Calculation Via Table Lookups Chapter Twelve

12.1 Chapter Overview

This chapter discusses arithmetic computation via table lookup. By the conclusion of this chapter you should be able to use table lookups to quickly compute complex functions. You will also learn how to construct these tables programmatically.

12.2 Tables

The term “table” has different meanings to different programmers. To most assembly language programmers, a table is nothing more than an array that is initialized with some data. The assembly language programmer often uses tables to compute complex or otherwise slow functions. Many very high level languages (e.g., SNOBOL4 and Icon) directly support a table data type. Tables in these languages are essentially arrays whose elements you can access with a non-integer index (e.g., floating point, string, or any other data type). HLA provides a table module that lets you index an array using a string. However, in this chapter we will adopt the assembly language programmer’s view of tables.

A table is an array containing preinitialized values that do not change during the execution of the program. A table can be compared to an array in the same way an integer constant can be compared to an integer variable. In assembly language, you can use tables for a variety of purposes: computing functions, controlling program flow, or simply “looking things up”. In general, tables provide a fast mechanism for performing some operation at the expense of some space in your program (the extra space holds the tabular data). In the following sections we’ll explore some of the many possible uses of tables in an assembly language program.

Note: since tables typically contain preinitialized data that does not change during program execution, the READONLY section is a good place to declare your table objects.

12.2.1 Function Computation via Table Look-up

Tables can do all kinds of things in assembly language. In HLLs, like Pascal, it’s real easy to create a formula which computes some value. A simple looking arithmetic expression can be equivalent to a considerable amount of 80x86 assembly language code. Assembly language programmers tend to compute many values via table look up rather than through the execution of some function. This has the advantage of being easier, and often more efficient as well. Consider the following Pascal statement:

if (character >= ‘a’) and (character <= ‘z’) then character := chr(ord(character) - 32);

This Pascal if statement converts the character variable character from lower case to upper case if character is in the range ‘a’..’z’. The HLA code that does the same thing is

```
mov( character, al );
if( al in ‘a’..’z’ ) then

    and( $5f, al );       // Same as SUB( 32, al ) in this code.

endif;
mov( al, character );
```

Note that HLA’s high level IF statement translates into four machine instructions in this particular example. Hence, this code requires a total of seven machine instructions.

Had you buried this code in a nested loop, you'd be hard pressed to improve the speed of this code without using a table look up. Using a table look up, however, allows you to reduce this sequence of instructions to just four instructions:

```
        mov( character, al );
        lea( ebx, CnvrtLower );
        xlat
        mov( al, character );
```

You're probably wondering how this code works and what is this new instruction, XLAT? The XLAT, or *translate*, instruction does the following:

```
                                mov( [ebx+al*1], al );
```

That is, it uses the current value of the AL register as an index into the array whose base address is contained in EBX. It fetches the byte at that index in the array and copies that byte into the AL register. Intel calls this the translate instruction because programmers typically use it to translate characters from one form to another using a lookup table. That's exactly how we are using it here.

In the previous example, *CnvrtLower* is a 256-byte table which contains the values 0..$60 at indices 0..$60, $41..$5A at indices $61..$7A, and $7B..$FF at indices $7Bh..0FF. Therefore, if AL contains a value in the range $0..$60, the XLAT instruction returns the value $0..$60, effectively leaving AL unchanged. However, if AL contains a value in the range $61..$7A (the ASCII codes for 'a'..'z') then the XLAT instruction replaces the value in AL with a value in the range $41..$5A. $41..$5A just happen to be the ASCII codes for 'A'..'Z'. Therefore, if AL originally contains an lower case character ($61..$7A), the XLAT instruction replaces the value in AL with a corresponding value in the range $61..$7A, effectively converting the original lower case character ($61..$7A) to an upper case character ($41..$5A). The remaining entries in the table, like entries $0..$60, simply contain the index into the table of their particular element. Therefore, if AL originally contains a value in the range $7A..$FF, the XLAT instruction will return the corresponding table entry that also contains $7A..$FF.

