL BYTE
   BYTE 48 DUP(0)               ; no translation
   BYTE '4590821367             '; ASCII codes 48-57
   BYTE 7 DUP (0)               ; no translation
   BYTE 'GVHZUSOBMIKPJCADLFTYEQNWXR'
   BYTE 6 DUP (0)               ; no translation
   BYTE 'gvhzusobmikpjcadlftyeqnwxr'
   BYTE 133 DUP(0)              ; no translation

.code
main PROC
   mov ax,@data
   mov ds,ax
   mov bx,OFFSET codetable

getchar:
   push bx
   mov ah,6                     ; input character, don't wait
   mov dl,0FFh
   int 21h                      ; call DOS
   pop bx
   jz  quit                     ; quit, no input waiting
   mov dl,al                    ; save char in DL
   xlat                         ; translate the character
   cmp al,0                     ; translatable?
   je  putchar                  ; no: write it as is
   mov dl,al                    ; yes: move new char to DL

putchar:
```

```
    mov  al,dl                  ; character is in DL
    call WriteChar              ; write AL to standard output
    jmp  getchar                ; get another char

quit:
    exit
main ENDP
END main
```