HLA Strings and the HLA String Library   Chapter One

1.1 Chapter Overview

A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes, words, or (on 80386 and later processors) double words. The 80x86 microprocessor family supports several instructions specifically designed to cope with strings. This chapter explores some of the uses of these string instructions.

The 80x86 CPUs can process three types of strings: byte strings , word strings, and double word strings. They can move strings, compare strings, search for a specific value within a string, initialize a string to a fixed value, and do other primitive operations on strings. The 80x86's string instructions are also useful for manipulating arrays, tables, and records. You can easily assign or compare such data structures using the string instructions. Using string instructions may speed up your array manipulation code considerably.

1.2 Character Strings

Since you'll encounter character strings more often than other types of strings, they deserve special attention. The following paragraphs describe character strings and various types of string operations.

At the most basic level, the 80x86's string instruction only operate upon arrays of characters. However, since most string data types contain an array of characters as a component, the 80x86's string instructions are handy for manipulating that portion of the string.

Probably the biggest difference between a character string and an array of characters is the length attribute. An array of characters contains a fixed number of characters. Never any more, never any less. A character string, however, has a dynamic run-time length, that is, the number of characters contained in the string at some point in the program. Character strings, unlike arrays of characters, have the ability to change their size during execution (within certain limits, of course).

To complicate things even more, there are two generic types of strings: statically allocated strings and dynamically allocated strings. Statically allocated strings are given a fixed, maximum length at program creation time. The length of the string may vary at run-time, but only between zero and this maximum length. Most systems allocate and deallocate dynamically allocated strings in a memory pool when using strings. Such strings may be any length (up to some reasonable maximum value). Accessing such strings is less efficient than accessing statically allocated strings. Furthermore, garbage collection[1] may take additional time. Nevertheless, dynamically allocated strings are much more space efficient than statically allocated strings and, in some instances, accessing dynamically allocated strings is faster as well.

A string with a dynamic length needs some way of keeping track of this length. While there are several possible ways to represent string lengths, the two most popular are length-prefixed strings and zero-terminated strings. A length-prefixed string consists of a single byte, word, or double word that contains the length of that string. Immediately following this length value, are the characters that make up the string. Assuming the use of byte prefix lengths, you could define the string "HELLO" as follows:

```
byte  5, "HELLO";
```

Length-prefixed strings are often called Pascal strings since this is the type of string variable supported by most versions of Pascal[2].

Another popular way to specify string lengths is to use zero-terminated strings. A zero-terminated string consists of a string of characters terminated with a zero byte. These types of strings are often called C-strings since they are the type used by the C/C++ programming language. If you are manually creating string val-

1. Reclaiming unused storage that occurs when the program is finished using some strings.
2. At least those versions of Pascal which support strings.

ues, zero terminated strings are a little easier to deal with because you don't have to count the characters in the string. Here's an example of a zero terminated string:

```
byte "HELLO", 0;
```

Pascal strings are much better than C/C++ strings for several reasons. First, computing the length of a Pascal string is trivial. You need only fetch the first byte (or word) of the string and you've got the length of the string. Computing the length of a C/C++ string is considerably less efficient. You must scan the entire string (e.g., using the SCASB instruction) for a zero byte. If the C/C++ string is long, this can take a long time. Furthermore, C/C++ strings cannot contain the NULL character. On the other hand, C/C++ strings can be any length, yet require only a single extra byte of overhead. Pascal strings, however, can be no longer than 255 characters when using only a single length byte. For strings longer than 255 bytes, you'll need two or more bytes to hold the length for a Pascal string. Since most strings are less than 256 characters in length, this isn't much of a disadvantage.

Common string functions like concatenation, length, substring, index, and others are much easier to write (and much more efficient) when using length-prefixed strings. So from a performance point of view, length-prefixed strings seem to be the better way to go. However, Windows requires the use of zero-terminated strings; so if you're going to call win32 APIs, you've either got to use zero-terminated strings or convert them before each call.

HLA takes a different approach. HLA's strings are both length-prefixed and zero terminated. Therefore, HLA strings require a few extra bytes but enjoy the advantages of both schemes. HLA's string functions like concatenation are very efficient without losing Windows compatibility.

HLA's strings are actually an extension of length prefixed strings because HLA's strings actually contain *two* lengths: a maximum length and a dynamic length. The dynamic length field is similar to the length field of Pascal strings insofar as it holds the current number of characters in the strring. HLA's length field, however, is four bytes so HLA strings may contain over four billion characters. The static length field holds the maximum number of characters the string may contain. By adding this extra field HLA can check the validity of operations like string concatenation and string assignment to verify that the destination string is large enough to hold the result. This is an extra integrity check that is often missing in string libraries found in typical high level languages.

In addition to providing two lengths, HLA also zero terminates its strings. This lets you pass HLA strings as parameters to Win32 and other functions that work with zero-terminated strings. Also, in those few instances where zero-terminated strings are more convenient, HLA's string format still shines. Of course, the drawback to zero-terminated strings is that you cannot put the NUL character (ASCII code zero) into such a string, fortunately the need to do so is not very great.

HLA's strings actually have another few attributes that improve their efficiency. First of all, HLA almost always aligns string data on double word boundaries. HLA also allocates data for a string in four-byte chunks. By aligning strings on double word boundaries and allocating storage that is an even multiple of four bytes long, HLA allows you to use double word string instructions when processing strings. Since the double word instructions are often four times faster than the byte versions, this is an important benefit. As a result of this storage and alignment, HLA's string library routines are very efficient.

Of course, HLA strings are not without their disadvantages. To represent a string containing n characters requires between $n+9$ and $n+12$ bytes in memory. HLA's strings require at least n+9 bytes because of the two double word length values and the zero terminating byte. Furthermore, since the entire object must be an even multiple of four bytes long, HLA strings may need up to three bytes of padding to ensure this.

HLA string variables are always pointers. HLA even treats string constants as literal pointer constants. The pointer points at the first byte of the character string. Successive memory locations contain successive characters in the string up to the zero terminating byte. This format is compatible with zero-terminated strings like those that C/C++ uses. The dynamic (current) length field is situated four bytes before the first character in the string (that is, at the pointer address minus four). The maximum (static) length field appears eight bytes before the first character of the string. Figure 1.1 shows the HLA string format.

© 2001, By Randall Hyde

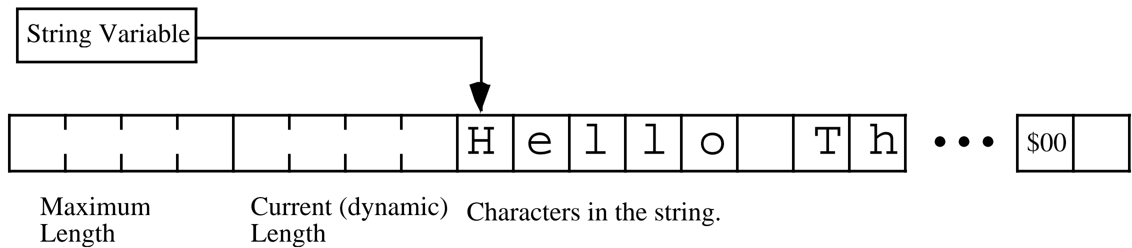

Figure 1.1 HLA String Format

For more information on strings and HLA strings, see the chapter on character strings in AoA.

1.3 HLA Standard Library String Functions

The HLA Standard Library contains a large number of efficient string functions that perform all the common string operations, and then some. This section discusses the HLA string functions and suggests some uses for many of these functions and other objects.

1.3.1 The stralloc and strfree Routines

```
procedure stralloc( strsize: uns32 ); returns( "eax" );
procedure strfree( strToFree: string );
```

This text has already discussed the *stralloc* and *strfree* routines in the chapter on character strings, but a review is probably useful here. These routines dynamically allocate and deallocate storage for a string object in memory. They are the principle mechanism HLA provides for allocating storage for string variables. Therefore, you need to be comfortable using these procedures.

The first thing to note about these routines is that they are not actually a part of the HLA String Library. They are actually members of the memory allocation package in the HLA Standard Library. The reason for mentioning this fact is just to point out that the names of these routines are *stralloc* and *strfree*. Most of the routines in the HLA Standard Library belong to the *str* namespace and, therefore, have names like *str.cpy* and *str.length*. Note that most HLA string function names have a period between the *str* and the base function name; this is not true for *stralloc* and *strfree* since they are not a part of the HLA string package[3].

The *stralloc* parameter specifies the maximum number of characters for the string it allocates. The *stralloc* routine allocates at least enough storage for this many characters plus the 9-12 bytes of overhead required for a string object. It initializes the *MaxStrLen* field to at least *strsize* (it could be as large as *strsize+3* depending on *strsize* and the need for padding bytes in the string object). This function also initializes the *length* field to zero and stores a zero byte in the first character position of the string data (that is, it zero terminates the empty string it creates). Since the other HLA string functions require double word aligned strings, *stralloc* returns a pointer that points at a double word boundary.

Upon return from *stralloc*, the EAX register contains the address of the string object. Generally you would store this 32-bit pointer into a string variable or pass it on to some other function that needs the address of a string object. Like any other string pointer, the value *stralloc* returns points at the first character position in the storage it allocates.

3. Of course, it would be trivial to add a pair of macros, *alloc* and *free*, to the HLA strings package that map *str.alloc* and *str.free* to the *stralloc* and *strfree* names. Feel free to do this if you want to make the use of these two functions a little more consistent with the other strung functions in the HLA Standard Library.

Internally, the *stralloc* routine calls *malloc* to allocate the storage for the string data on the heap. However, the pointer that *stralloc* returns is not the same value that *malloc* returns. This is because string objects require an eight-byte prefix that holds the *MaxStrLen* and *length* fields. Therefore, *stralloc* actually returns a pointer that is eight bytes beyond the value that the internal call to *malloc* returns. Therefore, you cannot call the *free* procedure to return this string storage to the heap because *free* requires a pointer to the beginning of the storage that *malloc* allocates[4]. Instead, call the *strfree* routine to return string object storage to the system. The *strfree's* parameter is the address of a string object that you allocated with *stralloc*.

Note that you must not use *strfree* to attempt to free storage for objects that you do not allocate (directly or indirectly) with *stralloc*. In particular, do not attempt to free statically initialized strings or strings you create with *str.strvar*.

Many of the HLA Standard Library string routines begin with a name of the form "str.a_*****". This "a_" prefix on the function name indicates that the string function automatically allocates storage for a new string by calling *stralloc*. These functions typically return a pointer to the new string in the EAX register, just like *stralloc*. When you are done with the string these functions create, you can free the storage for the string by calling *strfree*.

```
// stralloc and strfree demonstration program.

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

static
    str1    :string;
    str2    :string;

begin str_alloc_free_demo;

    // Allocate a string with a maximum length of 16 characters.

    stralloc( 16 );
    mov( eax, str1 );

    //  Initialize this string with the str.cpy routine:

    str.cpy( "Hello ", str1 );

    // Allocate storage for a second string with 16 characters
    // and initialize the string data:

    stralloc( 16 );
    mov( eax, str2 );
    str.cpy( "world", str2 );

    // Concatenate the two strings and print the pertinent data:

    str.cat( str1, str2 );

    mov( str1, ebx );
    mov( [ebx-4], ecx );    // Get the current string length.
    mov( [ebx-8], edx );    // Get the maximum string length.

    stdout.put
```

4. Technically, you could subtract eight from the value that stralloc returns and then call free with this value. Doing so is not a good idea, however, as the definition of a string object may change in the future and such a change would break code that assumes an eight-byte prefix.

```
    (
        "str1='",
        str1,
        "', length=",
        (type uns32 ecx ),
        ", maximum length=",
        (type uns32 edx),
        nl
    );

    mov( str2, ebx );
    mov( [ebx-4], ecx );    // Get the current string length.
    mov( [ebx-8], edx );    // Get the maximum string length.

    stdout.put
    (
        "str2='",
        str2,
        "', length=",
        (type uns32 ecx ),
        ", maximum length=",
        (type uns32 edx),
        nl
    );

    // Okay, we're done with the strings, free the storage
    // associated with them:

    strfree( str1 );
    strfree( str2 );

end str_alloc_free_demo;
```

Program 1.1 Example of stralloc and strfree Calls

1.3.2 The str.strRec Data Structure

The *str.strRec* data structure lets you directly access the maximum and current length prefix values of an HLA string. This allows you to use symbolic (and meaningful) names to access these fields rather than using numeric offsets like -4 and -8. By using str.strRec you don't have to remember which offset is associated with the two different length values.

