Characters and Character Sets Chapter Three

3.1 Chapter Overview

This chapter completes the discussion of the character data type by describing several character translation and classification functions found in the HLA Standard Library. These functions provide most of the character operations that aren't trivial to realize using a few 80x86 machine instructions.

This chapter also introduces another composite data type based on the character – the character set data type. Character sets and their associated operations, let you quickly test characters to see if they belong to some set. These operations also let you manipulate sets of characters using familiar operations like set union, intersection, and difference.

3.2 The HLA Standard Library CHARS.HHF Module

The HLA Standard Library chars.hhf module provides a couple of routines that convert characters from one form to another and several routines that classify characters according to their graphic representation. These functions are especially useful for processing user input to verify that it is correct.

The first two routines we will consider are the translation/conversion functions. These functions are *chars.toUpper* and *chars.toLower.* These functions use the following syntax:

```
chars.toLower( characterValue ); // Returns converted character in AL.
chars.toUpper( characterValue ); // Returns converted character in AL.
```

These two functions require a byte-sized parameter (typically a register or a *char* variable). They check the character to see if it is an alphabetic character; if it is not, then these functions return the unmodified parameter value in the AL register. If the character is an alphabetic character, then these functions may translate the value depending on the particular function. The *chars.toUpper* function translates lower case alphabetic characters to upper case; it returns upper case character unmodified. The chars.toLower function does the converse – it translates upper case characters to lower case characters and leaves lower case characters alone.

These two functions are especially useful when processing user input containing alphabetic characters. For example, suppose you expect a "Y" or "N" answer from the user at some point in your program. You code might look like the following:

```
forever

    stdout.put( "Answer 'Y' or 'N':" );
    stdin.FlushInput();   // Force input of new line of text.
    stdin.getc();  // Read user input in AL.
    breakif( al = 'Y' );
    breakif( al = 'N' );
    stdout.put( "Illegal input, please reenter", nl );

endfor;
```

The problem with this program is that the user must answer exactly "Y" or "N" (using upper case) or the program will reject the user's input. This means that the program will reject "y" and "n" since the ASCII codes for these characters are different than "Y" and "N".

One way to solve this problem is to include two additional BREAKIF statements in the code above that test for "y" and "n" as well as "Y" and "N". The problem with this approach is that AL will still contain one of four different characters, complicating tests of AL once the program exits the loop. A better solution is to use either *chars.toUpper* or *chars.toLower* to translate all alphabetic characters to a single case. Then you can test AL for a single pair of characters, both in the loop and outside the loop. The resulting code would look like the following:

```
forever

    stdout.put( “Answer ‘Y’ or ‘N’:” );
    stdin.FlushInput();   // Force input of new line of text.
    stdin.getc();         // Read user input in AL.
    chars.toUpper( al );  // Convert “y” and “n” to “Y” and “N”.
    breakif( al = ‘Y’ );
    breakif( al = ‘N’ );
    stdout.put( “Illegal input, please reenter”, nl );

endfor;
<< test for “Y” or “N” down here to determine user input >>
```

As you can see from this example, the case conversion functions can be quite useful when processing user input. As a final example, consider a program that presents a menu of options to the user and the user selects an option using an alphabetic character. Once again, you can use *chars.toUpper* or *chars.toLower* to map the input character to a single case so that it is easier to process the user’s input:

```
stdout.put( “Enter selection (A-G):” );
stdin.FlushInput();
stdin.getc();
chars.toLower( al );
if( al = ‘a’ ) then

    << Handle Menu Option A >>

elseif( al = ‘b’ ) then

    << Handle Menu Option B >>

elseif( al = ‘c’ ) then

    << Handle Menu Option C >>

elseif( al = ‘d’ ) then

    << Handle Menu Option D >>

elseif( al = ‘e’ ) then

    << Handle Menu Option E >>

elseif( al = ‘f’ ) then

    << Handle Menu Option F >>

elseif( al = ‘g’ ) then

    << Handle Menu Option G >>

else

    stdout.put( “Illegal input!” nl );

endif;
```

The remaining functions in the chars.hhf module all return a boolean result depending on the type of the character you pass them as a parameter. These classification functions let you quickly and easily test a character to determine if it's type is valid for the some intended use. These functions expect a single byte (char) parameter and they return true (1) or false (0) in the EAX register. These functions use the following calling syntax:

```
chars.isAlpha( c );    // Returns true if c is alphabetic
chars.isUpper( c );    // Returns true if c is upper case alphabetic.
chars.isLower( c );    // Returns true if c is lower case alphabetic.
chars.isAlphaNum( c ); // Returns true if c is alphabetic or numeric.
chars.isDigit( c );    // Returns true if c is a decimal digit.
chars.isXDigit( c );   // Returns true if c is a hexadecimal digit.
chars.isGraphic( c );  // See notes below.
chars.isSpace( c );    // Returns true if c is a whitespace character.
chars.isASCII( c );    // Returns true if c is in the range #$00..#$7f.
chars.isCtrl( c );     // Returns true if c is a control character.
```

Notes: Graphic characters are the printable characters whose ASCII codes fall in the range $21..$7E. Note that a space is not considered a graphic character (nor are the control characters). Whitespace characters are the space, the tab, the carriage return, and the linefeed. Control characters are those characters whose ASCII code is in the range $00..$1F and $7F.

These classification functions are great for validating user input. For example, if you want to check to ensure that a user has entered nothing but numeric characters in a string you read from the standard input, you could use code like the following:

```
stdin.a_gets();  // Read line of text from the user.
mov( eax, ebx ); // save ptr to string in EBX.
mov( ebx, ecx ); // Another copy of string pointer to test each char.
while( (type char [ecx]) <> #0 ) do // Repeat while not at end of string.

    breakif( !chars.isDigit( (type char [ecx] )));
    inc( ecx ); // Move on to the next character;

endwhile;
if( (type char [ecx] ) = #0 ) then

    << Valid string, process it >>

else

    << invalid string >>

endif;
```

Although the chars.hhf module's classification functions handle many common situations, you may find that you need to test a character to see if it belongs in a class that the chars.hhf module does not handle. Fear not, checking for such characters is very easy. The next section will explain how to do this.