As the complexity of the function increases, the performance benefits of the table look up method increase dramatically. While you would almost never use a look up table to convert lower case to upper case, consider what happens if you want to swap cases:

```
Via computation:

        mov( character, al );
        if( al in 'a'..'z' ) then

            and( $5f, al );

        elseif( al in 'A'..'Z' ) then

            or( $20, al );

        endif;
        mov( al, character ):
```

The IF and ELSEIF statements generate four and five actual machine instructions, respectively, so this code is equivalent to 13 actual machine instructions.

The table look up code to compute this same function is:

```
        mov( character, al );
        lea( ebx, SwapUL );
        xlat();
        mov( al, character );
```

As you can see, when using a table look up to compute a function only the table changes, the code remains the same.

Table look ups suffer from one major problem – functions computed via table look up have a limited domain. The domain of a function is the set of possible input values (parameters) it will accept. For example, the upper/lower case conversion functions above have the 256-character ASCII character set as their domain.

A function such as SIN or COS accepts the set of real numbers as possible input values. Clearly the domain for SIN and COS is much larger than for the upper/lower case conversion function. If you are going to do computations via table look up, you must limit the domain of a function to a small set. This is because each element in the domain of a function requires an entry in the look up table. You won't find it very practical to implement a function via table look up whose domain the set of real numbers.

Most look up tables are quite small, usually 10 to 128 entries. Rarely do look up tables grow beyond 1,000 entries. Most programmers don't have the patience to create (and verify the correctness) of a 1,000 entry table.

Another limitation of functions based on look up tables is that the elements in the domain of the function must be fairly contiguous. Table look ups take the input value for a function, use this input value as an index into the table, and return the value at that entry in the table. If you do not pass a function any values other than 0, 100, 1,000, and 10,000 it would seem an ideal candidate for implementation via table look up, its domain consists of only four items. However, the table would actually require 10,001 different elements due to the range of the input values. Therefore, you cannot efficiently create such a function via a table look up. Throughout this section on tables, we'll assume that the domain of the function is a fairly contiguous set of values.

The best functions you can implement via table look ups are those whose domain and range is always 0..255 (or some subset of this range). You can efficiently implement such functions on the 80x86 via the XLAT instruction. The upper/lower case conversion routines presented earlier are good examples of such a function. Any function in this class (those whose domain and range take on the values 0..255) can be computed using the same two instructions: "lea(table, ebx);" and "xlat();" The only thing that ever changes is the look up table.

You cannot (conveniently) use the XLAT instruction to compute a function value once the range or domain of the function takes on values outside 0..255. There are three situations to consider:

- The domain is outside 0..255 but the range is within 0..255,
- The domain is inside 0..255 but the range is outside 0..255, and
- Both the domain and range of the function take on values outside 0..255.

We will consider each of these cases separately.

If the domain of a function is outside 0..255 but the range of the function falls within this set of values, our look up table will require more than 256 entries but we can represent each entry with a single byte. Therefore, the look up table can be an array of bytes. Next to look ups involving the XLAT instruction, functions falling into this class are the most efficient. The following Pascal function invocation,

```
B := Func(X);
```

where *Func* is

```
function Func(X:dword):byte;
```

consists of the following HLA code:

```
    mov( X, ebx );
    mov( FuncTable[ ebx ], al );
    mov( al, B );
```

This code loads the function parameter into EBX, uses this value (in the range 0..??) as an index into the *FuncTable* table, fetches the byte at that location, and stores the result into *B*. Obviously, the table must contain a valid entry for each possible value of *X*. For example, suppose you wanted to map a cursor position on the video screen in the range 0..1999 (there are 2,000 character positions on an 80x25 video display) to its *X* or *Y* coordinate on the screen. You could easily compute the *X* coordinate via the function:

X:=Posn mod 80

and the *Y* coordinate with the formula

```
Y:=Posn div 80
```

(where *Posn* is the cursor position on the screen). This can be easily computed using the 80x86 code:

```
        mov( Posn, ax );
        div( 80, ax );

// X is now in AH, Y is now in AL
```

