37 Zero-terminated String Functions (zstring.hhf)

Although HLA's string format is more efficient (with respect to speed) than the zero-terminated string format that languages like C, C++, and Java use, HLA programs must often interact with code that expects zero-terminated strings. Examples include HLA (assembly) code you like with C/C++/Java programs and calls you make to operating systems like Windows and Linux (that expect zero terminated strings). Therefore, the HLA Standard Library provides a limited amount of support for zero-terminated strings so it can efficiently interact with external code that requires such strings.

When passing read-only string data to some code that expects a zero-terminated string, HLA's string format is upwards compatible with zero-terminated strings. No conversion is necessary. An HLA string variable holds the address of a sequence of characters that end with a zero byte (the zero-terminated format). So as long as the code you're calling doesn't attempt to write any data to the string object, you can pass HLA string objects to functions and procedures that expect zero-terminated strings.

If the procedure or function you're calling stores data into a destination string variable, then you generally should not pass an HLA string to that function. There are two problems with this: first, the function does not check the HLA string's maximum length field to ensure that string overflow does not occur; second, the external function does not properly set the HLA string's length field before returning. Furthermore, the external code may create it's own string data in some buffer and does not even allocate space for HLA's maximum length and dynamic length fields. To workaround these limitations, HLA provides various procedures in the Standard Library that manipulate zero-terminated strings so your programs can effectively communicate with external code that operates on such strings.

Before describing the support functions that HLA provides for zero-terminated strings, it's probably worthwhile to first discuss how one writes code that comfortably co-exists with such strings. As noted above, there are three major problems one must deal with when external code processes zero-terminated strings. We'll deal with these issues one at a time.

The first problem is that the external code does not check the maximum string length field before writing character data to a string object. Therefore, the external code cannot determine if a buffer overflow will occur when that function extends the string's length. Algorithms that depend upon the string function raising an exception when a buffer overflow occurs will not work properly when calling external code that manipulates zero-terminated strings. The solution to this problem is the same as the solution in C and C++: the programmer must take the responsibility of ensuring that there is sufficient buffer space available to hold the string the external function produces. Exactly how much space you must allocate as a maximum varies on a call by call basis, but usually you can pick a sufficiently large value that is safe and preallocate storage for an HLA string whose maximum length satisifies the program's requirements. Note that most operating system API functions that return variable length strings will let you specify a maximum length parameter so the OS will not overflow your string buffer; well-written library routines and other code that create variable length zero-terminated strings and generally provide this same functionality.

The second problem, the fact that the external code that manipulates the string's data does not update HLA's string length field, is solvable by computing the length of the zero-terminated string upon return from the external code and updating the length field yourself. A convenient way to handle this operation is to write a *wrapper function* that you call from your code. The wrapper function calls the external code and then computes and updates the HLA string length field before returning to the original caller. This saves having to compute the length on each and every invocation of the external code. The HLA Standard Library provides a string length function that efficiently computes the length of a zero-terminated string. You can call this function upon return from the external code and then store the return result into the HLA dynamic length field.

Some external functions may create their own zero-terminated strings rather than store their string data in a buffer you supply. Such functions will probably not allocate storage for the dynamic and maximum length fields that the HLA string format requires. Therefore, you cannot directly use such string data as an HLA string in your assembly code. There are two ways to handle such string data: (1) copy the zero-terminated string to an HLA string and then manipulate the HLA string, or, (2) process the zero-terminated string using functions that directly manipulate such strings. The HLA strings module provides a set of zero-terminated string functions that let you choose either mechanism. The choice of method (1) or (2) depends entirely upon how you intend to use the string data upon return to your HLA code. If you're going to do considerable string manipulation on that string data within your HLA code (and you want to use the full set of HLA string and pattern matching functions on the string data), it makes a lot of sense to first convert the string to the HLA format. On the other hand, if you're going to do very little manipulation, or if the external function expects your code to update the string data in place (so it can refer to a modified version of the original string data at the original address the external code allocates), then it's probably best to manipulate the string data in-place using a set of zero-terminated string functions. If you need to do considerable string manipulation on some data, but the external code expects you to leave the manipulated string in the original buffer it allocates, you can convert the string to an HLA string, do the modification, and then copy the resulting string back into the original buffer; however, all this copying can be expensive, so you should be careful about using this approach.