The *str.strRec* type definition is a RECORD with the following fields:

```
MaxStrLen
length
strData
```

The *MaxStrLen* field (obviously) specifies the offset (-8) of the maximum string length double word in a string. The *length* field specifies the offset (-4) to the current dynamic length field. The *strData* field specifies the offseet (0) of the first character in the string; generally, you do not use this last field because accessing the character data in a string is trivial (your string variable points directly at the first character in the string).

Generally, you use the str.strRec type to coerce a string pointer appearing in a 32-bit register. For example, if EAX contains the address of an HLA string variable, then "mov((type str.strRec [eax]).length, ecx);" extracts the current string length. In theory, you could use this type to declare string headers, but no one

really uses this data type for that purpose; instead, this type exists mainly as a mechanism for type coercion. The following sample program is a modification of the previous program that uses *str.strRec* rather than literal numeric offsets.

```
// str.strRec demonstration program.

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

static
    str1    :string;
    str2    :string;

begin strRec_demo;

    // Allocate a string with a maximum length of 16 characters.

    stralloc( 16 );
    mov( eax, str2 );

    //  Initialize this string with the str.cpy routine:

    str.cpy( "Hello ", str2 );

    // Allocate storage for a second string with 16 characters
    // and initialize the string data:

    stralloc( 16 );
    mov( eax, str1 );
    str.cpy( "world", str1 );

    // Concatenate the two strings and print the pertinent data:

    str.cat( str1, str2 );

    mov( str1, ebx );
    mov( (type str.strRec [ebx]).length, ecx );    // Get the current str len
    mov( (type str.strRec [ebx]).MaxStrLen, edx ); // Get the max str len

    stdout.put
    (
        "str1='",
        str1,
        "', length=",
        (type uns32 ecx ),
        ", maximum length=",
        (type uns32 edx),
        nl
    );

    mov( str2, ebx );
    mov( (type str.strRec [ebx]).length, ecx );    // Get the current str len
    mov( (type str.strRec [ebx]).MaxStrLen, edx ); // Get the max str len

    stdout.put
    (
        "str2='",
        str2,
        "', length=",
```

 © 2001, By Randall Hyde

```
        (type uns32 ecx ),
        ", maximum length=",
        (type uns32 edx),
        nl
    );

    // Free the storage associated with these strings:

    strfree( str1 );
    strfree( str2 );

end strRec_demo;
```

Program 1.2 Programming Example that uses the str.strRec Data Type

1.3.3 The str.strvar Macro

The *str.strvar* macro statically allocates storage for a string in the STATIC variable declaration section (you cannot use *str.strvar* in any of the other variable declaration sections). This provides a convenient mechanism for declaring static strings when you know the maximum size at compile-time.

Example:

```
static
    StaticString: str.strvar( 32 );
```

This macro invocation does two things: (1) it reserves sufficient storage for a string that can hold at least 32 characters (plus an additional nine bytes for the string overhead); (2) it allocates storage for a string pointer variable and initializes that variable with the address of the string storage. When you reference the object named *StaticString* you are actually accessing this pointer variable.

Note that *str.strvar* uses parentheses rather than square brackets to specify the string size. Syntactically, square brackets would be nice since this gives the illusion of declaring an array of characters. However, *str.strvar* is a macro and the character count is a parameter; macro parameters always appear within parentheses, so you must use parentheses in this declaration.

```
// str.strvar demonstration program.

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

static
    demoStr :str.strvar( 16 );

begin strvar_demo;

    // Initialize our string via str.a_cpy (note that a_cpy automatically
    // allocates storage for the string on the heap):

    str.cpy( "Hello World", demoStr );

    mov( demoStr, ebx );
    mov( (type str.strRec [ebx]).length, ecx );    // Get the current str len
    mov( (type str.strRec [ebx]).MaxStrLen, edx ); // Get the current str len

    stdout.put
```

```
    (
        "demoStr='",
        demoStr,
        "', length=",
        (type uns32 ecx ),
        ", maximum length=",
        (type uns32 edx),
        nl
    );

end strvar_demo;
```

Program 1.3 Program that Demonstrates the use of the str.strvar Declaration

1.3.4 The str.length Function and the str.mLength Macro

```
procedure str.length( s:string ); returns( "eax" );
macro str.mLength( s ); // s must be a 32-bit register or a string variable
```

The *str.length* function and str.mLength macro compute the length of an HLA string and copy this length into the EAX register. The macro version (*str.mLength*) is more efficient since it compiles into a single MOV instruction (accessing the *str.strRec*.length field directly). For this reason you should generally use the macro (*str.mLength*) to compute the length rather than the *str.length* function. You should only use the *str.length* function when you need procedure call semantics (e.g., when you need to pass the address of the length function to some other procedure).

You may question why HLA even provides a length function. After all, extracting the string's length using the *str.strRec* type definition is easy enough to do. The principle reason HLA provides a length function is because "str.length(s)" is much easier to read and understand than "mov((type str.strRec [eax]).length, eax);" Of course, the *str.mLength* function compiles directly into this instruction, so there is no efficiency reason for using the direct access mechanism. The only time you should really use the *str.strRec* RECORD type is when you need to move the string length into a register other than EAX.

The *str.length* and *str.mLength* parameters must be a string variable or a 32-bit register (which, presumably, contains the address of a string in memory). Remember, string variables are really nothing more than pointers, so when you pass a string variable as a parameter to an HLA string function, HLA passes the value of that pointer which happens to be the address of the first character in the string.

There is a big difference between the two calls "str.length(eax);" and "str.length((type string [eax]));" The first call assumes that EAX contains the value of a string pointer (that is, EAX points directly at the first character of the actual string); in this first example, HLA simply passes the value in the EAX register to the *str.length* function. In the second example, "str.length((type string [eax]));" , HLA assumes that EAX contains the address of a string variable (which is a pointer) and passes the 32-bit address at the location contained within EAX. In this example, EAX is a pointer to a string variable rather than the string itself.

Computing the length of a string is one of the most common string operations. In fact, length computation is probably the most oft-used string functions in a string library since most of the other string functions need to compute the string length in order to do their work. This is why HLA's length-prefixed string data structure is so important- computing the string length is a common operation and length-prefixed strings make this computation trivial.

```
// str.length demonstration program.

program strlength_demo;
```

```
#include( "stdlib.hhf" )

static
    demoStr :string;

begin strlength_demo;

    // Initialize our string via str.a_cpy (note that a_cpy automatically
    // allocates storage for the string on the heap):

    str.a_cpy( "Hello World" );
    mov( eax, demoStr );

    mov( eax, ebx );
    str.mLength( ebx );       // Can use a register or str var with str.mLength.
    mov( eax, ecx );
    str.length( demoStr );    // Can use a register or str var with str.length.

    stdout.put
    (
        "demoStr='",
        demoStr,
        "', length via str.mLength=",
        (type uns32 ecx ),
        ", length via str.length=",
        (type uns32 eax),
        nl
    );

    // Free the storage allocated by the str.a_cpy procedure:

    strfree( demoStr );


end strlength_demo;
```

Program 1.4 Example of str.length and str.mLength Function Calls

1.3.5 The str.init Function

```
procedure str.init( var b:byte; numBytes:dword ); returns( "eax" );
```

There are four ways you can allocate storage for an HLA compatible string: you can use the *str.strvar* macro (see “The str.strvar Macro” on page 931) to statically allocate storage for a string, you can initialize a string variable in a STATIC or READONLY section, you can dynamically allocate storage using a function like *stralloc*, or you can manually reserve the storage yourself. To manually reserve storage you must set aside enough storage for the string, the maximum length, the current length, the zero terminating byte, and any necessary padding bytes. You must also ensure that the string begins on a double word boundary and that the entire structure’s byte count is an even multiple of four[5]. After you reserve sufficient storage, you must also initialize the *MaxStrLen* and *length* fields and supply a zero terminating byte for the string. This turns out to be quite a bit of work. Fortunately, the *str.init* function takes care of most of this work for you.

5. HLA string functions require double word alignment and also require that the data area be an even number of double words.

This function initializes a block of memory for use as a string object. It takes the address of a character array variable *b* and aligns this address to a double word boundary. Then it initializes the *MaxStrLen*, *length*, and zero terminating byte fields at the resulting address. Finally, it returns a pointer to the newly created string object in EAX. The *numBytes* field specifies the size of the entire buffer area, not the desired maximum length of the string. The *numBytes* field must be 16 or greater, else this routine will raise an *ex.ValueOutOfRange* exception. Note that string initialization may consume as many as 15 bytes (up to three bytes to align the address on a double word boundary, four bytes for the *MaxStrLen* field, four bytes for the *length* field, and the string data area must be a multiple of four bytes long (including the zero terminating byte). This is why the *numBytes* field must be 16 or greater. Note that this function initializes the resulting string to the empty string. The *MaxStrLen* field will contain the maxium number of characters that you can store into the resulting string after subtracting the zero terminating byte, the sizes of the length fields, and any alignment bytes that were necessary.

In general, if you want the maximum string length to be at least *m* characters, you should reserve *m+16* bytes and pass the address of this buffer to *str.init*. Note that the actual maximum length HLA writes to the *MaxStrLen* field is the maximum number of characters one could legally put into the string (after subtracting the overhead and padding bytes). If you need to set a specific *MaxStrLength* value of exactly *m*, then allocate *m+16* bytes of storage, call *str.init* (passing the address of the buffer and *m+16*), and then store *m* into the *MaxStrLen* field upon return from *str.init*.

```
// str.init demonstration program.

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

static
    theStr  :string;
    unalign :byte;            // Do this so strData is not dword aligned.
    strData :byte[ 48 ];      // Storage for a string with 32 characters.

begin strinit_demo;

    // Create a string variable using the "strData" array to hold the
    // string data:

    str.init( strData, 48 );
    mov( eax, theStr );

    // Initialize our string via str.cpy:

    str.cpy( "Hello there World, how are you?", theStr );

    mov( theStr, ebx );
    str.length( ebx );
    mov( (type str.strRec [ebx]).MaxStrLen, edx );
    lea( esi, strData );

    stdout.put
    (
        "theStr='",
        theStr,
        "', length=",
        (type uns32 ecx ),
        ", max length=",
        (type uns32 edx),
        nl,
        "Address of strData: ",
        esi,
        nl,
```


© 2001, By Randall Hyde

```
        "Address of start of string data: ",
        (type dword theStr),
        nl
    );

end strinit_demo;
```

Program 1.5 Programming Example that uses the str.init Function

1.3.6 The str.cpy and str.a_cpy Functions

```
procedure str.cpy( src:string; dest:string );
procedure str.a_cpy( src:string ); returns( "eax" );
```

The *str.cpy* routine copies the character data from one string to another and adjusts the destination string's *length* field accordingly. The destination string's maximum string length must be at least as large as the current size of the source string or *str.cpy* will raise a string overflow exception. Before calling this routine, you must ensure that both strings have storage allocated for them or the program will raise an exception. Note that simply declaring a destination string variable does not allocate storage for the string object. You must call *stralloc* or somehow otherwise allocate data storage for the string. Failing to allocate storage for the destination string is probably the most common mistake beginning programmers make when calling the *str.cpy* routine.

Note that there is a fundamental difference between the following two code sequences:

```
mov( srcStr, eax );
mov( eax, destStr );
```

and

```
str.cpy( srcStr, destStr );
```

The two MOV instructions above copy a string by reference whereas the call to *str.cpy* copies the string by value. Usually, copying a string by reference is much faster than copying the string by value, since you need only copy four bytes (the string pointer) when copying by reference. Copy by value, on the other hand, requires copying the length value (four bytes), each character in the string (*length* bytes), plus a zero terminating byte. This is slower than simply copying a pointer and can be much slower if the string is long. However, keep in mind that if you copy a string by reference, then the two string objects are aliases of one another. Any change to you make to one of the strings is reflected in the other. When you copy a string by value (using *str.cpy*), each string variable has its own data, so changes to one string will not affect the other.