However, the DIV instruction on the 80x86 is very slow. If you need to do this computation for every character you write to the screen, you will seriously degrade the speed of your video display code. The following code, which realizes these two functions via table look up, would improve the performance of your code considerably:

```
movzx( Posn, ebx );          // Use a plain MOV instr if Posn is uns32
mov( YCoord[ebx], al );      // rather than an uns16 value.
mov( XCoord[ebx], ah );
```

If the domain of a function is within 0..255 but the range is outside this set, the look up table will contain 256 or fewer entries but each entry will require two or more bytes. If both the range and domains of the function are outside 0..255, each entry will require two or more bytes and the table will contain more than 256 entries.

Recall from the chapter on arrays that the formula for indexing into a single dimensional array (of which a table is a special case) is

```
Address := Base + index * size
```

If elements in the range of the function require two bytes, then the index must be multiplied by two before indexing into the table. Likewise, if each entry requires three, four, or more bytes, the index must be multiplied by the size of each table entry before being used as an index into the table. For example, suppose you have a function, F(x), defined by the following (pseudo) Pascal declaration:

```
function F(x:dword):word;
```

You can easily create this function using the following 80x86 code (and, of course, the appropriate table named *F*):

```
mov( X, ebx );
mov( F[ebx*2], ax );
```

Any function whose domain is small and mostly contiguous is a good candidate for computation via table look up. In some cases, non-contiguous domains are acceptable as well, as long as the domain can be coerced into an appropriate set of values. Such operations are called conditioning and are the subject of the next section.

12.2.2 Domain Conditioning

Domain conditioning is taking a set of values in the domain of a function and massaging them so that they are more acceptable as inputs to that function. Consider the following function:

$$\sin x = \langle \sin x | x \in [-2\pi, 2\pi] \rangle$$

This says that the (computer) function SIN(x) is equivalent to the (mathematical) function *sin x* where

$-2\pi \le x \le 2\pi$

As we all know, sine is a circular function which will accept any real valued input. The formula used to compute sine, however, only accept a small set of these values.

This range limitation doesn't present any real problems, by simply computing SIN(X mod (2*pi)) we can compute the sine of any input value. Modifying an input value so that we can easily compute a function is called conditioning the input. In the example above we computed "X mod 2*pi" and used the result as the input to the *sin* function. This truncates *X* to the domain *sin* needs without affecting the result. We can apply input conditioning to table look ups as well. In fact, scaling the index to handle word entries is a form of input conditioning. Consider the following Pascal function:

```
function val(x:word):word; begin
    case x of
        0: val := 1;
        1: val := 1;
        2: val := 4;
        3: val := 27;
        4: val := 256;
        otherwise val := 0;
    end;
end;
```

This function computes some value for x in the range 0..4 and it returns zero if *x* is outside this range. Since *x* can take on 65,536 different values (being a 16 bit word), creating a table containing 65,536 words where only the first five entries are non-zero seems to be quite wasteful. However, we can still compute this function using a table look up if we use input conditioning. The following assembly language code presents this principle:

```
        mov( 0, ax );         // AX = 0, assume X > 4.
        movzx( x, ebx );      // Note that H.O. bits of EBX must be zero!
        if( bx <= 4 ) then

            mov( val[ ebx*2 ], ax );

        endif;
```

This code checks to see if *x* is outside the range 0..4. If so, it manually sets AX to zero, otherwise it looks up the function value through the *val* table. With input conditioning, you can implement several functions that would otherwise be impractical to do via table look up.