The HLA Standard Library provides a small handful of important zero-terminated string functions. This set certainly isn't as extensive as the set of functions available for HLA strings, nor is it as extensive as the set of functions available, for example, in the C Standard Library. However, this small set of functions will probably cover 90-95% of the requirements you'll have for processing zero-terminated strings in HLA code. Generally, if you need other functionality, you can obtain it by calling C Standard Library functions from your HLA code or by first converting the string to an HLA string (and then copying the data back to the original buffer, if necessary). The following subsections describe the functions that the HLA Standard Library provides to support zero-terminated strings.

37.1 ZStrings Module

To use the zero-terminated string functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

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

37.2 Zstring Functions

`procedure zstr.len( zstr:zstring ); @returns( "eax" );`

The single parameter is the address of a zero-terminated string. This function returns the length of that string in the EAX register.

Note that the *zstr.len* function has a single untyped reference parameter. Generally, you'd pass the name of a buffer variable as the parameter to this function. If the address of the zero-terminated string appears in a register, you'll need to use one of the following three invocations to call this function:

```
// Manual invocation- assumes the string pointer is in EBX:

    push( ebx );
    call zstr.len;
    << length is in EAX >>
        .
        .
        .
    zstr.len( [ebx] );  // zlen expects a memory operand
        .
        .
        .
    zstr.len( val ebx ); // Tell HLA to use value of ebx.
```

The *zstr.len* function is especially useful for updating the length field of an HLA string you've passed to some external code that generates a zero-terminated string. Consider the following code that updates the length upon return from an external function:

```
// Allocate sufficient storage to hold the string result the external
// code will produce.  1024 was chosen at random for this example, you'll
// have to pick an appropriate value based on the size of the string
// the external procedure in your code produces.

  str.alloc( 1024 );
  mov( eax, strVar );
      .
      .
      .
  externalFunction( strVal );  // externalFunction overwrites strVal data.
  zstr.len( strVal );          // Compute the result string's length
```

```
    mov( strVar, ebx );          // Get pointer to string data.
    if( eax > (type str.strRec [ebx]).MaxStrLen )) then

        // If there was a string overflow, the overflow may
        // have wiped out some important data somewhere, so
        // it may be too late to raise this exception.  However,
        // better late than not notifying the caller at all.
        // Because the buffer overflow may have corrupted the application's
        // data, the application should attempt to terminate as
        // gracefully as possible at this point.

        raise( ex.StringOverflow );

    endif;

    // Okay, the string didn't overflow the buffer, update the
    // HLA string dynamic length field:

    mov( eax, (type str.strRec [ebx]).length );
```

HLA high-level calling sequence examples:

```
    zstr.len( zstrValue );
    mov( eax, zlen );
```

HLA low-level calling sequence examples:

```
    pushd( &zstrValue );
    call zstr.len;
    mov( eax, zlen );
        .
        .
        .
    lea( eax, zStrVar );
    push( eax );
    call zstr.len;
    mov( eax, zlen2 );
```

procedure zstr.zcmp(zsrc1:zstring; zsrc2:zstring); @returns("eax");

The *zstr.zcmp* function compares two zero-terminated strings and returns the comparison results in the EAX register and in the x86 flags. This comparison function sets the condition code bits so you can use the standard unsigned condition instructions (jump and set instructions) immediately upon return to test for less than, less than or equal, equal, not equal, greater than, or greater than or equal. This function also returns -1 ($FFFF_FFFF), zero, or one in EAX to indicate less than, equal, or greater than (respectively). Note that this function compares *zsrc1* to *zsrc2*. Therefore, this function returns -1 if zsrc1 < zrc2, zero if zsrc1 = zsrc2, and one if *zsrc1* > *zsrc2*.