Although str.cpy does not automatically allocate storage for the destination string, the need to do this arises quite often. The *str.a_cpy* handles this common requirement. As you can see above, the *str.a_cpy* routine does not have a destination operand. Instead, *str.a_cpy* calls *stralloc* to allocate sufficient storage for a new string and copies the source string to this new string. After copying the data, *str.a_cpy* returns a pointer to the new string in the EAX register. When you are done with this string data you should call *strfree* to return the storage back to the system.

```
// str.cpy demonstration program.

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

static
```

```
    strConst    :string := "This is a string";
    srcStr      :str.strvar( 32 );
    destStr     :string;
    smallStr    :str.strvar( 12 );

begin strcpy_demo;

    // Use str.cpy to initialize srcStr by copying the
    // static string constant <<strConst>> to srcStr.

    str.cpy( strConst, srcStr );
    str.length( srcStr );
    stdout.put
    (
        "srcStr='",
        srcStr,
        "', length=",
        (type uns32 eax ),
        nl
    );

    // Okay, now use str.a_cpy to make a copy of srcStr
    // whose storage is dynamically allocated on the heap:

    str.a_cpy( srcStr );
    mov( eax, destStr );
    str.mLength( eax );
    stdout.put
    (
        "destStr='",
        destStr,
        "', length=",
        (type uns32 eax ),
        nl
    );

    // Now let's demonstrate what can go wrong if a string
    // overflow occurs:

    try

        str.cpy( srcStr, smallStr );

      anyexception

        stdout.put( "An exception occured while copying srcStr to smallStr" nl );

    endtry;

    // Don't forget to free the storage associated with destStr:

    strfree( destStr );

end strcpy_demo;
```

Program 1.6 Program that uses the str.cpy and str.a_cpy Procedures

© 2001, By Randall Hyde

1.3.7 The str.cat and str.a_cat Functions

```
procedure str.cat( src: string; dest: string );
procedure str.a_cat( leftSrc: string; rightSrc: string ); returns( "eax" );
```

These two functions concatenate two strings. The *str.cat* procedure directly concatenates one string to the end of the destination string (that the second parameter specifies). The *str.a_cat* procedure creates a new string on the heap (by calling *stralloc*) and copies the string the first parameter specifies to this new string. Immediately thereafter, it concatenates the string object the second parameter specifies to the end of this new string. Finally, *str.a_cat* returns the address of the new string in the EAX register. Note that *str.a_cat,* unlike *str.cat*, does not affect the value of either string appearing in the parameter list. When you finish using the string that *str.a_cat* allocates, you can return the storage to the system by passing the address to *strfree*.

String concatenation is easily one of the most common string operations (the others being string copy and string comparison). Concatenation is a fundamental operation that you use to build larger strings up from smaller strings. A few common examples of string concatenation include applying suffixes (like ".HLA") to filenames and merging a person's first and last names together to form a single string.

Program 1.7 Examples of the str.cat and str.a_cat Procedures

1.3.8 The String Comparison Routines

```
procedure str.eq( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.ne( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.lt( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.le( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.gt( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.ge( lftOperand: string; rtOperand: string ); returns( "al" );

procedure str.ieq( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.ine( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.ilt( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.ile( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.igt( lftOperand: string; rtOperand: string ); returns( "al" );
procedure str.ige( lftOperand: string; rtOperand: string ); returns( "al" );
```

These procedures compare two strings. They are equivalent to the boolean expression:

```
lftOperand  op  rtOperand
```

where op represents one of the relational operators "=", "<>" ("!=" to C programmers), "<", "<=", ">", or ">=". These functions return true (1) or false (0) in the EAX register depending upon the result of the comparison[6]. For example, "str.lt(s, r);" returns true in EAX if $s < r$, it returns false otherwise. This feature lets you use these procedures as boolean expression. The following example shows how you could use *str.lt* in an IF statement:

```
if( str.lt( s, r )) then

    -- do something if s < r --
```

6. The RETURNS value is AL to allow a straight assignment to boolean variables, but these functions actually return zero or one in all of EAX.

```
endif;
```

As you've probably noticed, there are two different sets of string comparison functions. Those that have names of the form "str.i**" do case insensitive string comparisons. That is, these functions compare the strings ignoring differences in alphabetic case. For example, these functions treat "Hello" and "hello" as through they were the same string. Note that case insensitive comparisons are relatively inefficient compared with case sensitive comparisons, so you should only use these forms if you absolutely need a case insensitive comparison.

These functions do not modify their parameters.

```
// String comparisons demonstration program.

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

static
    str1    :string := "abcdefg";
    str2    :string := "hijklmn";
    str3    :string := "AbCdEfG";



    procedure cmpStrs( s1:string; s2:string ); nodisplay;
    var
        eq      :boolean;
        ne      :boolean;
        lt      :boolean;
        le      :boolean;
        ge      :boolean;
        gt      :boolean;

    begin cmpStrs;

        stdout.put( nl "String #1 = '", s1, "'" nl );
        stdout.put( "String #2 = '", s2, "'" nl nl );

        str.eq( s1, s2 ); mov( al, eq );
        str.ne( s1, s2 ); mov( al, ne );
        str.lt( s1, s2 ); mov( al, lt );
        str.le( s1, s2 ); mov( al, le );
        str.ge( s1, s2 ); mov( al, ge );
        str.gt( s1, s2 ); mov( al, gt );

        stdout.put
        (
            "eq = ", eq, nl,
            "ne = ", ne, nl,
            "lt = ", lt, nl,
            "le = ", le, nl,
            "ge = ", ge, nl,
            "gt = ", gt, nl
        );

        str.ieq( s1, s2 ); mov( al, eq );
        str.ine( s1, s2 ); mov( al, ne );
        str.ilt( s1, s2 ); mov( al, lt );
        str.ile( s1, s2 ); mov( al, le );
        str.ige( s1, s2 ); mov( al, ge );
```

 © 2001, By Randall Hyde

```
        str.igt( s1, s2 ); mov( al, gt );

        stdout.put
        (
            "ieq = ", eq, nl,
            "ine = ", ne, nl,
            "ilt = ", lt, nl,
            "ile = ", le, nl,
            "ige = ", ge, nl,
            "igt = ", gt, nl
        );

    end cmpStrs;



begin strcmp_demo;

    cmpStrs( str1, str2 );
    stdout.put( nl "------------------------------" nl );

    cmpStrs( str1, str3 );
    stdout.put( nl "------------------------------" nl );

    cmpStrs( str2, str3 );
    stdout.put( nl "------------------------------" nl );


    cmpStrs( str2, str1 );
    stdout.put( nl "------------------------------" nl );

    cmpStrs( str3, str1 );
    stdout.put( nl "------------------------------" nl );

    cmpStrs( str3, str2 );
    stdout.put( nl "------------------------------" nl );


end strcmp_demo;
```

Program 1.8 Examples of the HLA String Comparison Functions

1.3.9 The str.prefix and str.prefix2 Functions

```
procedure str.prefix( src:string; prefixStr:string ); returns( "al" );
procedure str.prefix2( src:string; offs:uns32; prefixStr:dword );
    returns("al");
```

The *str.prefix* and *str.prefix2* functions are similar to str.eq insofar as they compare two strings and return true or false based on the comparison. Unlike *str.eq*, however, these two functions return true if one string begins with the other (that is, if the second string is a *prefix* of the first string).

The *str.prefix* compares *prefixStr* against *src*. If *prefixStr* is equal to *src*, or the *src* string begins with the characters in *prefixStr* and contains additional characters, then the *str.prefix* function returns true in EAX[7]. If the *src* string does not begin with the characters in *prefixStr*, then *str.prefix* returns false.

The *str.prefix2* function lets you specify a starting index within the *src* string where this function begins searching for the *src* string.

Program 1.9 Examples of the HLA str.prefix Function

1.3.10 The str.substr and str.a_substr Functions

```
procedure str.substr( src:string; dest:string; index:dword; len:dword );
procedure str.a_substr( src:string; index:dword; len:dword ); returns("eax");
```

The *str.cat* and *str.a_cat* procedures let you assemble different strings to produce larger strings; the *str.substr* and *str.a_substr* function do the converse – they let you disassemble strings by extraction small substrings from a larger string. The substring functions are another set of very common string operations. Programs that do a bit of string manipulation will probably use the substring functions in addition to the copy and concatenation functions.

Like all the HLA string functions that produce a string result, the substring functions come in two flavors: one that stores the resulting substring into a string object you've preallocated (*str.substr*) and a second form that automatically allocates storage on the heap for the result (*str.a_substr*). As usual, this second form returns a pointer to the new string in EAX and you should recover this storage by calling *strfree* when you're done using the string data.

The substring functions extract a portion of an existing string by specifying the starting character position in the string (the *index* parameter) and the length of the resulting string (the *length* parameter). The *index* parameter specifies the zero-based index of the first character to copy into the substring. That is, if *index* contains zero then the substring functions begin copying the string data starting with the first character of the string; likewise, if *index* contains five, then the substring functions begin copying the string data with the sixth character in the source string. The value of the *index* parameter must be between zero and the current length of the source string minus one. The substring functions will raise an exception if *index* is outside this range.

The *length* parameter specifies the length of the destination string; that is, it specifies how many characters to copy from the source string to the destination string. If the sum of *index+length* exceeds the current length of the source string, then the substring functions only copy the data from location *index* to the end of the source string; in particular, these functions do not raise an exception if *index's* value is okay but the sum of *index* and *length* exceeds the length of the source string. You can take advantage of this fact to copy all the characters from some point in a string to the end of that string by specifying a really large value for the *length* parameter; the convention is to use -1, which is $FFFF_FFFF (the largest possible unsigned integer), for this purpose.

The *str.substr* function copies the substring data to the string object specified by the *dest* parameter. This string must have sufficient storage to hold a string whose maximum length is *length* characters (or from position *index* to the end of the source string if the sum *index+sum* exceeds the source string length). The *str.substr* function updates the destination string's length field (but does not change the *MaxStrLen* field) and zero terminates the resulting string.

The *str.a_substr* doesn't have a destination string parameter. Instead, this function allocates storage for the destination string on the heap, copies the substring to the new string object, and then returns a pointer to this string object in the EAX register. When allocating storage for the new string, the *str.a_substr* function

7. The RETURNS string is "AL", but these functions actually return true or false in all of EAX.

 © 2001, By Randall Hyde

allocates just enough storage to hold the string and the necessary overhead bytes (between nine and twelve bytes). This function will not raise a string overflow error since it always allocates sufficient storage to hold the destination string (note however, that a memory allocation failure can raise an exception).

Note that it is perfectly possible, and reasonable, to specify zero as the *length* parameter for these substring functions. Doing so will extract a zero length (empty) string from the source string.

A common use of the substring functions is to extract words, numbers, or other special sequences of characters from a string. To do this you must first locate the start of the special sequence in the string and then determine the length of that special sequence; then you can use one of the substring functions to easily extract the sequence of characters you want from the string. This is such a common operation that HLA provides a set of special routines that automatically extract such sequences for you. Details on these functions appear later in this chapter (see "The str.tokenize and str.tokenize2 Functions" on page 962).

```
// str.substr demonstration program.

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

static
    aLongStr    :string := "Extract a substring from this one";
    subStr1     :string;
    subStr2     :string;

begin substr_demo;

    // Allocate storage to hold the first substring:

    mov( stralloc( 64 ), subStr1 );

    // Extract the word "Extract" from the string above:

    str.substr( aLongStr, 0, 7, subStr1 );
    stdout.put( "Extracting 'Extract' = <", subStr1, ">" nl );

    // Extract all the different string lengths from the string above:

    stdout.put( nl "str.substr demonstration:" nl nl );

    mov( str.length( aLongStr ), edx );
    for( mov( 0, ecx ); ecx < edx; inc( ecx ) ) do

        str.substr( aLongStr, 0, ecx, subStr1 );
        stdout.put( "'", subStr1, "'" nl );

    endfor;
    stdout.newln();

    // Demonstrate the use of str.a_substr and exceeding the string length:

    str.a_substr( aLongStr, 30, 100 );
    mov( eax, subStr2 );
    stdout.put( "End of the string is '", subStr2, "'" nl );

    strfree( subStr2 );  // Free the storage allocated by str.a_substr

    // Demonstrate what happens if the index exceeds the string's bounds

    try
```

```
        str.substr( aLongStr, 64, 4 );

        // We won't get here

      anyexception

        stdout.put
        (
            "Exception occured when indexing beyond the length of aLongStr"
            nl
        );

    endtry;

end substr_demo;
```

Program 1.10 Using the str.substr and str.a_substr Procedures

1.3.11 The str.insert and str.a_insert Functions

```
procedure str.insert( src:string; dest:string; index:dword );
procedure str.a_insert( src:string; in_to:string; index:dword ); returns("eax");
```

These two functions insert a source string into a destination string. Unlike the concatenation functions, these routines let you insert the source string into the destination string at any character position, not just at the end of the string. Therefore, these functions are a generalization of the string concatenation operation.

The *str.insert* function inserts a copy of the *src* string into the *dest* string starting at character position *index* in the destination. The *index* value must be in the range 0..*str.length(dest)* or the program will raise an exception. The destination string must have sufficient storage to hold its original value plus the new string or the function will raise an exception.

The *str.a_insert* function does not modify its destination string (the *in_to* parameter). Instead, this function allocates storage for a new string on the heap, copies the data from the *in_to* string to this new string object, and then inserts the *src* string into this string object[8]. Like the other "str.a_****" routines, this function returns a pointer to the new string in EAX and you should free this storage by calling *strfree* when you are done using the string data.