12.2.3 Generating Tables

One big problem with using table look ups is creating the table in the first place. This is particularly true if there are a large number of entries in the table. Figuring out the data to place in the table, then laboriously entering the data, and, finally, checking that data to make sure it is valid, is a very time-staking and boring process. For many tables, there is no way around this process. For other tables there is a better way – use the computer to generate the table for you. An example is probably the best way to describe this. Consider the following modification to the sine function:

$$(\sin x) \times r = \langle \frac{(r \times (1000 \times \sin x))}{1000} | x \in [0, 359] \rangle$$

This states that *x* is an integer in the range 0..359 and *r* must be an integer. The computer can easily compute this with the following code:

```
        movzx( x, ebx );
        mov( Sines[ ebx*2], eax );   // Get SIN(X) * 1000
        imul( r, eax );              // Note that this extends EAX into EDX.
        idiv( 1000, edx:eax );       // Compute (R*(SIN(X)*1000)) / 1000
```

Note that integer multiplication and division are not associative. You cannot remove the multiplication by 1000 and the division by 1000 because they seem to cancel one another out. Furthermore, this code must compute this function in exactly this order. All that we need to complete this function is a table containing 360 different values corresponding to the sine of the angle (in degrees) times 1,000. Entering such a table into an assembly language program containing such values is extremely boring and you'd probably make several mistakes entering and verifying this data. However, you can have the program generate this table for you. Consider the following HLA program:

```
program GenerateSines;
#include( “stdlib.hhf” );

var
    outFile: dword;
    angle:   int32;
    r:       int32;

readonly
    RoundMode: uns16 := $23f;



begin GenerateSines;

    // Open the file:

    mov( fileio.openNew( “sines.hla” ), outFile );

    // Emit the initial part of the declaration to the output file:

    fileio.put
    (
        outFile,
        stdio.tab,
        “sines: int32[360] := “ nl,
        stdio.tab, stdio.tab, stdio.tab, “[“ nl );

    // Enable rounding control (round to the nearest integer).

    fldcw( RoundMode );

    // Emit the sines table:

    for( mov( 0, angle); angle < 359; inc( angle )) do

        // Convert angle in degrees to an angle in radians
        // using “radians := angle * 2.0 * pi / 360.0;”

        fild( angle );
        fld( 2.0 );
        fmul();
        fldpi();
        fmul();
        fld( 360.0 );
        fdiv();

        // Okay, compute the sine of ST0

        fsin();

        // Multiply by 1000 and store the rounded result into
        // the integer variable r.
```

```
            fld( 1000.0 );
            fmul();
            fistp( r );

            // Write out the integers eight per line to the source file:
            // Note: if (angle AND %111) is zero, then angle is evenly
            // divisible by eight and we should output a newline first.

            test( %111, angle );
            if( @z ) then

                fileio.put
                (
                    outFile,
                    nl,
                    stdio.tab,
                    stdio.tab,
                    stdio.tab,
                    stdio.tab,
                    r:5,
                    ‘,’
                );

            else

                fileio.put( outFile, r:5, ‘,’ );

            endif;

        endfor;

        // Output sine(359) as a special case (no comma following it).
        // Note: this value was computed manually with a calculator.

        fileio.put
        (
            outFile,
            “  -17”,
            nl,
            stdio.tab,
            stdio.tab,
            stdio.tab,
            “];”,
            nl
        );
        fileio.close( outFile );

    end GenerateSines;
```