3.3 Character Sets

Character sets are another composite data type, like strings, built upon the character data type. A character set is a mathematical set of characters with the most important attribute being membership. That is, a character is either a member of a set or it is not a member of a set. The concept of sequence (e.g., whether one character comes before another, as in a string) is completely foreign to a character set. If two characters are members of a set, their order in the set is irrelevant. Also, membership is a binary relation; a character is either in the set or it is not in the set; you cannot have multiple copies of the same character in a character set. Finally, there are various operations that are possible on character sets including the mathematical set operations of union, intersection, difference, and membership test.

HLA implements a restricted form of character sets that allows set members to be any of the 128 standard ASCII characters (i.e., HLA's character set facilities do not support extended character codes in the range #128..#255). Despite this restriction, however, HLA's character set facilities are very powerful and are very handy when writing programs that work with string data. The following sections describe the implementation and use of HLA's character set facilities so you may take advantage of character sets in your own programs.

3.4 Character Set Implementation in HLA

There are many different ways to represent character sets in an assembly language program. HLA implements character sets by using an array of 128 boolean values. Each boolean value determines whether the corresponding character is or is not a member of the character set; i.e., a true boolean value indicates that the specified character is a member of the set, a false value indicates that the corresponding character is not a member of the set. To conserve memory, HLA allocates only a single bit for each character in the set; therefore, HLA character sets consume 16 bytes of memory since there are 128 bits in 16 bytes. This array of 128 bits is organized in memory as shown in Figure 3.1.

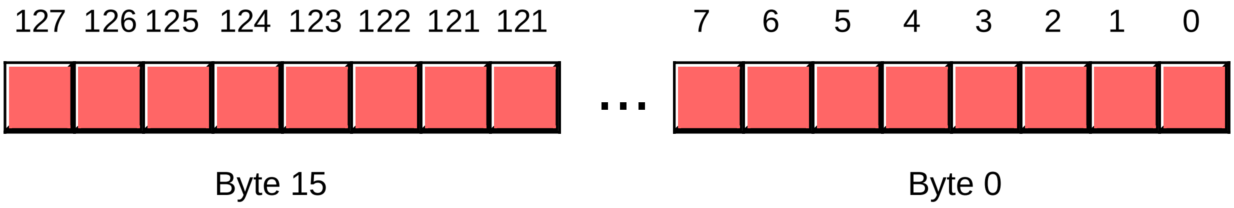

Figure 3.1 Bit Layout of a Character Set Object

Bit zero of byte zero corresponds to ASCII code zero (the NUL character). If this bit is one, then the character set contains the NUL character; if this bit contains false, then the character set does not contain the NUL character. Likewise, bit zero of byte one (the eighth bit in the 128-bit array) corresponds to the backspace character (ASCII code is eight). Bit one of byte eight corresponds to ASCII code 65, an upper case 'A'. Bit 65 will contain a one if 'A' is a current member of the character set, it will contain zero if 'A' is not a member of the set.

While there are other possible ways to implement character sets, this bit vector implementation has the advantage that it is very easy to implement set operations like union, intersection, difference comparison, and membership tests.

HLA supports character set variables using the *cset* data type. To declare a character set variable, you would use a declaration like the following:

```
static
    CharSetVar: cset;
```

This declaration will reserve 16 bytes of storage to hold the 128 bits needed to represent an ASCII character set.

Although it is possible to manipulate the bits in a character set using instructions like AND, OR, XOR, etc., the 80x86 instruction set includes several bit test, set, reset, and complement instructions that are nearly perfect for manipulating character sets. The BT (bit test) instruction, for example will copy a single bit in memory to the carry flag. The BT instruction allows the following syntactical forms:

bt(*BitNumber*, *BitsToTest*);

bt(reg_{16}, reg_{16});

```
        bt( reg32, reg32 );
        bt( constant, reg16 );
        bt( constant, reg32 );

        bt( reg16, mem16 );
        bt( reg32, mem32 );  //HLA treats cset objects as dwords within bt.
        bt( constant, mem16 );
        bt( constant, mem32 );  //HLA treats cset objects as dwords within bt.
```

The first operand holds a bit number, the second operand specifies a register or memory location whose bit should be copied into the carry flag. If the second operand is a register, the first operand must contain a value in the range 0..*n*-1, where *n* is the number of bits in the second operand. If the first operand is a constant and the second operand is a memory location, the constant must be in the range 0..255. Here are some examples of these instructions:

```
    bt( 7, ax );             // Copies bit #7 of AX into the carry flag (CF).
    mov( 20, eax );
    bt( eax, ebx );          // Copies bit #20 of EBX into CF.

// Copies bit #0 of the byte at CharSetVar+3 into CF.

    bt( 24, CharSetVar );

    // Copies bit #4 of the byte at DWmem+2 into CF.

    bt( eax, CharSetVar );
```

The BT instruction turns out to be quite useful for testing set membership. For example, to see if the character ‘A’ is a member of a character set, you could use a code sequence like the following:

```
        bt( ‘A’, CharSetVar );
        if( @c ) then

            << Do something if ‘A’ is a member of the set >>

        endif;
```

The BTS (bit test and set), BTR (bit test and reset), and BTC (bit test and complement) instructions are also quite useful for manipulating character set variables. Like the BT instruction, these instructions copy the specified bit into the carry flag; after copying the specified bit, these instructions will set, clear, or invert (respectively) the specified bit. Therefore, you can use the BTS instruction to add a character to a character set via set union (that is, it adds a character to the set if the character was not already a member of the set, otherwise the set is unaffected). You can use the BTR instruction to remove a character from a character set via set intersection (That is, it removes a character from the set if and only if it was previously in the set; otherwise it has no effect on the set). The BTC instruction lets you add a character to the set if it wasn’t previously in the set, it removes the character from the set if it was previously a member (that is, it toggles the membership of that character in the set).