When copying the source string to the destination, the string insertion routines insert the source string before the character at position *index* in the destination string. Note that the index value may lie in the range 0..*str.length(dest)* or 0..*str.length(in_to)*. Most string functions only allow values in the range 0..(*str.length(stringValue)*-1). The insert procedures allow the *index* value to be one greater; doing so tells these routines to insert the source string at the end of the destination string. In this case, the string insertion routines degenerate into string concatenation[9].

```
// str.insert demonstration program.

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

8. Technically, this isn’t exactly how this string function operates, but the actual operation is a little more difficult to follow. This description is semantically correct, however.
9. Note that the string concatenation routines are slightly more efficient to use. So you should use them if you simply want to concatenate to strings. This behavior of the string insertion procedures is useful when you are calculating the insertion index.

© 2001, By Randall Hyde

```
static
    insertInMe      :string := "Insert into this string";
    strToInsert     :string := " 'a string to insert'";
    dest1           :string;
    dest2           :string;

begin insert_demo;

    // Allocate storage to hold the first combined string:

    mov( stralloc( 64 ), dest1 );

    // Display the strings we're going to work with:

    stdout.put( "Insert into: '", insertInMe, "'" nl );
    stdout.put( "String to insert:", strToInsert, nl );

    // Insert strToInsert at the fifth character position in insertInMe
    // (note that we can't actually insert into insertInMe because
    // the string it points at is a literal string whose length is fixed,
    // therefore, we will actually insert strtoInsert into the copy of
    // insertInMe that we've made in dest1):

    str.insert( strToInsert, dest1, 6 );

    stdout.put( nl "Combined string: <", dest1, ">" nl );

    // Demonstrate the same thing using str.a_insert:

    str.a_insert( insertInMe, strToInsert, 6 );
    mov( eax, dest2 );
    stdout.put( "Combined via str.a_insert: <", dest2, ">" nl );


    // Demonstrate what happens if the index exceeds the string's bounds

    try

        str.insert( dest1, strToInsert, 64 );

        // We won't get here

      anyexception

        stdout.put
        (
            "Exception occured when indexing beyond the length of dest1"
            nl
        );

    endtry;

end insert_demo;
```

Program 1.11 Using the str.insert and str.a_insert Procedures

1.3.12 The str.delete and str.a_delete Functions

```
procedure str.delete( dest:string; index:dword; length:dword );
procedure str.a_delete( src:string; index:dword; length:dword ); returns("eax");
```

These functions remove characters from the string parameter. They remove the number of characters the *length* parameter specifies starting at the zero-based position found in the *index* parameter. The *str.delete* procedure removes the characters directly from the string the *dest* parameter specifies. The *str.a_delete* procedure does not modify its string parameter; instead, it makes a copy of the string on the heap and deletes the characters from that copy. The *str.a_delete* procedure returns a pointer to the new string in the EAX register. Like the other "str.a_*****" routines, you should call *strfree* to release this string storage when you are done using it.

The string delete procedures will raise an exception if the *index* parameter is greater than the current length of the string. If index is equal to the length of the string, then these procedures do not delete any characters from the string. If the sum of *index* and *length* is greater than the current length of the string, then these routines will delete all the characters from position *index* to the end of the string. You can use this behavior to delete all the characters from some position to the end of the string by specifying a large value for the length (the convention is to use -1 for this purpose).

```
// str.delete demonstration program.

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

static
    aLongStr    :string := "Delete a substring from this one";
    dest1       :string;
    dest2       :string;

begin delete_demo;

    // Allocation storage for a string so we can demonstrate str.delete:

    mov( stralloc( 64 ), dest1 );
    strcpy( aLongStr, dest 1 );

    // Okay, demonstrate deleting a substring from an existing string
    // (Delete "sub" from "substring" in aLongStr):

    str.delete( dest1, 9, 3 );
    stdout.put( "Original string: '", aLongStr, "'" nl );
    stdout.put( "Resultant string: '", dest1, "'" nl nl );

    // Okay, now demonstrate the str.a_delete procedure.
    // Also demonstrate what happens when the length exceeds the
    // string bounds (but the starting index is within bounds):

    str.a_delete( aLongStr, 18, 100 );
    mov( eax, dest2 );
    stdout.put( "Original string: '", aLongStr, "'" nl );
    stdout.put( "Resultant string: '", dest2, "'" nl nl );


    // Demonstrate what happens if the index exceeds the string's bounds

    try
```

```
        str.delete( dest1, 64, 4 );

        // We won't get here

      anyexception

        stdout.put
        (
            "Exception occured when indexing beyond the length of dest1"
            nl
        );

    endtry;

end delete_demo;
```

Program 1.12 Using the str.delete and str.a_delete Procedures

1.3.13 The str.replace and str.a_replace Functions

```
procedure str.replace( dst:string; from:string; to:string );
procedure str.a_replace( src:string; from:string; to:string ); returns( "eax" );
```

These two functions replace characters in a string via a small lookup table. They scan through the *dst/src* string a character at a time and search through the *from* string for this character. If the routines do not find this character, they copy the current character to the destination string. If these routines find the current character in the *from* string, then they copy the character at the corresponding position in the *to* string to the destination string (in place of the original character).

As usual for the HLA string functions, the difference between *str.replace* and *str.a_replace* is that the *str.replace* function manipulates the *dst* string directly while the *str.a_replace* procedure copies and translates the characters from *src* to a new destination string it allocates on the heap via *stralloc*. Of course, you should free the strings *str.a_replace* allocates by calling *strfree* when you are done using the string data.

Usually, the *from* and *to* strings will be the same length because these routines use the index into the *from* string to select the translation character in the *to* string. However, it is not an error if these two strings have different lengths. If the *to* string is longer than the from string, then the *replace* routines simply ignore the extra characters in the *to* string. If the *to* string is shorter than the *from* string, then the *replace* routines will delete any characters found in the *from* string that don't have a corresponding character in the *to* string.

An example may help clarify the purpose of these routines. In past chapters, you've seen how to use the XLAT instruction to translate lower case to upper case characters. One drawback to using XLAT is that you have to create a 256-byte lookup table. You can accomplish this with somewhat less effort using the *str.replace* procedure. Here's the code that will translate lower case to upper case within a string:

```
str.replace
(
    theString,
    "abcdefghijklmnopqrstuvwxyz",
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
);
```

If *theString* contains "Hello", then the call above looks up "H" in the second parameter and doesn't find it. Therefore, it doesn't change the first character of *theString*. Next, *str.replace* looks up "e" in the second parameter; this time it finds the character so it replaces "e" in *theString* with the character at the corresponding position (5) in the third parameter. The fifth character position contains an "E", so *str.replace* substitutes an "E" for the "e" in the second character position of *theString*. This process repeats for the remaining char-

acters in *theString*; since they are all lower case characters (present in the second parameter) the *str.replace* routine converts them to upper case.

Note that these routines are not particularly efficient. For each character appearing in the first string parameter, these functions have to scan through the second parameter. If the first parameter is *n* characters long and the second string is *m* characters long, this process could require as many as *n*m* comparisons. If the *from* string is rather long, you will get much better performance by using a lookup table and the XLAT instruction (that requires only *n* steps). Certainly you should never use these functions for case conversion (as in this example) because the HLA Standard Library already provides efficient routines for translating the case of characters within a string (see "The str.upper, str.a_upper, str.lower, and str.a_lower Functions" on page 951). Nevertheless, these functions are convenient to use and are not especially inefficient if the *from* string is not very large (say less than 10 characters or so).

```
// str.replace demonstration program.

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

static
    digitsStr    :string := "Count 1 the 2 number 3 of 4 digits 5 in 6 "
                            "this 7 string 8";
    dest1       :string;
    dest2       :string;

begin replace_demo;

    // Allocation storage for a string so we can demonstrate str.replace:

    mov( stralloc( 64 ), dest1 );
    strcpy( digitsStr, dest 1 );

    // Convert all the digits to periods and delete everything else.
    // After this conversion, the length of the string will tell us
    // how many digits were in the string.

    str.replace( dest1, "0123456789", ".........." );
    str.length( dest1 );
    stdout.put
    (
        "Original string: '", digitsStr, "'" nl
        "Result string:   '", dest1, "'" nl
        "Length of result: ", (type uns32 eax), nl
    );

    // As above, but demonstrate str.a_replace and count the number
    // of alphabetic characters.

    str.a_replace
    (
        digitsStr,
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "...................................................."
    );
    mov( eax, dest2 );
    str.length( dest2 );
    stdout.put
    (
        "Original string: '", digitsStr, "'" nl
        "Result string:   '", dest2, "'" nl
```

 © 2001, By Randall Hyde

```
        "Length of result: ", (type uns32 eax), nl
    );
    strfree( dest2 );

end replace_demo;
```

Program 1.13 Example that Uses the str.replace and str.a_replace Routines

1.3.14 The str.setstr and str.a_setstr Functions

```
procedure str.setstr( fill:char; dest:string; count:dword );
procedure str.a_setstr( fill:char; count:dword ); returns( "eax" );
```

The *str.set* and *str.a_set* functions create a new character string whose length the *count* parameter specifies. These routines fill the string with *count* copies of the *fill* character. The *str.set* routine fills the *dest* string with the characters; the *dest* string's *MaxStrLen* value must be greater than or equal to *count* or *str.setstr* will raise a string overflow exception. The *str.a_strset* function allocates sufficient storage for a new string on the heap and initializes this string with the specified number of characters; *str.a_strset* returns a pointer to this new string in the EAX register. As usual, you should call *strfree* to deallocate the string *str.a_setstr* creates when you are done with the string.

These functions are especially useful for creating "padding" strings when formatting data for output. If you have some code that translates some data object's representation to a string for output, you can use str.setstr (or str.a_setstr) along with string concatenation to adjust the output string to some minimum width. The example below demonstrates how you could do this:

```
// str.setstr demonstration program.

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

static
    topBottom   :str.strvar(40);
    leftRight   :string;

begin setstr_demo;

    // We are going to use topBottom and leftRight to draw a
    // 30x20 box on the display.  Begin by initializing these
    // strings using str.setstr.

    str.setstr( topBottom, '*', 30 );
    str.a_setstr( ' ', 28 );
    mov( eax, leftRight );

    // Okay, draw the box using the strings we've created:

    stdout.put( topBottom, nl );
    for( mov( 18, ecx ); ecx > 0; dec( ecx )) do

        stdout.put( '*', leftRight, '*', nl );

    endfor;
    stdout.put( topBottom, nl );
```

```
    // Okay, free up the storage we allocated via str.a_setstr above.

    strfree( leftRight );

end setstr_demo;
```

Program 1.14 Using str.strset and str.a_strset to Format Output Strings

1.3.15 The str.index/str.index2 and str.rindex/str.rindex2 Functions

```
procedure str.index( source:string; searchStr:string ); returns( "eax" );
procedure str.rindex( source:string; searchStr:string ); returns( "eax" );
procedure str.index2( source:string; offs:uns32; searchStr:string );
    returns( "eax" );
procedure str.rindex2( source:string; offs:uns32; searchStr:string );
    returns( "eax" );
```

The *str.index* and *str.rindex* search for an occurrence of one string (*searchStr*) within another string (*source*). They return a zero-based index of the position of the *searchStr* within the source string in the EAX register. The term "position" means the index of the first character in source that matches the first character of *searchStr* once these routines locate *searchStr* within *source*. If these routines cannot find the *searchStr* within the *source* string, they return -1 ($FFFF_FFFF) in the EAX register.

Note that if the length of the *searchStr* is greater than the length of the *source* string these functions will always return -1. If the lengths of the two strings are equal, this function returns zero if the two strings are equal, it returns -1 otherwise.

The *str.index* function returns the index of the first occurrence of *searchStr* within *source*. If multiple occurrences exist, this function ignores all but the first occurrence. The *str.rindex* (*reverse* index) locates the last occurrence of the *searchStr* in *source* (that is, this function searches for the *searchStr* in the backwards direction starting at the end of the string).

These functions use a "brute-force" algorithm that is fine for short source strings but is inefficient for really large *source* and *searchStr* combinations. For most strings (where the source string is less than 100-200 characters) using *str.index* and *str.rindex* is probably okay; however, if you want to search through strings that are thousands of characters long, there are better algorithms available (Boyer-Moore string matching comes to mind). For short strings, the overhead of these fancier algorithms diminishes their effectiveness, so don't be afraid to use *str.index* and *str.rindex* on short strings.

The *str.index2* and *str.rindex2* work much like str.index and str.rindex except they let specify a starting position in the source string where these function begin searching for the second string. If these functions find the search string within the source string, they return the index from the beginning of the source string (not from the *offs* value) to the location of the substring they locate.

You should not use these functions to search for individual characters within the *source* string. The next section describes a more efficient solution for searching for single characters within a string.