Program 12.1 An HLA Program that Generates a Table of Sines

The program above produces the following output:

```
sines: int32[360] :=
    [

            0,   17,   35,   52,   70,   87,  105,  122,
          139,  156,  174,  191,  208,  225,  242,  259,
          276,  292,  309,  326,  342,  358,  375,  391,
          407,  423,  438,  454,  469,  485,  500,  515,
          530,  545,  559,  574,  588,  602,  616,  629,
          643,  656,  669,  682,  695,  707,  719,  731,
          743,  755,  766,  777,  788,  799,  809,  819,
          829,  839,  848,  857,  866,  875,  883,  891,
          899,  906,  914,  921,  927,  934,  940,  946,
          951,  956,  961,  966,  970,  974,  978,  982,
          985,  988,  990,  993,  995,  996,  998,  999,
          999, 1000, 1000, 1000,  999,  999,  998,  996,
          995,  993,  990,  988,  985,  982,  978,  974,
          970,  966,  961,  956,  951,  946,  940,  934,
          927,  921,  914,  906,  899,  891,  883,  875,
          866,  857,  848,  839,  829,  819,  809,  799,
          788,  777,  766,  755,  743,  731,  719,  707,
          695,  682,  669,  656,  643,  629,  616,  602,
          588,  574,  559,  545,  530,  515,  500,  485,
          469,  454,  438,  423,  407,  391,  375,  358,
          342,  326,  309,  292,  276,  259,  242,  225,
          208,  191,  174,  156,  139,  122,  105,   87,
           70,   52,   35,   17,    0,  -17,  -35,  -52,
          -70,  -87, -105, -122, -139, -156, -174, -191,
         -208, -225, -242, -259, -276, -292, -309, -326,
         -342, -358, -375, -391, -407, -423, -438, -454,
         -469, -485, -500, -515, -530, -545, -559, -574,
         -588, -602, -616, -629, -643, -656, -669, -682,
         -695, -707, -719, -731, -743, -755, -766, -777,
         -788, -799, -809, -819, -829, -839, -848, -857,
         -866, -875, -883, -891, -899, -906, -914, -921,
         -927, -934, -940, -946, -951, -956, -961, -966,
         -970, -974, -978, -982, -985, -988, -990, -993,
         -995, -996, -998, -999, -999,-1000,-1000,-1000,
         -999, -999, -998, -996, -995, -993, -990, -988,
         -985, -982, -978, -974, -970, -966, -961, -956,
         -951, -946, -940, -934, -927, -921, -914, -906,
         -899, -891, -883, -875, -866, -857, -848, -839,
         -829, -819, -809, -799, -788, -777, -766, -755,
         -743, -731, -719, -707, -695, -682, -669, -656,
         -643, -629, -616, -602, -588, -574, -559, -545,
         -530, -515, -500, -485, -469, -454, -438, -423,
         -407, -391, -375, -358, -342, -326, -309, -292,
         -276, -259, -242, -225, -208, -191, -174, -156,
         -139, -122, -105,  -87,  -70,  -52,  -35,  -17
    ];
```

Obviously it's much easier to write the HLA program that generated this data than to enter (and verify) this data by hand. Of course, you don't even have to write the table generation program in HLA. If you prefer, you might find it easier to write the program in Pascal/Delphi, C/C++, or some other high level language. Obviously, the program will only execute once, so the performance of the table generation program is not an issue. If it's easier to write the table generation program in a high level language, by all means do so. Note, also, that HLA has a built-in interpreter that allows you to easily create tables without having to use an external program. For more details, see the chapter on macros and the HLA compile-time language.

Once you run your table generation program, all that remains to be done is to cut and paste the table from the file (sines.hla in this example) into the program that will actually use the table.

12.3 High Performance Implementation of cs.rangeChar

Way back in Chapter Three this volume made the comment that the implementation of the *cs.rangeChar* was not very efficient when generating large character sets (see "Character Set Functions That Build Sets" on page 449). That chapter also mentioned that a table lookup would be a better solution for this function if you generate large character sets. That chapter also promised an table lookup implementation of *cs.rangeChar*. This section fulfills that promise.

Program 12.2 provides a table lookup implementation of this function. To understand how this function works, consider the two tables (*StartRange* and *EndRange*) appearing in this program.

Each element in the *StartRange* table is a character set whose binary representation contains all one bits from bit position zero through the index into the table. That is, element zero contains a single '1' bit in bit position zero; element one contains one bits in bit positions zero and one; element two contains one bits in bit positions zero, one, and two; etc.

Each element of the *EndRange* table contains one bits from the bit position specified by the index into the table through to bit position 127. Therefore, element zero of this array contains all one bits from positions zero through 127; element one of this array contains a zero in bit position zero and ones in bit positions one through 127; element two of this array contains zeros in bit positions zero and one and it contains ones in bit positions two through 127; etc.