The HLA Standard Library provides lots of character set handling routines. See “Character Set Support in the HLA Standard Library” on page 445. for more details about HLA’s character set facilities.

3.5 HLA Character Set Constants and Character Set Expressions

HLA supports literal character set constants. These *cset* constants make it easy to initialize *cset* variables at compile time and they make it very easy to pass character set constants as procedure parameters. An HLA character set constant takes the following form:

```
            { Comma_separated_list_of_characters_and_character_ranges }
```

The following is an example of a simple character set holding the numeric digit characters:

```
{ ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’ }
```

When specifying a character set literal that has several contiguous values, HLA lets you concisely specify the values using only the starting and ending values of the range thusly:

```
{ ‘0’..’9’ }
```

You may combine characters and various ranges within the same character set constant. For example, the following character set constant is all the alphanumeric characters:

```
{ ‘0’..’9’, ‘a’..’z’, ‘A’..’Z’ }
```

You can use these *cset* literal constants in the CONST and VAL sections. The following example demonstrates how to create the symbolic constant *AlphaNumeric* using the character set above:

```
const
    AlphaNumeric: cset := {‘0’..’9’, ‘a’..’z’, ‘A’..’Z’ };
```

After the above declaration, you can use the identifier *AlphaNumeric* anywhere the character set literal is legal.

You can also use character set literals (and, of course, character set symbolic constants) as the initializer field for a STATIC or READONLY variable. The following code fragment demonstrates this:

```
static
    Alphabetic: cset := { ‘a’..’z’, ‘A’..’Z’ };
```

Anywhere you can use a character set literal constant, a character set constant expression is also legal. HLA supports the following operators in character set constant expressions:

CSetConst + *CSetConst*	Computes the union of the two sets[1].
CSetConst * *CSetConst*	Computes the intersection of the two sets[2].
CSetConst - *CSetConst*	Computes the set difference of the two sets[3].
-*CSetConst*	Computes the set complement[4].

Note that these operators only produce compile-time results. That is, the expressions above are computed by the compiler during compilation, they do not emit any machine code. If you want to perform these operations on two different sets while your program is running, the HLA Standard Library provides routines you can call to achieve the results you desire. HLA also provides other compile-time character set operators. See the chapter on the compile-time language and macros for more details.

3.6 The IN Operator in HLA HLL Boolean Expressions

The HLA IN operator can dramatically reduce the logic in your HLA programs. This text has waited until now to discuss this operator because certain forms require a knowledge of character sets and character set constants. Now that you’ve seen character set constants, there is no need to delay the introduction of this important language feature.

In addition to the standard boolean expressions in IF, WHILE, REPEAT..UNTIL, and other statements, HLA also supports boolean expressions that take the following forms:

reg_8 in *CSetConstant*

[1] The set union is the set of all characters that are in either set.

[2] The set intersection is the set of all characters that appear in both operand sets.

[3] The set difference is the set of characters that appear in the first set but do not appear in the second set.

[4] The set complement is the set of all characters not in the set.

reg_8 not in *CSetConstant*
reg_8 in *CSetVariable*
reg_8 not in *CSetVariable*

These four forms of the IN and NOT IN operators check to see if a character in an eight-bit register is a member of a character set (either a character set constant or a character set variable). The following code fragment demonstrates these operators:

```
const
    Alphabetic: cset := {‘a’..’z’, ‘A’..’Z’};
        .
        .
        .
    stdin.getc();
    if( al in Alphabetic ) then

        stdout.put( “You entered an alphabetic character” nl );

    elseif( al in {‘0’..’9’} ) then

        stdout.put( “You entered a numeric character” nl );

    endif;
```

3.7 Character Set Support in the HLA Standard Library

As noted in the previous sections, the HLA Standard Library provides several routines that provide character set support. The character set support routines fall into four categories: standard character set functions, character set tests, character set conversions, and character set I/O. This section describes these routines in the HLA Standard Library.

To begin with, let’s consider the Standard Library routines that help you construct character sets. These routines include: *cs.empty*, *cs.cpy*, *cs.charToCset*, *cs.unionChar*, *cs.removeChar*, *cs.rangeChar*, *cs.strToCset*, and *cs.unionStr*. These procedures let you build up character sets at run-time using character and string objects.

The *cs.empty* procedure initializes a character set variable to the empty set by setting all the bits in the character set to zero. This procedure call uses the following syntax (*CSvar* is a character set variable):

```
cs.empty( CSvar );
```

The *cs.cpy* procedure copies one character set to another, replacing any data previously held by the destination character set. The syntax for *cs.cpy* is

```
cs.cpy( srcCsetValue, destCsetVar );
```

The *cs.cpy* source character set can be either a character set constant or a character set variable. The destination character set must be a character set variable.

The *cs.unionChar* procedure adds a character to a character set. It uses the following calling sequence:

```
cs.unionChar( CharVar, CSvar );
```

This call will add the first parameter, a character, to the set via set union. Note that you could use the BTS instruction to achieve this same result although the *cs.unionChar* call is often more convenient (though slower).

The *cs.charToCset* function creates a singleton set (a set containing a single character). The calling format for this function is

```
cs.charToCset( CharValue, CSvar );
```

The first operand, the character value *CharValue*, can be an eight-bit register, a constant, or a character variable. The second operand (*CSvar*) must be a character set variable. This function clears the destination character set to all zeros and then adds the specified character to the character set.

The *cs.removeChar* procedure lets you remove a single character from a character set without affecting the other characters in the set. This function uses the same syntax as *cs.charToCset* and the parameters have the same attributes. The calling sequence is

```
cs.removeChar( CharValue, CSVar );
```

The *cs.rangeChar* constructs a character set containing all the characters between two characters you pass as parameters. This function sets all bits outside the range of these two characters to zero. The calling sequence is

```
cs.rangeChar( LowerBoundChar, UpperBoundChar, CSVar );
```

The *LowerBoundChar* and *UpperBoundChar* parameters can be constants, registers, or character variables. *CSVar*, the destination character set, must be a *cset* variable.