This function is especiially useful for comparing two zero-terminated strings that some external code returns to your HLA program if you don't need to do any further manipulation of the string data. This function is also useful for comparing an HLA string against a zero-terminated string (since HLA strings are zero terminated). Technically, you could use this function to compare two HLA strings (since they are zero-terminated), but the standard HLA string comparison functions are probably more efficient for this purpose.

HLA high-level calling sequence examples:

```
    zstr.zcmp( zstr1, zstr2 );
    if( @ae ) then // zstr1 >= zstr2
          .
          .
          .
       endif;
       zstr.zcmp( someZStr, "Hello World" );
       mov( eax, cmpResult );
```

HLA low-level calling sequence examples:

```
    lea( eax, SomeCharBuffer );
    push( eax );
    push( zStrVar );// Note: zstring vars are pointer vars
    call zstr.zcmp;
    jnae notAE;
          .
          .
          .
       notAE:

       push( someZStr );
       push( HelloWorldStr );
       call zstr.zcmp;
       mov( eax, cmpResult );
```

procedure zstr.cpy(src:zstring; dest:zstring);

The *zstr.cpy* function copies one zero-terminated string to another. The destination buffer must be large enough to hold the source string and it is the caller's responsbility to ensure this. The *zstr.cpy* routine has no way to determine the maximum size of the destination buffer, so it cannot check for buffer overflow (this is typical for zero-terminated string functions).

Since HLA strings are zero-terminated, you can use this function to copy an HLA string to a zero-terminated string:

```
// Assumptions: hlaString is the name of an HLA String variable and
// destZStr is the name of an array of characters or byte array.

zstr.cpy( hlaString, destZStr );
```

Of course, you can also use the *zstr.cpy* function to copy one zero-terminated string to another. You'd typically use *zstr.cpy* in this capacity to copy a string returned by one external function to a buffer for use by another external function that expects a zero-terminated string.

procedure zstr.cat(src:zstring; dest:zstring);

This function concatenates one zero-terminated string to the end of another. The caller must ensure that the destination buffer is large enough to hold the resulting string; the *zstr.cat* function has no way to verify the size of the destination buffer, so it cannot check for buffer overflow (this is typical for zero-terminated string functions).

This string is useful for manipulating zero-terminated strings some external code provides without the overhead of first converting the strings to HLA strings. If you call two external functions that return zero-terminated strings and you need to pass their concatenated result to some other external function that expects a zero-terminated string, and there is no string manipulation in your HLA code, then using *zstr.cat* is more efficent than converting the strings to an HLA string and using the HLA string concation function.

When using this function, don't forget that it's parameters are untyped reference parameters. When passing the address of a buffer variable you may specify the name of the buffer directly. However, when passing a pointer to the buffer, you'll probably need to use the VAL operator to tell HLA to pass the pointer's value rather than the pointer's address. Here are some examples of *zstr.cat* invocations:

```
static
  buffer1   :char[256];
  buffer2   :char[254];
  bufptr1   :zstring;
  bufptr2   :zstring;
      .
      .
      .
  lea( eax, buffer1 );
  mov( eax, bufptr1 );
  lea( eax, buffer2 );
  mov( eax, bufptr2 );
      .
      .
      .
  zstr.cat( bufptr1, bufptr2 );
  zstr.cat( buffer2, edi );
  zstr.cat( bufptr2, esi );
```

You can also use the *zstr.cat* procedure to copy data from an HLA string to a zero-terminated string:

```
static
  hlaStr    :string;
  zs           :char[256];
  zPtr      :zstring;
      .
      .
      .
  lea( eax, zs );
  mov( eax, zPtr );
      .
      .
      .
  zstr.zcat( hlaStr, zPtr );
  zstr.zcat( hlaStr, esi );
```

It really does not make any sense to specify an HLA string variable as the destination operand. *zstr.cat* does not update the HLA string's length field, so if you supply an HLA string as the destination operand, the *zstr.cat* procedure may corrupt the HLA string, forcing you to manually compute the length yourself. If you need to copy a zero-terminated string to an HLA string, use the *zstr.cat* function instead.