The *fastRangeChar* function builds a character set containing all the characters between two characters specified as parameters. The calling sequence for this function is

```
fastRangeChar( LowBoundChar, HighBoundChar, CsetVariable );
```

This function constructs the character set "{ LowBoundChar..HighBoundChar }" and stores this character set into *CsetVariable*.

As you may recall from the discussion of *cs.rangeChar's* low-level implementation, it constructed the character set by running a FOR loop from the *LowBoundChar* through to the *HighBoundChar* and set the corresponding bit in the character set on each iteration of the loop. So to build the character set {'a'..'z'} the loop would have to execute 26 times. The *fastRangeChar* function avoids this iteration by construction a set containing all elements from #0 to *HighBoundChar* and intersecting this set with a second character set containing all the characters from *LowBoundChar* to #127. The *fastRangeChar* function doesn't actually build these two sets, of course, it uses *HighBoundChar* and *LowBoundChar* as indices into the *StartRange* and *EndRange* tables, respectively. The intersection of these two table elements computes the desired result set. As you'll see by looking at *fastRangeChar*, this function computes the intersection of these two sets on the fly by using the AND instruction. Without further ado, here's the program:

```
program csRangeChar;
#include( "stdlib.hhf" )

static

    // Note: the following tables were generated
    // by the genRangeChar program:

    StartRange: cset[128] :=
            [
                {#0},
                {#0..#1},
                {#0..#2},
                {#0..#3},
                {#0..#4},
                {#0..#5},
{#0..#6},
{#0..#7},
{#0..#8},
{#0..#9},
{#0..#10},
{#0..#11},
{#0..#12},
{#0..#13},
{#0..#14},
{#0..#15},
{#0..#16},
{#0..#17},
{#0..#18},
{#0..#19},
{#0..#20},
{#0..#21},
{#0..#22},
{#0..#23},
{#0..#24},
{#0..#25},
{#0..#26},
{#0..#27},
{#0..#28},
{#0..#29},
{#0..#30},
{#0..#31},
{#0..#32},
{#0..#33},
{#0..#34},
{#0..#35},
{#0..#36},
{#0..#37},
{#0..#38},
{#0..#39},
{#0..#40},
{#0..#41},
{#0..#42},
{#0..#43},
{#0..#44},
{#0..#45},
{#0..#46},
{#0..#47},
{#0..#48},
{#0..#49},
{#0..#50},
{#0..#51},
{#0..#52},
{#0..#53},
{#0..#54},
{#0..#55},
{#0..#56},
{#0..#57},
{#0..#58},
{#0..#59},
{#0..#60},
{#0..#61},
{#0..#62},
{#0..#63},
{#0..#64},
{#0..#65},
{#0..#66},
                    {#0..#67},
                    {#0..#68},
                    {#0..#69},
                    {#0..#70},
                    {#0..#71},
                    {#0..#72},
                    {#0..#73},
                    {#0..#74},
                    {#0..#75},
                    {#0..#76},
                    {#0..#77},
                    {#0..#78},
                    {#0..#79},
                    {#0..#80},
                    {#0..#81},
                    {#0..#82},
                    {#0..#83},
                    {#0..#84},
                    {#0..#85},
                    {#0..#86},
                    {#0..#87},
                    {#0..#88},
                    {#0..#89},
                    {#0..#90},
                    {#0..#91},
                    {#0..#92},
                    {#0..#93},
                    {#0..#94},
                    {#0..#95},
                    {#0..#96},
                    {#0..#97},
                    {#0..#98},
                    {#0..#99},
                    {#0..#100},
                    {#0..#101},
                    {#0..#102},
                    {#0..#103},
                    {#0..#104},
                    {#0..#105},
                    {#0..#106},
                    {#0..#107},
                    {#0..#108},
                    {#0..#109},
                    {#0..#110},
                    {#0..#111},
                    {#0..#112},
                    {#0..#113},
                    {#0..#114},
                    {#0..#115},
                    {#0..#116},
                    {#0..#117},
                    {#0..#118},
                    {#0..#119},
                    {#0..#120},
                    {#0..#121},
                    {#0..#122},
                    {#0..#123},
                    {#0..#124},
                    {#0..#125},
                    {#0..#126},
                    {#0..#127}
```

```
                ];

        EndRange: cset[128] :=
                [
                    {#0..#127},
                    {#1..#127},
                    {#2..#127},
                    {#3..#127},
                    {#4..#127},
                    {#5..#127},
                    {#6..#127},
                    {#7..#127},
                    {#8..#127},
                    {#9..#127},
                    {#10..#127},
                    {#11..#127},
                    {#12..#127},
                    {#13..#127},
                    {#14..#127},
                    {#15..#127},
                    {#16..#127},
                    {#17..#127},
                    {#18..#127},
                    {#19..#127},
                    {#20..#127},
                    {#21..#127},
                    {#22..#127},
                    {#23..#127},
                    {#24..#127},
                    {#25..#127},
                    {#26..#127},
                    {#27..#127},
                    {#28..#127},
                    {#29..#127},
                    {#30..#127},
                    {#31..#127},
                    {#32..#127},
                    {#33..#127},
                    {#34..#127},
                    {#35..#127},
                    {#36..#127},
                    {#37..#127},
                    {#38..#127},
                    {#39..#127},
                    {#40..#127},
                    {#41..#127},
                    {#42..#127},
                    {#43..#127},
                    {#44..#127},
                    {#45..#127},
                    {#46..#127},
                    {#47..#127},
                    {#48..#127},
                    {#49..#127},