The *cs.strToCset* procedure creates a new character set containing the union of all the characters in a character string. This procedure begins by setting the destination character set to the empty set and then it unions in the characters in the string one by one until it exhausts all characters in the string. The calling sequence is

```
cs.strToCset( StringValue, CSVar );
```

Technically, the *StringValue* parameter can be a string constant as well as a string variable, however, it doesn't make any sense to call *cs.strToCset* in this fashion since *cs.cpy* is a much more efficient way to initialize a character set with a constant set of characters. As usual, the destination character set must be a *cset* variable. Typically, you'd use this function to create a character set based on a string input by the user.

The *cs.unionStr* procedure will add the characters in a string to an existing character set. Like *cs.strToCset*, you'd normally use this function to union characters into a set based on a string input by the user. The calling sequence for this is

```
cs.unionStr( StringValue, CSVar );
```

Standard set operations include union, intersection, and set difference. The HLA Standard Library routines *cs.setunion*, *cs.intersection*, and *cs.difference* provide these operations, respectively[5]. These routines all use the same calling sequence:

```
 cs.setunion( srcCset, destCset );
cs.intersection( srcCset, destCset );
 cs.difference( srcCset, destCset );
```

The first parameter can be a character set constant or a character set variable. The second parameter must be a character set variable. These procedures compute "destCset := destCset op srcCset" where *op* represents set union, intersection, or difference, depending on the function call.

The third category of character set routines test character sets in various ways. They typically return a boolean value indicating the result of the test. The HLA character set routines in this category include *cs.IsEmpty, cs.member, cs.subset, cs.psubset, cs.superset, cs.psuperset, cs.eq,* and *cs.ne*.

The *cs.IsEmpty* function tests a character set to see if it is the empty set. The function returns true or false in the EAX register. This function uses the following calling sequence:

```
cs.IsEmpty( CSetValue );
```

[5] "cs.setunion" was used rather than "cs.union" because "union" is an HLA reserved word.

The single parameter may be a constant or a character set variable, although it doesn't make much sense to pass a character set constant to this procedure (since you would know at compile-time whether this set is empty or not empty).

The *cs.member* function tests to see if a character value is a member of a set. This function returns true in the EAX register if the supplied character is a member of the specified set. Note that you can use the BT instruction to (more efficiently) test this same condition. However, the *cs.member* function is probably a little more convenient to use. The calling sequence for *cs.member* is

```
cs.member( CharValue, CsetValue );
```

The first parameter is a register, character variable, or a constant. The second parameter is either a character set constant or a character set variable. It would be unusual for both parameters to be constants.

The *cs.subset*, *cs.psubset* (proper subset), *cs.superset*, and *cs.psuperset* (proper superset) functions let you check to see if one character set is a subset or superset of another. The calling sequence for these four routines is nearly identical, it is one of the following:

```
 cs.subset( CsetValue1, CsetValue2 );
 cs.psubset( CsetValue1, CsetValue2 );
cs.superset( CsetValue1, CsetValue2 );
cs.psuperset( CsetValue1, CsetValue2 );
```

These routines compare the first parameter against the second parameter and return true or false in the EAX register depending upon the result of the comparison. One set is a subset of another if all the members of the first character set can be found in the second character set. It is a proper subset if the second character set also contains characters not found in the first (left) character set. Likewise, one character set is a superset of another if it contains all the characters in the second (right) set (and, possibly, more). A proper superset contains additional characters above and beyond those found in the second set. The parameters can be either character set variables or character set constants; however, it would be unusual for both parameters to be character set constants (since you can determine this at compile time, there would be no need to call a run-time function to compute this).

The *cs.eq* and *cs.ne* check to see if two sets are equal or not equal. These functions return true or false in EAX depending upon the set comparison. The calling sequence is identical to the sub/superset functions above:

```
cs.eq( CsetValue1, CsetValue2 );
cs.ne( CsetValue1, CsetValue2 );
```

The *cs.extract* routine removes an arbitrary character from a character set and returns that character in the EAX register[6]. The calling sequence is the following:

```
cs.extract( CsetVar );
```

The single parameter must be a character set variable. Note that this function will modify the character set variable by removing some character from the character set. This function returns $FFFF_FFFF (-1) in EAX if the character set was empty prior to the call.

In addition to the routines found in the *cs* (character set) library module, the string and standard output modules also provide functions that allow or expect character set parameters. For example, if you supply a character set value as a parameter to *stdout.put*, the *stdout.put* routine will print the characters currently in the set. See the HLA Standard Library documentation for more details on character set handling procedures.

3.8 Using Character Sets in Your HLA Programs

Character sets are valuable for many different applications in your programs. For example, in the volume on Advanced String Handling you'll discover how to use character sets to match complex patterns.

[6] This routine returns the character in AL and zeros out the H.O. three bytes of EAX.

However, such use of character sets is a little beyond the scope of this chapter, so at this point we'll concentrate on another common use of character sets: validating user input. This section will also present a couple of other applications for character sets to help you start thinking about how you could use them in your program.