```
// str.index/str.rindex demonstration program.

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

static
    source      :string := "the world says ""hello there"" slowly";
```

```
begin strindex_demo;

    stdout.put( "Original string: '", source, "'" nl );

    // Output the character position underneath each character
    // so the user can easily see what's happening:

    stdout.put( "                  " );
    mov( 0, dl );
    for( mov( 0, ecx ); ecx < str.length( source ); inc( ecx )) do

        stdout.put( (type uns8 dl ) );
        inc( dl );
        if( dl > 9 ) then

            mov( 0, dl );

        endif;

    endfor;
    stdout.put( nl "                  " );
    mov( 0, dl );
    mov( 0, dh );
    for( mov( 0, ecx ); ecx < str.length( source ); inc( ecx )) do

        inc( dl );
        if( dl > 9 ) then

            inc( dh );
            stdout.put( (type uns8 dh));
            mov( 0, dl );

        endif;

    endfor;
    stdout.put( nl nl );

    // Use str.index and str.rindex to locate the substring "the" within
    // the source string:

    str.index( source, "the" );
    mov( eax, ecx );
    str.rindex( source, "the" );

    stdout.put
    (
        "First location of ""the"" within """,
        source,
        """ is ",
        (type uns32 ecx),
        nl
        "Last location of ""the"" within """,
        source,
        """ is ",
        (type uns32 eax),
        nl
    );


end strindex_demo;
```

Program 1.15 Using the str.index and str.rindex Functions

1.3.16 The str.chpos/str.chpos2 and str.rchpos/str.rchpos2 Functions

```
procedure str.chpos( source:string; searchFor:char ); returns( "eax" );
procedure str.rchpos( source:string; searchFor:char ); returns( "eax" );
procedure str.chpos2( source:string; offs:uns32; searchFor:char );
    returns( "eax" );
procedure str.rchpos2( source:string; offs:uns32; searchFor:char );
    returns( "eax" );
```

These two functions are very similar to the *str.index* and *str.rindex* functions of the previous section. The difference is that these routines search for a single character (*searchFor*) within the *source* string rather than a sequence of characters. These functions return the zero-based index of the *searchFor* character within the *source* string, assuming that the character is present within the string. These functions return -1 in EAX if the character is not present in the string.

The *str.chpos* function searches for the first occurrence of the *searchFor* character within the *source* string. It ignores any additional matching characters after the first occurrence it locates. The *str.rchpos* function locates that last occurrence of *searchFor* within *source* (that is, *str.rchpos* searches backwards through the *source* string for the *searchFor* character). The *str.rchpos* function ignores any earlier characters once it locates the last occurrence of the character within the string.

The *str.chpos2* and *str.rchpos2* procedures work in an identical manner to *str.chpos* and *str.rchpos* except that they let you specify a starting index in the source string. Note that these procedures return an index from the first character in the string rather than from the starting position in the string.

```
// str.chpos/str.rchpos demonstration program.

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

static
    source      :string := "the world says ""hello there"" slowly";

begin strchpos_demo;

    stdout.put( "Original string: '", source, "'" nl );

    // Output the character position underneath each character
    // so the user can easily see what's happening:

    stdout.put( "                  " );
    mov( 0, dl );
    for( mov( 0, ecx ); ecx < str.length( source ); inc( ecx )) do

        stdout.put( (type uns8 dl ) );
        inc( dl );
        if( dl > 9 ) then

            mov( 0, dl );

        endif;

    endfor;
    stdout.put( nl "                  " );
```

```
    mov( 0, dl );
    mov( 0, dh );
    for( mov( 0, ecx ); ecx < str.length( source ); inc( ecx )) do

        inc( dl );
        if( dl > 9 ) then

            inc( dh );
            stdout.put( (type uns8 dh));
            mov( 0, dl );

        endif;

    endfor;
    stdout.put( nl nl );

    // Use str.chpos and str.rchpos to locate the last space within
    // the source string:

    str.chpos( source, ' ' );
    mov( eax, ecx );
    str.rchpos( source, ' ' );

    stdout.put
    (
        "First space within """,
        source,
        """ is at position ",
        (type uns32 ecx),
        nl
        "Last space within """,
        source,
        """ is at position ",
        (type uns32 eax),
        nl
    );


end strchpos_demo;
```

Program 1.16   Using the str.chpos and str.rchpos Functions

1.3.17 The str.upper, str.a_upper, str.lower, and str.a_lower Functions

```
procedure str.lower( dest:string );
procedure str.upper( dest:string );
procedure str.a_lower( src:string ); returns( "eax" );
procedure str.a_upper( src:string ); returns( "eax" );
```

These functions translate alphabetic characters in their parameter strings to upper case (*str.upper* and *str.a_upper*) or to lower case (*str.lower* and *str.a_lower*). The *str.lower* and *str.upper* functions translate the characters directly in the *dest* string parameter. The *str.a_lower* and *str.a_upper* functions copy the *src* string and translate the data while copying it; they return a pointer to the new string in EAX. As with all *str.a_xxxx* routines, you should free the storage by calling *strfree* when you are done with the strings that *str.a_lower* and *str.a_upper* create.

```
// String comparisons demonstration program.

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

static
    str1    :string := "abcdefg";
    str2    :string := "hijklmn";
    str3    :string := "AbCdEfG";

    l_result1   :str.strvar(16);
    l_result2   :str.strvar(16);
    l_result3   :str.strvar(16);

    u_result1   :str.strvar(16);
    u_result2   :str.strvar(16);
    u_result3   :str.strvar(16);

    la_result1  :string;
    la_result2  :string;
    la_result3  :string;

    ua_result1  :string;
    ua_result2  :string;
    ua_result3  :string;

begin struplow_demo;

    // Use the str.lower and str.upper functions
    // to convert str1...str3 to all lower and
    // all upper case:

    str.lower( str1, l_result1 );
    str.lower( str2, l_result2 );
    str.lower( str3, l_result3 );

    str.upper( str1, u_result1 );
    str.upper( str2, u_result2 );
    str.upper( str3, u_result3 );

    // Use the str.a_lower and str.a_upper functions
    // to convert str1..str3 to all lower and all upper
    // case, while allocating storage for the results.

    mov( str.a_lower( str1 ), la_result1 );
    mov( str.a_lower( str2 ), la_result2 );
    mov( str.a_lower( str3 ), la_result3 );

    mov( str.a_upper( str1 ), ua_result1 );
    mov( str.a_upper( str2 ), ua_result2 );
    mov( str.a_upper( str3 ), ua_result3 );

    // Compare and displays the strings we've processed.
    // Compare each combination of original x (l_result,
    // la_result, u_result, ua_result) and display the
    // result.  Also do a case insensitive comparison
    // between the original string and l_result/u_result
    // so the user can see the difference.
```


© 2001, By Randall Hyde

```
// str1:

stdout.put( "str1:       '", str1, "'" nl );
stdout.put( "l_result1:  '", l_result1, "'" nl );
stdout.put( "la_result1: '", la_result1, "'" nl );
stdout.put( "u_result1:  '", u_result1, "'" nl );
stdout.put( "ua_result1: '", ua_result1, "'" nl );

str.eq( str1, l_result1 );
stdout.put( "str1 == l_result1 is (", (type boolean al), "'" nl );
str.eq( str1, la_result1 );
stdout.put( "str1 == la_result1 is (", (type boolean al), "'" nl );

str.eq( str1, u_result1 );
stdout.put( "str1 == u_result1 is (", (type boolean al), "'" nl );
str.eq( str1, ua_result1 );
stdout.put( "str1 == ua_result1 is (", (type boolean al), "'" nl );

str.ieq( str1, u_result1 );
stdout.put
(
    "case insensitive str1 == u_result1 is (",
    (type boolean al),
    "'"
    nl
);

str.ieq( str1, l_result1 );
stdout.put
(
    "case insensitive str1 == l_result1 is (",
    (type boolean al),
    "'"
    nl
);



// str2:

stdout.put( "str2:       '", str2, "'" nl );
stdout.put( "l_result2:  '", l_result2, "'" nl );
stdout.put( "la_result2: '", la_result2, "'" nl );
stdout.put( "u_result2:  '", u_result2, "'" nl );
stdout.put( "ua_result2: '", ua_result2, "'" nl );

str.eq( str2, l_result2 );
stdout.put( "str2 == l_result2 is (", (type boolean al), "'" nl );
str.eq( str2, la_result2 );
stdout.put( "str2 == la_result2 is (", (type boolean al), "'" nl );

str.eq( str2, u_result2 );
stdout.put( "str2 == u_result2 is (", (type boolean al), "'" nl );
str.eq( str2, ua_result2 );
stdout.put( "str2 == ua_result2 is (", (type boolean al), "'" nl );

str.ieq( str2, u_result2 );
stdout.put
(
    "case insensitive str2 == u_result2 is (",
    (type boolean al),
```

```
        "'"
        nl
    );

    str.ieq( str2, l_result2 );
    stdout.put
    (
        "case insensitive str2 == l_result2 is (",
        (type boolean al),
        "'"
        nl
    );


    // str3:

    stdout.put( "str3:       '", str3, "'" nl );
    stdout.put( "l_result3:  '", l_result3, "'" nl );
    stdout.put( "la_result3: '", la_result3, "'" nl );
    stdout.put( "u_result3:  '", u_result3, "'" nl );
    stdout.put( "ua_result3: '", ua_result3, "'" nl );

    str.eq( str3, l_result3 );
    stdout.put( "str3 == l_result3 is (", (type boolean al), "'" nl );
    str.eq( str3, la_result3 );
    stdout.put( "str3 == la_result3 is (", (type boolean al), "'" nl );

    str.eq( str3, u_result3 );
    stdout.put( "str3 == u_result3 is (", (type boolean al), "'" nl );
    str.eq( str3, ua_result3 );
    stdout.put( "str3 == ua_result3 is (", (type boolean al), "'" nl );

    str.ieq( str3, u_result3 );
    stdout.put
    (
        "case insensitive str3 == u_result3 is (",
        (type boolean al),
        "'"
        nl
    );

    str.ieq( str3, l_result3 );
    stdout.put
    (
        "case insensitive str3 == l_result3 is (",
        (type boolean al),
        "'"
        nl
    );



    // Free the storage associated with the calls to
    // str.a_upper and str.a_lower:

    strfree( la_result1 );
    strfree( la_result2 );
    strfree( la_result3 );

    strfree( ua_result1 );
    strfree( ua_result2 );
```

```
    strfree( ua_result3 );

end struplow_demo;
```

Program 1.17 The str.lower, str.upper, str.a_lower, and str.a_upper Functions

1.3.18 The str.delspace and str.a_delspace Functions

```
procedure str.delspace( dest:string );
procedure str.a_delspace( src:string ); returns( "eax" );
```

These two procedures delete leading spaces from a string. The *str.delspace* routine directly deletes the characters from the *dest* string parameter. It does this by shifting all the characters past the first run of spaces over the top of the leading spaces (and adjusting the string's length as appropriate). The *str.a_delspace* function makes a copy of the *src* string and then deletes the leading spaces from this copy in a similar fashion. the *str.a_delspace* function returns a pointer to the new string in EAX; you must free this storage by calling *strfree* when you are done using it.

These functions are quite useful for removing leading blanks from user input when you need to compare the user input against some string or against some set of strings. By removing leading blanks, the comparison is easier because you don't have to worry about the user accidentally hitting the space bar at the beginning of their string and not realizing the input string won't match some string without the leading spaces.

```
// str.delspace demonstration program.

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

static
    inputStr        :string;
    destStr         :string;

begin delspace_demo;

    // Read the input string from the user:

    stdout.put( "Enter a string: " );
    stdin.a_gets();
    mov( eax, inputStr );

    // Delete any leading spaces from the string:

    str.delspace( inputStr );
    if( str.ieq( inputStr, "Hello" )) then

        stdout.put( "You entered ""Hello""" nl );

    else

        stdout.put( "You did not enter ""Hello""" nl );

    endif;

    // Free the storage allocated via stdin.a_gets():
```

```
    strfree( inputStr );

    // Repeat the above to demonstrate str.a_delspace:

    stdout.put( "Enter a string: " );
    stdin.a_gets();
    mov( eax, inputStr );

    // Delete any leading spaces from the string:

    str.a_delspace( inputStr );
    mov( eax, destStr );
    if( str.ieq( destStr, "There" )) then

        stdout.put( "You entered ""There""" nl );

    else

        stdout.put( "You did not enter ""There""" nl );

    endif;

    // Free the storage allocated via stdin.a_gets() and
    // inputStr:

    strfree( inputStr );
    strfree( destStr );



end delspace_demo;
```

Program 1.18 Demonstration of the str.delspace and str.a_delspace Functions

1.3.19 The str.trim and str.a_trim Functions

```
procedure str.trim( dest:string );
procedure str.a_trim( src:string ); returns( "eax" );
```

The *str.trim* and *str.a_trim* functions are very similar to the *str.delspace* and *str.a_delspace* functions. These functions delete spaces at the beginning and end of the string rather than just at the beginning of the string. As you've come to expect, the *str.trim* function deletes the characters directly from parameter string while the *str.a_trim* routine allocates storage for a new string and trims the spaces from the copy (returning a pointer to this string in EAX; don't forget to deallocate this storage when you are done with it).