                    {#50..#127},
                    {#51..#127},
                    {#52..#127},
                    {#53..#127},
                    {#54..#127},
                    {#55..#127},
                    {#56..#127},
                                        {#57..#127},
                                        {#58..#127},
                                        {#59..#127},
                                        {#60..#127},
                                        {#61..#127},
                                        {#62..#127},
                                        {#63..#127},
                                        {#64..#127},
                                        {#65..#127},
                                        {#66..#127},
                                        {#67..#127},
                                        {#68..#127},
                                        {#69..#127},
                                        {#70..#127},
                                        {#71..#127},
                                        {#72..#127},
                                        {#73..#127},
                                        {#74..#127},
                                        {#75..#127},
                                        {#76..#127},
                                        {#77..#127},
                                        {#78..#127},
                                        {#79..#127},
                                        {#80..#127},
                                        {#81..#127},
                                        {#82..#127},
                                        {#83..#127},
                                        {#84..#127},
                                        {#85..#127},
                                        {#86..#127},
                                        {#87..#127},
                                        {#88..#127},
                                        {#89..#127},
                                        {#90..#127},
                                        {#91..#127},
                                        {#92..#127},
                                        {#93..#127},
                                        {#94..#127},
                                        {#95..#127},
                                        {#96..#127},
                                        {#97..#127},
                                        {#98..#127},
                                        {#99..#127},
                                        {#100..#127},
                                        {#101..#127},
                                        {#102..#127},
                                        {#103..#127},
                                        {#104..#127},
                                        {#105..#127},
                                        {#106..#127},
                                        {#107..#127},
                                        {#108..#127},
                                        {#109..#127},
                                        {#110..#127},
                                        {#111..#127},
                                        {#112..#127},
                                        {#113..#127},
                                        {#114..#127},
                                        {#115..#127},
                                        {#116..#127},
                                        {#117..#127},
                    {#118..#127},
                    {#119..#127},
                    {#120..#127},
                    {#121..#127},
                    {#122..#127},
                    {#123..#127},
                    {#124..#127},
                    {#125..#127},
                    {#126..#127},
                    {#127}
                ];


/**********************************************************************/
/*                                                                    */
/* fastRangeChar-                                                     */
/*                                                                    */
/* A fast implementation of cs.rangeChar that uses a table lookup     */
/* to speed up the generation of the character set for really large   */
/* sets (note: because of memory latencies, this function is probably */
/* slower than cs.rangeChar for small character sets).                */
/*                                                                    */
/**********************************************************************/

procedure fastRangeChar( LowBound:char; HighBound:char; var csDest:cset );
begin fastRangeChar;

    push( eax );
    push( ebx );
    push( esi );
    push( edi );
    mov( csDest, ebx );     // Get pointer to destination character set.

    // Copy EndRange[ LowBound ] into csDest.  This adds all the
    // characters from LowBound to #127 into csDest.  Then intersect
    // this set with StartRange[ HighBound ] to trim off all the
    // characters after HighBound.

    movzx( LowBound, esi );
    shl( 4, esi );                // *16 'cause each element is 16 bytes.
    movzx( HighBound, edi );
    shl( 4, edi );

    mov( (type dword EndRange[ esi + 0 ]), eax );
    and( (type dword StartRange[ edi + 0 ]), eax );  // Does the intersection.
    mov( eax, (type dword [ ebx+0 ]));

    mov( (type dword EndRange[ esi + 4 ]), eax );
    and( (type dword StartRange[ edi + 4 ]), eax );
    mov( eax, (type dword [ ebx+4 ]));

    mov( (type dword EndRange[ esi + 8 ]), eax );
    and( (type dword StartRange[ edi + 8 ]), eax );
    mov( eax, (type dword [ ebx+8 ]));

    mov( (type dword EndRange[ esi + 12 ]), eax );
    and( (type dword StartRange[ edi + 12 ]), eax );
    mov( eax, (type dword [ ebx+12 ]));

    pop( edi );
    pop( esi );
    pop( ebx );
    pop( eax );

end fastRangeChar;



static
    TestCset: cset := {};

begin csRangeChar;

    fastRangeChar( ‘a’, ‘z’, TestCset );
    stdout.put( “Result from fastRangeChar: {“, TestCset, “}” nl );

end csRangeChar;
```