Consider the following short code segment that gets a yes/no type answer from the user:

```
static
    answer: char;
        .
        .
        .
    repeat
            .
            .
            .
        stdout.put( "Would you like to play again? " );
        stdin.FlushInput();
        stdin.get( answer );

    until( answer = 'n' );
```

A major problem with this code sequence is that it will only stop if the user presses a lower case 'n' character. If they type anything other than 'n' (including upper case 'N') the program will treat this as an affirmative answer and transfer back to the beginning of the repeat..until loop. A better solution would be to validate the user input before the UNTIL clause above to ensure that the user has only typed "n", "N", "y", or "Y". The following code sequence will accomplish this:

```
    repeat
            .
            .
            .
        repeat

            stdout.put( "Would you like to play again? " );
            stdin.FlushInput();
            stdin.get( answer );

        until( cs.member( answer, { 'n', 'N', 'Y', 'y' } );
        if( answer = 'N' ) then

            mov( 'n', answer );

        endif;

    until( answer = 'n' );
```

While an excellent use for character sets is to validate user input, especially when you must restrict the user to a small set of non-contiguous input characters, you should not use the *cs.member* function to test to see if a character value is within literal set. For example, you should never do something like the following:

```
    repeat

        stdout.put( "Enter a character between 0..9: " );
        stdin.getc();

    until( cs.member( al, {'0'..'9' } );
```

While there is nothing logically wrong with this code, keep in mind that HLA run-time boolean expressions allow simple membership tests using the IN operator. You could write the code above far more efficiently using the following sequence:

```
repeat

    stdout.put( “Enter a character between 0..9: “ );
    stdin.getc();

until( al in ‘0’..’9’ );
```

The place where the *cs.member* function becomes useful is when you need to see if an input character is within a set of characters that you build at run time.

3.9 Low-level Implementation of Set Operations

Although the HLA Standard Library character set module simplifies the use of character sets within your assembly language programs, it is instructive to look at how all these functions operate so you know the cost associated with each function. Also, since many of these functions are quite trivial, you might want to implement them in-line for performance reasons. The following subsections describe how each of the functions operate.

3.9.1 Character Set Functions That Build Sets

The first group of functions we will look at in the Character Set module are those that construct or copy character sets. These functions are *cs.empty, cs.cpy, cs.charToCset, cs.unionChar, cs.removeChar, cs.rangeChar, cs.strToCset,* and *cs.unionStr.*

Creating an empty set is, perhaps, the easiest of all the operations. To create an empty set all we need to is zero out all 128 bits in the *cset* object. Program 3.1 provides the implementation of this function.

```
// Program that demonstrates the implmentation of
// the cs.empty function.

program csEmpty;
#include( “stdlib.hhf” )

static
    csetDest: cset;
    csetSrc: cset := {‘a’..’z’, ‘A’..’Z’};

begin csEmpty;

    // How to create an empty set (cs.empty):
    // (Zero out all bits in the cset)

    mov( 0, eax );
    mov( eax, (type dword csetDest ));
    mov( eax, (type dword csetDest[4] ));
    mov( eax, (type dword csetDest[8] ));
    mov( eax, (type dword csetDest[12] ));

    stdout.put( “Empty set = {“, csetDest, “}” nl );

end csEmpty;
```

Program 3.1 cs.empty Implementation

Note that cset objects are 16 bytes long. Therefore, this code zeros out those 16 bytes by storing EAX into the four consecutive double words that comprise the object. Note the use of type coercion in the MOV statements; this is necessary since *cset* objects are not the same size as *dword* objects.

To copy one character set to another is only a little more difficult than creating an empty set. All we have to do is copy the 16 bytes from the source character set to the destination character set. We can accomplish this with four pairs of double word MOV statements. Program 3.2 provides the sample implementation.

```
// Program that demonstrates the implmentation of
// the cs.empty function.

program csCpy;
#include( “stdlib.hhf” )

static
    csetDest: cset;
    csetSrc: cset := {‘a’..’z’, ‘A’..’Z’};

begin csCpy;

    // How to create an empty set (cs.empty):
    // (Zero out all bits in the cset)

    mov( (type dword csetSrc), eax );
    mov( eax, (type dword csetDest ));

    mov( (type dword csetSrc[4]), eax );
    mov( eax, (type dword csetDest[4] ));

    mov( (type dword csetSrc[8]), eax );
    mov( eax, (type dword csetDest[8] ));

    mov( (type dword csetSrc[12]), eax );
    mov( eax, (type dword csetDest[12] ));

    stdout.put( “Copied set = {“, csetDest, “}” nl );

end csCpy;
```

Program 3.2 cs.cpy Implementation

The *cs.charToCset* function creates a singleton set containing the specified character. To implement this function we first begin by creating an empty set (using the same code as cs.empty) and then we set the bit corresponding to the single character in the character set. We can use the BTS (bit test and set) instruction to easily set the specified bit in the cset object. Program 3.3 provides the implementation of this function.

```
// Program that demonstrates the implmentation of
// the cs.charToCset function.

program cscharToCset;
#include( “stdlib.hhf” )

static
```

```
    csetDest: cset;
    chrValue: char := ‘a’;

begin cscharToCset;

    // Begin by creating an empty set:

    mov( 0, eax );
    mov( eax, (type dword csetDest ));
    mov( eax, (type dword csetDest[4] ));
    mov( eax, (type dword csetDest[8] ));
    mov( eax, (type dword csetDest[12] ));

    // Okay, use the BTS instruction to set the specified bit in
    // the character set.

    movzx( chrValue, eax );
    bts( eax, csetDest );

    stdout.put( “Singleton set = {“, csetDest, “}” nl );

end cscharToCset;
```

Program 3.3 cs.charToCset Implementation

If you study this code carefully, you will note an interesting fact: the BTS instruction’s operands are not the same size (*dword* and *cset*). Since programmers often use the BTx instructions to manipulate items in a character set, HLA allows you to specify a *cset* object as the destination operand of a BTx(reg_{32}, mem) instruction. Technically, the memory operand should be a double word object; HLA automatically coerces *cset* objects to *dword* for these instructions. Note that BTS requires a 16 or 32-bit register. Therefore, this code zero extends the character’s value into EAX prior to executing the BTS instruction. Note that the value in EAX must not exceed 127 or this code will manipulate data beyond the end of the character set in memory. The use of a BOUND instruction might be warranted here if you can’t ensure that the *chrValue* variable contains a value in the range 0..127.