```
// str.trim demonstration program.

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

static
    inputStr        :string;
```

```
    destStr         :string;

begin trim_demo;

    // Read the input string from the user:

    stdout.put( "Enter a string: " );
    stdin.a_gets();
    mov( eax, inputStr );

    // Delete any leading and trailing spaces from the string:

    str.trim( inputStr );
    if( str.ieq( inputStr, "Hello" )) then

        stdout.put( "You entered ""Hello""" nl );

    else

        stdout.put( "You did not enter ""Hello""" nl );

    endif;

    // Free the storage allocated via stdin.a_gets():

    strfree( inputStr );

    // Repeat the above to demonstrate str.a_trim:

    stdout.put( "Enter a string: " );
    stdin.a_gets();
    mov( eax, inputStr );

    // Delete any leading spaces from the string:

    str.a_trim( inputStr );
    mov( eax, destStr );
    if( str.ieq( destStr, "There" )) then

        stdout.put( "You entered ""There""" nl );

    else

        stdout.put( "You did not enter ""There""" nl );

    endif;

    // Free the storage allocated via stdin.a_gets() and
    // inputStr:

    strfree( inputStr );
    strfree( destStr );



end trim_demo;
```

Program 1.19 Example That Uses the str.trim and str.a_trim Functions

1.3.20 The str.span, str.span2, str.rspan, and str.rspan2 Functions

```
procedure str.span( src:string; skipCset:cset ); returns("eax");
procedure str.span2( src:string; start:uns32; skipCset:cset ); returns("eax");
procedure str.rspan( src:string; skipCset:cset ); returns("eax");
procedure str.rspan2( src:string; start:uns32; skipCset:cset ); returns("eax");
```

The string spanning functions search for a character in a string (*src*) which is not a member of a given character set (*skipCset*). These functions return (in EAX) the zero-based index of the first character they locate which is not a member of the *skipCset* character set. They all return -1 in EAX if all the characters in *src* are members of the *skipCset* character set. None of these functions affect the value of the string parameter.

The *str.span* function searches forward from the beginning of the *src* string (that is, string with character position zero) until it finds a character that is not in *skipCset*. It returns the index of this character (or -1) in EAX. The *str.span2* function works in nearly an identical fashion, but it provides an additional parameter (*start*) that lets you specify the starting position in the string. One common use of *str.span2* is to continue processing data in a string after you've already located a character in a string that is not a member of *skipCset*. By incrementing *str.span's* return value and passing this as the *start* parameter to *str.span2*, you can continue scanning through the string immediately after the offending character.

The *str.rspan* and *str.rspan2* functions (reverse span) operate in a similar fashion to *str.span* and *str.span2* except they start at the end of the string and search backwards (towards the beginning of the string). You can use *str.rspan2* to continue scanning through a string after a previous *str.rspan* or *str.rspan2* call, except of course, you must decrement the return value from the previous call rather than increment it (as when doing this with *str.span* and *str.span2*).

```
// str.span/str.rspan demonstration program.

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

static
    index       :int32;
    source      :string;
    curWord     :str.strvar(128);

begin strspan_demo;

    // This demo program will demonstrate a simple "lexical analyzer"
    // that extracts the English words from a simple string (English
    // words are any sequence of alphabetic characters delimited by
    // non-alphabetic characters.
    //
    // First, create the string we're going to manipulate:

    str.a_cpy(  "the world says: ""hello there"" slowly", source );
    stdout.put( "Original string: '", source, "'" nl );

    // Okay now start the lexical analyzer:

    while( str.length( source ) > 0 ) do

        // Skip any delimiter characters in the string:
```

 © 2001, By Randall Hyde

```
        str.span( source, -{'a'..'z', 'A'..'Z'} );
        if( eax <> -1 ) then

            // Okay, we've got some prefix containing
            // delimiter characters, use str.delete to
            // remove them:

            str.delete( source, 0, eax );

        endif;

        // Okay, now extract the word at the beginning of the
        // string (if there is a word there):

        str.span( source, {'a'..'z', 'A'..'Z'} );
        if( eax <> -1 ) then

            str.substr( source, curWord, 0, eax );
            stdout.put( "Next word: '", curWord, "'" nl );

            // Remove the word from the source string:

            str.delete( source, 0, eax );

        endif;

    endwhile;

    // Free up the storage associated with with the source string
    // we created (note that str.delete does not free the storage
    // associated with the substrings it deletes from source).

    strfree( source );
    stdout.put( nl nl nl );



    // Repeat the code above using the str.span2 function so that
    // we don't destroy the original source string.  This is also
    // quite a bit faster since we don't have to move data around
    // in the source string when we delete the data.

    // First, create the string we're going to manipulate:

    str.a_cpy(  "the world says: ""hello there"" slowly", source );
    stdout.put( "Original string: '", source, "'" nl );

    // Okay now start the lexical analyzer:

    mov( 0, index );
    while( str.length( source ) > index ) do

        // Skip any delimiter characters in the string:

        str.span2( source, index, -{'a'..'z', 'A'..'Z'} );
        if( eax <> -1 ) then

            // Skip the delimiter characters by adjusting the
            // index:
```

```
            mov( eax, index );

        endif;

        // Okay, now extract the word immediately after the
        // delimiter substring (if there is a word there):

        str.span2( source, index, {'a'..'z', 'A'..'Z'} );
        if( eax <> -1 ) then

            mov( eax, ecx );        // Compute the length of the
            sub( index, ecx );      //  word.

            str.substr( source, curWord, index, ecx );
            stdout.put( "Next word: '", curWord, "'" nl );

            // Skip the word by setting the current index
            // to the character just beyond the word:

            mov( eax, index );

        endif;

    endwhile;

    // Free up the storage associated with with the source string.

    strfree( source );
    stdout.put( nl nl nl );


    // Okay, repeat the two operations above except use str.rspan
    // and str.rspan2 to work backwards through the string.
    // This will display the words in the reverse order they appear
    // in the string.
    //
    // First, create the string we're going to manipulate:

    str.a_cpy(  "the world says: ""hello there"" slowly", source );
    stdout.put( "Original string: '", source, "'" nl );

    // Okay now start the lexical analyzer:

    while( str.length( source ) > 0 ) do

        // Skip any delimiter characters in the string:

        str.rspan( source, -{'a'..'z', 'A'..'Z'} );
        if( eax <> -1 ) then

            // Okay, we've got some prefix containing
            // delimiter characters, use str.delete to
            // remove them.  Note that a length of -1
            // tells delete to delete all remaining
            // characters in the string.

            str.delete( source, eax, -1 );

        endif;

        // Okay, now extract the word at the end of the
```

```
    // string (if there is a word there):

    str.rspan( source, {'a'..'z', 'A'..'Z'} );
    if( eax <> -1 ) then

        // A length of -1 tells the str.substr function
        // to extract all remaining characters in the string.

        str.substr( source, curWord, eax, -1 );
        stdout.put( "Last word: '", curWord, "'" nl );

        // Remove the word from the source string:

        str.delete( source, eax, -1 );

    endif;

endwhile;

// Free up the storage associated with with the source string
// we created (note that str.delete does not free the storage
// associated with the substrings it deletes from source).

strfree( source );
stdout.put( nl nl nl );



// Repeat the code above using the str.rspan2 function so that
// we don't destroy the original source string.  This is also
// quite a bit faster since we don't have to move data around
// in the source string when we delete the data.

// First, create the string we're going to manipulate:

str.a_cpy(  "the world says: ""hello there"" slowly", source );
stdout.put( "Original string: '", source, "'" nl );

// Okay now start the lexical analyzer:

mov( str.length( source ), index );
dec( index );
while( index > 0 ) do

    // Skip any delimiter characters in the string:

    str.rspan2( source, index, -{'a'..'z', 'A'..'Z'} );
    if( eax <> -1 ) then

        // Skip the delimiter characters by adjusting the
        // index:

        mov( eax, index );

    endif;

    // Okay, now extract the word immediately after the
    // delimiter substring (if there is a word there):

    str.rspan2( source, index, {'a'..'z', 'A'..'Z'} );
    if( eax <> -1 ) then
```

```
            mov( index, ecx );      // Compute the length of the
            sub( eax, ecx );        //  word.

            str.substr( source, curWord, index, ecx );
            stdout.put( "Next word: '", curWord, "'" nl );

            // Skip the word by setting the current index
            // to the character just beyond the word:

            mov( eax, index );

        endif;

    endwhile;

    // Free up the storage associated with with the source string.

    strfree( source );
    stdout.put( nl nl nl );



end strspan_demo;
```

Program 1.20 String Scanning Using the str.span, str.span2, str.rspan, and str.rspan2 Functions

1.3.21 The str.brk, str.brk2, str.rbrk, and str.rbrk2 Functions

```
procedure str.brk( src:string; fndCset:cset ); returns("eax");
procedure str.brk2( src:string; start:uns32; fndCset:cset ); returns("eax");
procedure str.rbrk( src:string; fndCset:cset ); returns("eax");
procedure str.rbrk2( src:string; start:uns32; fndCset:cset ); returns("eax");
```

These functions are very similar in operation to the spanning functions insofar as they skip over characters in a source string. The difference is that these function skip characters until they find a character in the *fndCset* parameter (rather than skipping characters in the set). Other than this one behavioral difference, you use these functions in a manner identical to the spanning functions.

Program 1.21 String Scanning Using the str.brk, str.brk2, str.rbrk, and str.rbrk2 Functions

1.3.22 The str.tokenize and str.tokenize2 Functions

```
procedure str.tokenize( src: string; var dest:dword ); returns( "eax" );
procedure str.tokenize2( src:string; var dest:dword; delims:cset );
    returns( "eax" );
```

These two routines lexically scan[10] a string and break it up into "lexemes" (words), returning an array of pointers to each of the lexemes. The only difference between the two routines is that the tokenize routine uses the following default set of delimiter characters:

```
{' ', #9, ',', '<', '>', '|', '\', '/', '-'}
```

This character set roughly corresponds to the delimiters used by the Windows Command Window interpreter. If you do not wish to use this particular set of delimiter characters, you may call *str.tokenize2* and specify the characters you're interested in.

The tokenize routines begin by skipping over all delimiter characters at the beginning of the string. Once they locate a non-delimiter character, they skip forward until they find the end of the string or the next delimiter character. Then they allocate storage for a new string on the heap and copy the delimited text to this new string. A pointer to the new string is stored into the double word array passed as the second parameter to *str.tokenize(2)*. This process is repeated for each lexeme found in the *src* string.

Warning: the *dest* parameter should be an array of strings. This array must be large enough to hold pointers to each lexeme found in the string. In theory, there could be as many as string_length/2 lexemes in the source string.

On return from these functions, the EAX register will contain the number of lexemes found and processed in the *src* string (i.e., EAX will contain the number of valid elements in the *dest* array). When you are done with the strings allocated on the heap, you should free them by calling *strfree*. Note that you need to call *strfree* for each active pointer stored in the *dest* array.

The *str.tokenize* and *str.tokenize2* routines are among the most powerful string functions in the HLA Standard Library. Unfortunately, this power comes at a price. Hopefully, the following example will clearly demonstrate how to use these functions.

Program 1.22 Demonstration of the str.tokenize and str.tokenize2 Functions

1.3.23 String Conversion Functions

This section describes several string functions that are actually a part of the HLA Standard Library conversions module rather than the string module. However, since these functions convert data to and from string form, it makes sense to discuss them along with the other HLA string routines. The following subsections describe many of the string conversion routines and how you might use them in your programs.

1.3.23.1 Hexadecimal Conversion Routines