Program 12.2 Table Lookup Implementation of cs.rangeChar

Naturally, the *StartRange* and *EndRange* tables were not hand-generated. An HLA program generated these two tables (with a combined 512 elements). Program 12.3 is the program that generated these tables.

```
program GenerateRangeChar;
#include( “stdlib.hhf” );

var
    outFile: dword;



begin GenerateRangeChar;

    // Open the file:

    mov( fileio.openNew( “rangeCharCset.hla” ), outFile );

    // Emit the initial part of the declaration to the output file:

    fileio.put
    (
        outFile,
        stdio.tab,
        “StartRange: cset[128] := “ nl,
        stdio.tab, stdio.tab, stdio.tab, “[“ nl,
        stdio.tab, stdio.tab, stdio.tab, stdio.tab, “{#0},” nl  // element zero

    );
    for( mov( 1, ecx ); ecx < 127; inc( ecx )) do

        fileio.put
        (
            outFile,
            stdio.tab, stdio.tab, stdio.tab, stdio.tab,
                "{#0..#",
                (type uns32 ecx),
                "}," nl
            );

        endfor;
        fileio.put
        (
            outFile,
            stdio.tab, stdio.tab, stdio.tab, stdio.tab,
            "{#0..#127}" nl,
            stdio.tab, stdio.tab, stdio.tab, "];" nl
        );


        // Now emit the second table to the file:

        fileio.put
        (
            outFile,
            nl,
            stdio.tab,
            "EndRange: cset[128] := " nl,
            stdio.tab, stdio.tab, stdio.tab, "[" nl
        );
        for( mov( 0, ecx ); ecx < 127; inc( ecx )) do

            fileio.put
            (
                outFile,
                stdio.tab, stdio.tab, stdio.tab, stdio.tab,
                "{#",
                (type uns32 ecx),
                "..#127}," nl
            );

        endfor;
        fileio.put
        (
            outFile,
            stdio.tab, stdio.tab, stdio.tab, stdio.tab, "{#127}" nl,
            stdio.tab, stdio.tab, stdio.tab, "];" nl nl
        );
        fileio.close( outFile );

    end GenerateRangeChar;
```

Program 12.3   Table Generation Program for the csRangeChar Program