The *cs.unionChar* adds a single character to the character set (if that character was not already present in the character set). This code is actually a bit simpler than the cs.charToCset function; the only difference is that the code does not clear the set to begin with – it simply sets the bit corresponding to the given character. Program 3.4 provides the implementation.

```
// Program that demonstrates the implmentation of
// the cs.unionChar function.

program csUnionChar;
#include( “stdlib.hhf” )

static
    csetDest: cset := {‘0’..’9’};
    chrValue: char := ‘a’;

begin csUnionChar;


    // Okay, use the BTS instruction to add the specified bit to
    // the character set.
    movzx( chrValue, eax );
    bts( eax, csetDest );

    stdout.put( “New set = {“, csetDest, “}” nl );

end csUnionChar;
```

Program 3.4 cs.unionChar Implementation

Once again, note that this code assumes that the character value is in the range 0..127. If it is possible for the character to fall outside this range, you should check the value before attempting to union the character into the character set. You can use an IF statement or the BOUND instruction for this check.

The *cs.removeChar* function removes a character from a character set, provided that character was a member of the set. If the character was not originally in the character set, then *cs.removeChar* does not affect the original character set. To accomplish this, the code must clear the bit associated with the character to remove from the set. Program 3.5 uses the BTR (bit test and reset) instruction to achieve this.

```
// Program that demonstrates the implmentation of
// the cs.removeChar function.

program csRemoveChar;
#include( “stdlib.hhf” )

static
    csetDest: cset := {‘0’..’9’};
    chrVal1: char := ‘0’;
    chrVal2: char := ‘a’;

begin csRemoveChar;


    // Okay, use the BTC instruction to remove the specified bit from
    // the character set.

    movzx( chrVal1, eax );
    btr( eax, csetDest );

    stdout.put( “Set w/o ‘0’ = {“, csetDest, “}” nl );

    // Now remove a character not in the set to demonstrate
    // that removal of a non-existant character doesn’t affect
    // the set:

    movzx( chrVal2, eax );
    btr( eax, csetDest );
    stdout.put( “Final set = {“, csetDest, “}” nl );

end csRemoveChar;
```

Program 3.5 cs.removeChar Implementation

Don’t forget to use a BOUND instruction or an IF statement in Program 3.5 if it is possible for the character’s value to fall outside the range 0..127. This will prevent the code from manipulating memory beyond the end of the character set.

The *cs.rangeChar* function creates a set containing all the characters between two specified boundaries. This function begins by creating an empty set; then it loops over the range of character to insert, inserting each character into the set as appropriate. Program 3.6 provides an example implementation of this function.

```
// Program that demonstrates the implmentation of
// the cs.rangeChar function.

program csRangeChar;
#include( “stdlib.hhf” )

static
    csetDest: cset;
    startRange: char := ‘a’;
    endRange: char := ‘z’;

begin csRangeChar;


    // Begin by creating the empty set:

    mov( 0, eax );
    mov( eax, (type dword csetDest ));
    mov( eax, (type dword csetDest[4] ));
    mov( eax, (type dword csetDest[8] ));
    mov( eax, (type dword csetDest[12] ));

    // Run the following loop for each character between
    // ‘startRange’ and ‘endRange’ and set the corresponding
    // bit in the cset for each character in the range.

    movzx( startRange, eax );
    while( al <= endRange ) do

        bts( eax, csetDest );
        inc( al );

    endwhile;
        stdout.put( “Final set = {“, csetDest, “}” nl );

end csRangeChar;
```

Program 3.6 cs.rangeChar Implementation

One interesting thing to note about the code in Program 3.6 is how it takes advantage of the fact that AL contains the actual character index even though it has to use EAX with the BTS instruction. As usual, you should check the range of the two values if there is any possibility that they could be outside the range 0..127.

One problem with this particular implementation of the *cs.rangeChar* function is that it is not particularly efficient if you ask it to create a set with a lot of characters in it. As you can see by studying the code, the execution time of this function is proportional to the number of characters in the range. In particular, the loop in this function iterates once for each character in the range. So if the range is large the loop executes many more times than if the range is small. There is a more efficient solution to this problem using table lookups whose execution time is independent of the size of the character set it creates. For more details on using table lookups, see “Calculation Via Table Lookups” on page 647.

The *cs.strToCset* function scans through an HLA string character by character and creates a new character set by adding each character in the string to an empty character set.

```
// Program that demonstrates the implmentation of
// the cs.strToCset function.

program csStrToCset;
#include( “stdlib.hhf” )

static
    StrToAdd: string := “Hello_World”;
    csetDest: cset;

begin csStrToCset;


    // Begin by creating the empty set:

    mov( 0, eax );
    mov( eax, (type dword csetDest ));
    mov( eax, (type dword csetDest[4] ));
    mov( eax, (type dword csetDest[8] ));
    mov( eax, (type dword csetDest[12] ));

    // For each character in the source string, add that character
    // to the set.

    mov( StrToAdd, eax );
    while( (type char [eax]) <> #0 ) do  // While not at end of string.

        movzx( (type char [eax]), ebx );
        bts( ebx, csetDest );
        inc( eax );

    endwhile;
    stdout.put( “Final set = {“, csetDest, “}” nl );

end csStrToCset;
```

Program 3.7 cs.strToCset Implementation

This code begins by fetching the pointer to the first character in the string. The loop repeats for each character in the string up to the zero terminating byte of the string. For each character, this code uses the BTS instruction to set the corresponding bit in the destination character set. As usual, don’t forget to use an IF statement or BOUND instruction if it is possible for the characters in the string to have values outside the range 0..127.

The *cs.unionStr* function is very similar to the *cs.strToCset* function; in fact, the only difference is that it doesn’t create an empty character set prior to adding the characters in a string to the destination character set.

```
// Program that demonstrates the implmentation of
// the cs.unionStr function.

program csUnionStr;
#include( “stdlib.hhf” )

static
    StrToAdd: string := “Hello_World”;
    csetDest: cset := {‘0’..’9’};

begin csUnionStr;



    // For each character in the source string, add that character
    // to the set.

    mov( StrToAdd, eax );
    while( (type char [eax]) <> #0 ) do  // While not at end of string.

        movzx( (type char [eax]), ebx );
        bts( ebx, csetDest );
        inc( eax );

    endwhile;
    stdout.put( “Final set = {“, csetDest, “}” nl );

end csUnionStr;
```

Program 3.8 cs.unionStr Implementation

3.9.2 Traditional Set Operations

The previous section describes how to construct character sets from characters and strings. In this section we’ll take a look at how you can manipulate character sets using the traditional set operations of intersection, union, and difference.