```
procedure conv.hToStr ( h:byte; buffer:string );
procedure conv.wToStr ( w:word; buffer:string );
procedure conv.dToStr ( d:dword; buffer:string );
procedure conv.qToStr ( q:qword; buffer:string );
procedure conv.tbToStr( tb:tbyte; buffer:string );

procedure conv.strToh( s:string; index:dword ); returns( "al" );
procedure conv.strTow( s:string; index:dword ); returns( "ax" );
procedure conv.strTod( s:string; index:dword ); returns( "eax" );
procedure conv.strToq( s:string; index:dword ); returns( "edx:eax" );
```

10. Some people mistakenly refer to this process as "parsing." However, parsing actually means to figure out the meaning of the string, not just break it up into a sequence of components.

These routines are the general-purpose hexadecimal conversion routines. They convert their first parameter to a string of characters and store those character into the string variable passed as the second parameter. The string's maximum length must be large enough to hold the full result or an exception will occur.

The *conv.hToStr* function always creates a string that is two characters long. If the value of the *h* parameter is less than $10, then the string contains a leading zero. Similarly, the remaining functions always produce strings that are four (*conv.wToStr*), eight (*conv.dToStr*), 16 (*conv.qToStr*), or 20 (*convtbToStr*) characters long. If you do not wish to have leading zeros in the string, you must explicitly remove them yourself after the conversion.

The *conv.strToh, conv.strTow, conv.strTod,* and *conv.strToq* functions convert their string parameter (*s*) to a binary integer and leave the result in AL (*conv.strToh*), AX (*conv.strTow*), EAX (*conv.strTod*), or EDX:EAX (*conv.strToq*). The second parameter, *index*, specifies the starting position in the string for the translation. Typically, you would specify zero to begin the conversion at the first character position in the string; however, if the string appears after the first character position, you may specify the starting position via the *index* parameter.

Note that there is no *conv.strTotb* function that converts a string to a ten-byte value. If you need such a routine, you will have to write it yourself. Fortunately, you will rarely need such a routine and if you do require it, it is easy to write (see the source code for the *conv.atoh* function for details).

These functions will raise a conversion error exception if the string (beginning at position *index*) does not begin with a valid hexadecimal character or the sequence of hexadecimal characters ends with an invalid delimiter (you can select the valid delimiters, by default this set includes spaces, commas, semicolons, tabs, returns, and the end of string). If the value is too large to fit into the destination parameter, these functions will raise an overflow exception.

Program 1.23 Demonstration of the Hexadecimal Conversion Routines

1.3.23.2 Integer Conversion Routines

```
procedure conv.i64ToStr( q:qword; width:int32; fill:char; buffer:string );
procedure conv.i32ToStr( d:int32; width:int32; fill:char; buffer:string );
procedure conv.i16ToStr( w:int16; width:int32; fill:char; buffer:string );
procedure conv.i8ToStr ( b:int8; width:int32; fill:char; buffer:string );

procedure conv.u64ToStr( q:qword; width:int32; fill:char; buffer:string );
procedure conv.u32ToStr( d:uns32; width:int32; fill:char; buffer:string );
procedure conv.u16ToStr( w:uns16; width:int32; fill:char; buffer:string );
procedure conv.u8ToStr ( b:uns8; width:int32; fill:char; buffer:string );

procedure conv.strToi8( s:string; index:dword ); returns( "al" );
procedure conv.strToi16( s:string; index:dword ); returns( "ax" );
procedure conv.strToi32( s:string; index:dword ); returns( "eax" );
procedure conv.strToi64( s:string; index:dword ); returns( "edx:eax" );

procedure conv.strTou8( s:string; index:dword ); returns( "al" );
procedure conv.strTou16( s:string; index:dword ); returns( "ax" );
procedure conv.strTou32( s:string; index:dword ); returns( "eax" );
procedure conv.strTou64( s:string; index:dword ); returns( "edx:eax" );
```

These functions translate an eight, 16, 32, or 64-bit value to a string holding the decimal representation of that value. There are two sets of routines listed above; the *conv.uXXXX* functions translate unsigned inte-

ger values to their string equivalents, the *conv.iXXXX* routines convert signed integer values to their string equivalent.

The first parameter (*q, d, w,* or *b*) specifies the value to translate. The last parameter (*buffer*) specifies the string variable into which these functions store their converted result. This string must have storage alloctaed for it and it must be large enough to hold the converted data or these routines will raise a string overflow exception.

The second and third parameters (*width* and *fill*, respectively) specify the output format. The *width* parameter specifies the minimum field width; that is, the minimum length of the string that these functions produce. If the absolute value of this parameter is less than the minimum number of decimal characters these functions need to represent the string data, then these functions ignore the *width* and *fill* parameters. However, if the exact number of characters necessary to represent the integer value is less than the absolute value of the *width* parameter, then these functions always create a string that is exactly *width* characters long. If width's absolute value is greater than the length of the decimal string, then these functions always create a string whose length is exactly *abs(width)* characters long. These conversion functions store the value of the *fill* parameter in the extra character positions in the string. If *width's* value is positive, then these functions right justify the numeric value in the string (that is, the *fill* characters appear at the beginning of the string). On the other hand, if *width's* value is negative, then these functions left justify the value in the string (that is, the *fill* characters appear at the end of the string).

The *conv.strTou8, conv.strTou16, conv.strTou32,* and *conv.strTou64* functions convert their string parameter (*s*) to an unsigned binary integer and leave the result in AL (*conv.strTou8*), AX (*conv.strTou16*), EAX (*conv.strTou32*), or EDX:EAX (*conv.strTou64*). The *conv.strToi8, conv.strToi16, conv.strToi32,* and *conv.strToi64* functions convert their string parameter (*s*) to a signed binary integer and leave the result in AL, AX, EAX, or EDX:EAX (respectively).

The second parameter, *index*, specifies the starting position in the string for the translation. Typically, you would specify zero to begin the conversion at the first character position in the string; however, if the string appears after the first character position, you may specify the starting position via the *index* parameter.

These functions will raise a conversion error exception if the string (beginning at position *index*) does not begin with a valid decimal character or, in the case of the signed integer conversions, a single minus sign. These functions will also raise that exception if the sequence of decimal characters ends with an invalid delimiter (you can select the valid delimiters, by default this set includes spaces, commas, semicolons, tabs, returns, and the end of string). In the case of the signed integer conversions, the string must contain at least one decimal digit (that is, the string cannot consist of the single character '-'). If the value is too large to fit into the destination parameter, these functions will raise an overflow exception.

Program 1.24 Demonstration of the Integer Conversion Routines

1.3.23.3 Floating Point Conversion Routines

```
procedure conv.r80ToStr
(
   r80: real80;
   width: uns32;
   decimalpts: uns32;
   fill: char;
   buffer: string
);

procedure conv.r64ToStr
```

```
(
    r64: real64;
    width: uns32;
    decimalpts: uns32;
    fill: char;
    buffer: string
);

procedure conv.r32ToStr
(
    r32: real32;
    width: uns32;
    decimalpts: uns32;
    fill: char;
    buffer: string
);

procedure conv.e80ToStr
(
    r80: real80;
    width: uns32;
    fill: char;
    buffer: string
);

procedure conv.e64ToStr
(
    r64: real64;
    width: uns32;
    fill: char;
    buffer: string
);

procedure conv.e32ToStr
(
    r32: real32;
    width: uns32;
    fill: char;
    buffer: string
);

procedure conv.strToFlt( fpStr:string; index:dword ); returns( "st0" );
```

The *conv.rXXToStr* functions convert a floating point value (32, 64, or 80 bits) to the decimal representation of that value and store the result in the string *buffer*. The *r32, r64,* and *r80* parameters specify the floating point value to convert to a string.

The absolute value of the *width* parameter specifies the field width; these routines create a string that contains exactly this many characters. If this value is positive, these functions right justify the value in the string (padding any leading characters with the *fill* character). If this value is negative, then these routines left justify the value (padding any following characters with the *fill* character). Note that the width value must include space for a leading minus sign (if the number is negative) as well as a decimal point character.

The *decimalpts* parameter specifies the number of digits to display to the right of the decimal point. Note that this value does not affect the size of the string that these conversion routines produce. Therefore, *decimalpts*' value must be less than *width* (at least *width-1* or *width-2* if the value is negative).

Example:

Assume you have the value 12345.6789. A *width* of eight with *decimalpts* equal to two produces the string "12345.68" (notice that these conversion routines automatically round up the value).

© 2001, By Randall Hyde

If the number of output digits plus the decimal point and any necessary sign is less than the field width, then these conversion routines fill the extra print positions with the *fill* character. Typically, the *fill* parameter contains the space character; however, in certain special circumstances you may want to specify a different character. For example, when printing checks it is common to right justify the value and print leading asterisk characters ("*") so that it is more difficult to tamper with the check.

The *conv.eXXToStr* routines also convert a floating point value to a string of characters. These routines convert the data using scientific notation, e.g., "1.23456789e+4". Since the conversion fixes the position of the decimal point between the most significant and the next most significant digits, there is no need for a *decimalpts* parameter in these functions. Other than the conversion format and the lack of the decimalpts parameter, these routines behave identically to the *conv.rXXToStr* routines.

The *width* parameter in the *conv.rXXToStr* and *conv.eXXToStr* function calls behaves a little differently than this same parameter in the integer conversion routines. For the integer conversions, the *width* parameter specifies the minimum field width. If the integer value requires more than the number of digits than *width* specifies, the integer conversion routines will go ahead and create a string large enough to correctly represent the intetger value. For floating point conversions, however, this parameter specifies the exact field width. If the conversion routines cannot fit the value into a string of this size, then the floating point conversion fill the string (of length *width*) with asterisks.

The *conv.strToFlt* function converts a string containing a legal floating point constant into a binary floating point value. The *fpStr* parameter, begining at character position *index*, must contain some legal representation of a floating point constant. This string may contain a floating point constant in either decimal notation or scientific notation. Note that integer constants (i.e., decimal representation without the decimal point) are also legal. The *conv.strToFlt* function will convert such integer constants to their real equivalent.

Program 1.25 Demonstration of the Floating Point Conversion Routines

1.3.23.4 Miscellaneous String Conversion Routines

```
procedure conv.cStrToStr( var buffer:byte; dest:string );
procedure conv.a_cStrToStr( var buffer:byte ); returns( "eax" );
procedure conv.roman( Arabic:uns32; rmn:string );
procedure conv.a_roman( Arabic:uns32 ); returns( "eax" );
```

The conv module in the HLA Standard Library contains a few additional string conversion routines that are worthy of mention. One pair of routines converts between string formats and another pair is a set of numeric conversion routines that convert integer values to their Roman numeral equivalent.

The *conv.cStrToStr* and *conv.a_cStrToStr* functions convert C strings (i.e., zero terminated strings) to an HLA compatible string format. This conversion is especially useful when interfacing HLA code with code written in C/C++ (e.g., Win32 calls). The *buffer* parameter to these two functions must be the address of the first byte of the string you wish to convert. This must be the address of zero or more characters that end with a byte containing zero.

The *conv.cStrToStr* function computes the length of this string and then copies the string data to the *dest* string that the second parameter specifies. The destination string must have sufficient storage to hold the converted string. If the length of the zero terminated string exceeds the maximum possible length of the destination string, then this function raises a string overflow exception.

The *conv.a_cStrToStr* function automatically allocates storage for the conversion on the heap. It then copies the string to this new storage and returns a pointer to the new string in the EAX register. As typical for routines that call *stralloc*, you should call *strfree* to deallocate the storage for this string when you finish using it.

The *conv.roman* and *conv.a_roman* routines convert an unsigned decimal integer to a string providing the Roman numeral representation of the value. Because of limitations of the Roman numbering system and the ASCII character set, these routines will only convert values in the range 1..5500 to a correct string. The *conv.roman* function converts its *Arabic* parameter to an appropriate string and stores this at the location the *rmn* parameter specifies. As usual, you must preallocate storage for *rmn* and it must have sufficient storage to hold the result. The *conv.a_roman* function allocates storage for the string on the heap and returns a pointer to this string in EAX. You should call *strfree* to deallocate this storage when you are done with the string.

You may wonder why the HLA Standard Library contains a conversion from integer to the Roman number system; especially given the limitations of the Roman numbering system. The main reason for this conversion is because most texts use the Roman numbering system for tables of contents and prefaces. Also, Roman numbers are common in copyright notices (in films and videos). Hence, this conversion is useful once in a while. Although conversion from integer to the Roman system has some modern use, going the other way (Roman numeral input to integer) doesn't appear to be useful in the modern world, hence the HLA Standard Library does not provide this conversion for you (warning: this does make an excellent programming project, however).

In addition to these four string conversion routines, other modules in the HLA Standard Library provide string conversion. For example, the Date and Time modules contain routines that convert between internal representation and string representation of dates and times. The Standard Library provides other conversions as well. Please consult the Standard Library documentation for more details.

Program 1.26 Examples of the Miscellaneous Conversion Functions

1.3.24 Data Formatting String Routines

```
procedure str.catbool( b:boolean; dest:string );
procedure str.catcsize( c:char; width:int32; fill:char; dest:string );
procedure str.catc( c:char; dest:string );
procedure str.cats( s:string; dest:string );
procedure str.catssize( s:string; width:int32; fill:char; dest:string );
procedure str.catcset( c:cset; dest:string );

procedure str.cath( b:byte; dest:string );
procedure str.catw( w:word; dest:string );
procedure str.catdw( d:dword; dest:string );
procedure str.catqw( q:qword; dest:string );

procedure str.catu8( u8:byte; dest:string );
procedure str.catu8size( u8:byte; width:int32; fill:char; dest:string );
procedure str.catu16( u16:word; dest:string );
procedure str.catu16size( u16:word; width:int32; fill:char; dest:string );

procedure str.catu32( u32:dword; dest:string );
procedure str.catu32size( u32:dword; width:int32; fill:char; dest:string );

procedure str.catu64( u64:qword; dest:string );
procedure str.catu64size( u64:qword; width:int32; fill:char; dest:string );

procedure str.cati8( i8:byte; dest:string );
procedure str.cati8size( i8:byte; width:int32; fill:char; dest:string );

procedure str.cati16( i16:word; dest:string );
```


© 2001, By Randall Hyde

```
procedure str.cati16size( i16:word; width:int32; fill:char; dest:string );

procedure str.cati32( i32:dword; dest:string );
procedure str.cati32size( i32:dword; width:int32; fill:char; dest:string );

procedure str.cati64( i64:qword; dest:string );
procedure str.cati64size( i64:qword; width:int32; fill:char; dest:string);

procedure str.catr80(
    r:real80;
    width:int32;
    decpts:uns32;
    fill:char;
    dest:string
);

procedure str.catr64
(
    r:real64;
    width:int32;
    decpts:uns32;
    fill:char;
    dest:string
);

procedure str.catr32
(
    r:real32;
    width:int32;
    decpts:uns32;
    fill:char;
    dest:string
);

procedure str.cate80( r:real80; width:int32; dest:string );
procedure str.cate64( r:real64; width:int32; dest:string );
procedure str.cate32( r:real32; width:int32; dest:string);
```

This (rather large) set of string conversion routines are useful for building up larger strings from smaller components. Their intent is to mimic the *stdout.putXXX* routines found in the HLA Standard Library. In fact, if you look you'll discover that for every *stdout.putX* routine, there is a corresponding *str.catX* routine. Therefore, you can use this routines to construct strings in much the same way you use the *stdout.putX* routines to send data to the standard output device. Because of the similarity of these routines to the *stdout.putX* functions and the other string conversion functions, this section will not discuss each of these functions in order to keep this section down to a reasonable size. If you have questions about a particular function, check out the description of the corresponding string conversion function in one of the previous sections.

As their names suggest, these routines concatenate the converted string to the end of an existing string. That is the principle difference between these functions and the other string conversion functions appearing in this chapter. For example, the *str.cath* function converts its byte parameter (*b*) to a string of two hexadecimal characters and concatentates this to the end of the *dest* string (the second parameter). The *dest* string must have sufficient storage to hold its original string plus the new characters or this function raises the string overflow exception.

Note that there is no *str.a_cath* routine (nor any other "a_" versions of the above functions). That is because you usually call several of these functions in quick succession to build up a single string. Constantly allocating new storage and copying the string data around would be very inefficient. That is why there are no versions of these routines that allocate the storage automatically. If you need an "a_" version of one of these routines, it's very easy to write a "wrapper" function that provides this capability (see the exercises).

The most common use of these functions is to build up a large string by several different calls, with each call contributing a portion of the overall string. As an example, consider the following sequence:

```
var
    s:string;
        .
        .
        .
    mov( stralloc( 256 ), s ); // Allocate storage for the string.
    str.cpy( "I = ", s );
    str.cati32( I, s );
    str.cats( ", J = ", s );
    str.cati32( J, s );
    str.cats( ", R = ", s );
    str.catr64( R, 10, 2, ' ', s );

    // Now, s contains something like "I = 10, J = 20, R = 1234567.89"
```

Building up a complete string via a sequence of calls, as the above example demonstrates, is a bit of a pain. Therefore, the HLA string module provides a special macro that makes constructing strings like this very easy: the *str.put* macro. You use the *str.put* macro much the same way that you use the *stdout.put* macro. However, rather than write the data to the standard output, the *str.put* macro stores the data it converts into a string variable. The first parameter of the *str.put* macro specifies the destination for this string, e.g., you could encode the previous sequence as follows:

```
    mov( stralloc( 256 ), s );
    str.put( destStr, "I = ", I, ", J = ", J, ", R = ", R:10:2 );
```

The destStr parameter must be a string variable and it must point at a string object in memory that is large enough to hold the entire string the *str.put* invocation produces. In the example above, a 256 character string is certainly sufficient to hold this data (in fact, a much shorter string would probably be okay if this is the only data the program will write to it).

One of the most important uses of the *str.put* macro (and, in general, of the *str.catXXX* routines) is to format data for various Win32 API calls. Most Windows functions that write data to the video display require string parameters. If you wish to display other types of data in a dialog box or other window, you will have to first convert that text to a string and then pass the string along to Windows. The *str.put* macro and the *str.catXXX* functions provide a convenient way to do this, leveraging your knowledge of the HLA Standard Library's *stdout* functions. The following example program demonstrates the use of *str.put* in this capacity.

Program 1.27 Using the *str.put* Macro to Prepare Data for a Windows Dialog Box

1.4 The HLA Conversions Module

```
conv.setUnderscores( OnOff );  // OnOff is a boolean value
conv.getUnderscores();
```

When converting numeric data types to strings, HLA offers the option of inserting underscores at appropriate places in the numbers (i.e., where you would normally expect a comma to go). This feature in the library can be activated or deactivated with the *conv.setUnderscores* function. You can test the current state of the underscore conversion by calling *conv.getUnderscores* which returns the boolean result in EAX (true means underscores will be output).

```
conv.setDelimiters( Delims );          // Delims is a character set value,
conv.getDelimiters( var Delims );      //  see the next chapter for details.
```

During the conversion from string to a numeric form, HLA will generate an exception if it encounters a character that is not a numeric digit in the specified base or the character is not an appropriate delimiter character. By default, the delimiter characters are members of the following set:

```
    Delimiters: cset :=
        {
            #0, #9, #10, #13,
            ' ',
            ',',
            ';',
            ':'
        };
```

You can obtain the current delimiter set by calling the *conv.getDelimiters* function. You can change the current set of delimiter characters by calling the *conv.setDelimiters* function.

```
conv.hToStr ( b, strVar );    // b is a byte value
conv.wToStr ( w, strVar );    // w is a word value
conv.dToStr ( d, strVar );    // d is a dword value
conv.qToStr ( q, strVar );    // q is a qword value
conv.tbToStr( tb, strVar );   // tb is a tbyte value
```

These routines are the general-purpose hexadecimal conversion routines. They convert their first parameter to a string of hexadecimal characters and store those character into the string variable passed as the second parameter. The string's maximum length must be large enough to hold the full result or an ex.StringOverflow exception will occur. Bytes require exactly two characters, words require four characters, dwords require eight characters, qwords require 16 characters, and tbytes require 20 characters. You must allocate a sufficient amount of storage for the string (e.g., by calling *stralloc*) prior to calling these functions.

```
conv.i64ToStr( q, width, fill, strVar );
conv.i32ToStr( d, width, fill, strVar );
conv.i16ToStr( w, width, fill, strVar );
conv.i8ToStr ( b,  width, fill, strVar );

conv.u64ToStr( q, width, fill, strVar );
conv.u32ToStr( d, width, fill, strVar );
conv.u16ToStr( w, width, fill, strVar );
conv.u8ToStr ( b,  width, fill, strVar );
```

Notes: *b* must be a byte-sized object, *w* must be a word-sized object, *d* must be a dword-sized object, *q* must be a qword-sized object; *width* is an int32 value; *fill* must be a character value; *strVar* must be a string variable preallocated to a sufficient length.

These routines translate their first parameter to a string that has a minimum of "width" print positions. If the number would require fewer than "width" print positions, the routines copy the "fill" character to the remaining positions in the destination string. If width is positive, the number is right justified in the string. If width is negative, the number is left justified in the string. These routines store the converted value into the *strVar* string parameter. The *strVar* variable must point at an allocated string with sufficient storage for all the characters in the number. The minimum length of this string should be either width or the number of character positions the conversion requires; note that the *conv.XXsize* functions (described next) will compute the minimum length requirements for you.

These functions are quite similar to the *stdout.putiXXX* and *stdout.putuXXX* routines except they write their data to a string rather than the standard output. See the discussion of these *stdout* routines for more details.

```
conv.i8Size();       // AL contains value to test.
conv.i16Size();      // AX contains value to check.
conv.i32Size();      // EAX contains value to test.
conv.i64Size( q );   // q must be a qword value.
conv.u8Size();       // AL contains value to check.
conv.u16Size();      // AX contains value to test.
conv.u32Size();      // EAX contains value to check.
conv.u64Size( q );   // q must be a qword value.
```

These routines return the size, in character positions, it would take to print the integer (signed or unsigned) passed in the AL register (*conv.i8Size*/*conv.u8Size*), AX register (*conv.i16Size*/*conv.u16Size*), EAX register (*conv.i32Size*/*conv.u32Size*), or in the *q* parameter (*conv.i64Size*/*conv.u64Size*). They return their value in the EAX register. The count includes room for a minus sign if the number is negative (*conv.iXXSize* routines, only). The value these functions returns in the EAX register specifies the minimum length of the string that you should pass to the integer to decimal conversion routines. If you pass the value these functions return in EAX as a parameter to the stralloc function, you will get a string of the appropriate size for the *conv.iXXToStr* and *conv.uXXToStr* functions.

conv.r80ToStr
(
r80: real80;
width: uns32;
decimalpts: uns32;
fill: char;
buffer: string
)

conv.r64ToStr
(
r64: real64;
width: uns32;
decimalpts: uns32;
fill: char;
buffer: string
)

conv.r32ToStr
(
r32: real32;
width: uns32;
decimalpts: uns32;
fill: char;
buffer: string
)

These routines convert single, double, and extended floating point values into their string equivalents. These routines display the number in decimal notation. The decimalpts parameter specifies the number of digits to the right of the decimal point, the remaining parameters are equivalent to those in the iXXToStr routines.

```
conv.e80ToStr
(
        r80:        real80;
        width:      uns32;
        fill:       char;
        buffer:     string
)

conv.e64ToStr
(
        r64:        real64;
        width:      uns32;
        fill:       char;
        buffer:     string
)

conv.e32ToStr
(
        r32:        real32;
        width:      uns32;
        fill:       char;
        buffer:     string
)
```

Similar to the rXXToStr routines above, except these routines output their values in scientific notation.

```
conv.strToi8( s:string; index:dword )
conv.strToi16( s:string; index:dword )
conv.strToi32( s:string; index:dword )
conv.strToi64( s:string; index:dword )

conv.strTou8( s:string; index:dword )
conv.strTou16( s:string; index:dword )
conv.strTou32( s:string; index:dword )
conv.strTou64( s:string; index:dword )

conv.strToh( s:string; index:dword )
conv.strTow( s:string; index:dword )
conv.strTod( s:string; index:dword )
conv.strToq( s:string; index:dword )

conv.strToFlt( s:string; index:dword )
```

These routines convert characters from a string to the corresponding numeric forms. The index parameter is the index of the position in the string where conversion begins.

The eight-bit routines return their result in AL; the 16-bit routines return their result in AX; the 32-bit routines return their result in EAX; and the 64-bit routines return their result in EDX:EAX. The StrToFlt routine returns its value in ST0.

conv.cStrToStr(var buffer:byte; dest:string)

This function converts a "C-String" (zero terminated sequence of characters) to an HLA string. The "buffer" parameter points at the zero terminated string, cStrToStr stores the resulting string into the dest operand.

Note: a function that converts HLA strings to zero-terminated strings is not necessary since HLA strings are already zero-terminated and the string variable points at the first character of the string; hence, HLA strings are already compatible with C-Strings.

conv.a_cStrToStr(var buffer:byte)

This function also converts zero terminated strings to HLA strings. However, instead of storing the string data at a specified location, this routine allocates storage for the string on the heap and returns a pointer to the new string in EAX. You should use strfree() to clean up the storage after you are done with the string.

conv.roman(Arabic:uns32; rmn:string)

This procedure converts the specified integer value (Arabic) into a string that contains the Roman numeral representation of the value. Note that this routine only converts integer values in the range 1..3,999 to Roman numeral form. Since ASCII text doesn't allow overbars (that multiply roman digits by 1,000), this function doesn't handle really large Roman numbers. A different character set would be necessary for that.

conv.a_roman(Arabic:uns32)

Just like the routine above, but this one allocates storage for the string and returns a pointer to the string in the EAX register.