The union of two sets A and B is the collection of all items that are in set A, set B, or both. In the bit array representation of a set, this means that a bit in the destination character set will be one if either or both of the corresponding bits in sets A or B are set. This of course, corresponds to the logical OR operation. Therefore, we can easily create the set union of two sets by logically ORing their bytes together. Program 3.9 provides the complete implementation of this function.

```
// Program that demonstrates the implmentation of
// the cs.setunion function.

program cssetUnion;
#include( “stdlib.hhf” )

static
    csetSrc1: cset := {‘a’..’z’};
    csetSrc2: cset := {‘A’..’Z’};
    csetDest: cset;

begin cssetUnion;
```

```
    // To compute the union of csetSrc1 and csetSrc2 all we have
    // to do is logically OR the two sets together.

    mov( (type dword csetSrc1), eax );
    or( (type dword csetSrc2), eax );
    mov( eax, (type dword csetDest));

    mov( (type dword csetSrc1[4]), eax );
    or( (type dword csetSrc2[4]), eax );
    mov( eax, (type dword csetDest[4]));

    mov( (type dword csetSrc1[8]), eax );
    or( (type dword csetSrc2[8]), eax );
    mov( eax, (type dword csetDest[8]));

    mov( (type dword csetSrc1[12]), eax );
    or( (type dword csetSrc2[12]), eax );
    mov( eax, (type dword csetDest[12]));

    stdout.put( "Final set = {", csetDest, "}" nl );

end cssetUnion;
```

Program 3.9 cs.setunion Implementation

The intersection of two sets is those elements that are members of both sets. In the bit array representation of character sets that HLA uses, this means that a bit is set in the destination character set if the corresponding bit is set in both the source sets; this corresponds to the logical AND operation; therefore, to compute the set intersection of two character sets, all you need do is logically AND the 16 bytes of the two source sets together. Program 3.10 provides a sample implementation.

```
// Program that demonstrates the implmentation of
// the cs.intersection function.

program csIntersection;
#include( "stdlib.hhf" )

static
    csetSrc1: cset := {'a'..'z'};
    csetSrc2: cset := {'A'..'z'};
    csetDest: cset;

begin csIntersection;

    // To compute the intersection of csetSrc1 and csetSrc2 all we have
    // to do is logically AND the two sets together.

    mov( (type dword csetSrc1), eax );
    and( (type dword csetSrc2), eax );
    mov( eax, (type dword csetDest));

    mov( (type dword csetSrc1[4]), eax );
    and( (type dword csetSrc2[4]), eax );
    mov( eax, (type dword csetDest[4]));
    mov( (type dword csetSrc1[8]), eax );
    and( (type dword csetSrc2[8]), eax );
    mov( eax, (type dword csetDest[8]));

    mov( (type dword csetSrc1[12]), eax );
    and( (type dword csetSrc2[12]), eax );
    mov( eax, (type dword csetDest[12]));

    stdout.put( “ Set A = {“, csetSrc1, “}” nl );
    stdout.put( “ Set B = {“, csetSrc2, “}” nl );
    stdout.put( “Intersection of A and B = {“, csetDest, “}” nl );

end csIntersection;
```

Program 3.10 cs.intersection Implementation

The difference of two sets is all the elements in the first set that are not also present in the second set. To compute this result we must logically AND the values from the first set with the inverted values of the second set; i.e., to compute C := A - B we use the following expression:

```
C := A and (not B);
```

Program 3.11 provides the code to implement this operation.

```
// Program that demonstrates the implmentation of
// the cs.difference function.

program csDifference;
#include( “stdlib.hhf” )

static
    csetSrc1: cset := {‘0’..’9’, ‘a’..’z’};
    csetSrc2: cset := {‘A’..’z’};
    csetDest: cset;

begin csDifference;

    // To compute the difference of csetSrc1 and csetSrc2 all we have
    // to do is logically AND A and NOT B together.

    mov( (type dword csetSrc2), eax );
    not( eax );
    and( (type dword csetSrc1), eax );
    mov( eax, (type dword csetDest));

    mov( (type dword csetSrc2[4]), eax );
    not( eax );
    and( (type dword csetSrc1[4]), eax );
    mov( eax, (type dword csetDest[4]));

    mov( (type dword csetSrc2[8]), eax );
    not( eax );
    and( (type dword csetSrc1[8]), eax );
    mov( eax, (type dword csetDest[8]));

    mov( (type dword csetSrc2[12]), eax );
    not( eax );
    and( (type dword csetSrc1[12]), eax );
    mov( eax, (type dword csetDest[12]));

    stdout.put( " Set A = {", csetSrc1, "}" nl );
    stdout.put( " Set B = {", csetSrc2, "}" nl );
    stdout.put( "Difference of A and B = {", csetDest, "}" nl );

end csDifference;
```

Program 3.11 cs.difference Implementation

3.9.3 Testing Character Sets

In addition to manipulating the members of a character set, the need often arises to compare character sets, check to see if a character is a member of a set, and check to see if a set is empty. In this section we'll discuss how HLA implements the relational operations on character sets.

Occasionally you'll want to check a character set to see if it contains any members. Although you could achieve this by creating a static *cset* variable with no elements and comparing the set in question against this empty set, there is a more efficient way to do this – just check to see if all the bits in the set in question are zero. An easy way to do this, that uses, is to logically OR the four double words in a cset object together. If the result is zero, then all the bits in the *cset* variable are zero and, hence, the character set is empty.

```
// Program that demonstrates the implmentation of
// the cs.IisEmpty function.

program csIsEmpty;
#include( "stdlib.hhf" )

static
    csetSrc1: cset := {};
    csetSrc2: cset := {'A'..'Z'};

begin csIsEmpty;

    // To see if a set is empty, simply OR all the dwords
    // together and see if the result is zero:

    mov( (type dword csetSrc1[0]), eax );
    or( (type dword csetSrc1[4]), eax );
    or( (type dword csetSrc1[8]), eax );
    or( (type dword csetSrc1[12]), eax );

    if( @z ) then

        stdout.put( "csetSrc1 is empty ({", csetSrc1, "})" nl );

    else

        stdout.put( "csetSrc1 is not empty ({", csetSrc1, "})" nl );

    endif;

    // Repeat the test for csetSrc2:
```

```
    mov( (type dword csetSrc2[0]), eax );
    or( (type dword csetSrc2[4]), eax );
    or( (type dword csetSrc2[8]), eax );
    or( (type dword csetSrc2[12]), eax );

    if( @z ) then

        stdout.put( “csetSrc2 is empty ({“, csetSrc2, “})” nl );

    else

        stdout.put( “csetSrc2 is not empty ({“, csetSrc2, “})” nl );

    endif;

end csIsEmpty;
```

Program 3.12 Implementation of cs.IsEmpty

Perhaps the most common check on a character set is set membership; that is, checking to see if some character is a member of a given character set. As you’ve seen already (see “Character Set Implementation in HLA” on page 442), the BT instruction is perfect for this. Since we’ve already discussed how to use the BT instruction (along with, perhaps, a MOVZX instruction), there is no need to repeat the implementation of this operation here.

Two sets are equal if and only if all the bits are equal in the two set objects. Therefore, we can implement the cs.ne and cs.eq (set inequality and set equality) functions by comparing the four double words in a *cset* object and noting if there are any differences. Program 3.13 demonstrates how you can do this.

```
// Program that demonstrates the implmentation of
// the cs.eq and cs.ne functions.

program cseqne;
#include( “stdlib.hhf” )

static
    csetSrc1: cset := {‘a’..’z’};
    csetSrc2: cset := {‘a’..’z’};
    csetSrc3: cset := {‘A’..’Z’};

begin cseqne;

    // To see if a set equal to another, check to make sure
    // all four dwords are equal.  One sneaky way to do this
    // is to use the XOR operator (XOR is “not equals” as you
    // may recall).

    mov( (type dword csetSrc1[0]), eax ); // Set EAX to zero if these
    xor( (type dword csetSrc2[0]), eax ); //  two dwords are equal
    mov( eax, ebx );  // Accumulate result here.

    mov( (type dword csetSrc1[4]), eax );
    xor( (type dword csetSrc2[4]), eax );
    or( eax, ebx );
```

```
        mov( (type dword csetSrc1[8]), eax );
        xor( (type dword csetSrc2[8]), eax );
        or( eax, ebx );

        mov( (type dword csetSrc1[12]), eax );
        xor( (type dword csetSrc2[12]), eax );
        or( eax, ebx );

        // At this point, EBX is zero if the two csets are equal
        // (also, the zero flag is set if they are equal).

        if( @z ) then

            stdout.put( “csetSrc1 is equal to csetSrc2” nl );

        else

            stdout.put( “csetSrc1 is not equal to csetSrc2” nl );

        endif;

        // Implementation of cs.ne:

        mov( (type dword csetSrc1[0]), eax ); // Set EAX to zero if these
        xor( (type dword csetSrc3[0]), eax ); //  two dwords are equal
        mov( eax, ebx );  // Accumulate result here.

        mov( (type dword csetSrc1[4]), eax );
        xor( (type dword csetSrc3[4]), eax );
        or( eax, ebx );

        mov( (type dword csetSrc1[8]), eax );
        xor( (type dword csetSrc3[8]), eax );
        or( eax, ebx );

        mov( (type dword csetSrc1[12]), eax );
        xor( (type dword csetSrc3[12]), eax );
        or( eax, ebx );

        // At this point, EBX is non-zero if the two csets are not equal
        // (also, the zero flag is clear if they are not equal).

        if( @nz ) then

            stdout.put( “csetSrc1 is not equal to csetSrc3” nl );

        else

            stdout.put( “csetSrc1 is equal to csetSrc3” nl );

        endif;


    end cseqne;
```

Program 3.13 Implementation of cs.ne and cs.eq

The remaining tests on character sets roughly correspond to tests for less than or greater than; though in set theory we refer to these as superset and subset. One set is a subset of another if the second set contains all the elements of the first set; the second set may contain additional elements. The subset relationship is roughly equivalent to "less than or equal." The proper subset relation of two sets states that the elements of one set are all present in a second set and the two sets are not equal (i.e., the second set contains additional elements). This is roughly equivalent to the "less than" relationship.

Testing for a subset is an easy task. All you have to do is take the set intersection of the two sets and verify that the intersection of the two is equal to the first set. That is, A <= B if and only if:

```
A == ( A * B )  "*" denotes set intersection
```

Testing for a proper subset is a little more work. The same relationship above must hold but the resulting inspection must not be equal to B. That is, A < B if and only if,

```
(A == ( A * B )) and (B <> ( A * B ))
```

The algorithms for superset and proper superset are nearly identical. They are:

```
B == ( A * B )                          A >= B
(B == ( A * B )) and (A <> ( A * B ))   A >  B
```

The implementation of these four relational operations is left as an exercise.

3.10 Putting It All Together

This chapter describes HLA's implementation of character sets. Character sets are a very useful tool for validating user input and for other character scanning and manipulation operations. HLA uses a bit array implementation for character set objects. HLA's implementation allows for 128 different character values in a character set.

The HLA Standard Library provides a wide set of functions that let you build, manipulate, and compare character sets. Although these functions are convenient to use, most of the character set operations are so simple that you can implement them directly using in-line code. This chapter provided the implementation of many of the HLA Standard Library character set functions.

Note that this chapter does not cover all the uses of character sets in an assembly language program. In the volume on "Advanced String Handling" you will see many more uses for character sets in your programs.