16 The File System Module (filesys.hhf)

The filesys functions perform file system manipulations (as opposed to the fileio package that operates on files).

16.1 Filename and Pathname String Functions

These functions test and manipulate pathname strings. Most of them do not do any actual file system access, they simply modify a string passed to them.

The file and pathname string functions are relatively OS-neutral. As long as you avoid a few OS-specific filename features (such as Win32 drive letter prefixes and the use of backslash [‘\’] characters in Unix-like OS filenames), you’ll find that these functions are highly portable between various operating systems. These functions automatically convert pathname separator characters to the native OS (except for the functions that explicitly produce Win32-style or Unix-style pathnames). So you can write applications that process pathnames and automatically work with whatever OS the application is running under.

This documentation will use the standard UNIX ‘/’ pathname separator character. Anywhere you see a "/" used in the following examples, just note that you can also use a "\" and the function will still work properly (you can even have a mixture of these two separator characters in the same pathname string and the function will accept this). Keep in mind that most filesys pathname functions will convert all ‘/’ and ‘\’ characters to the native directory separator character; take care if you expect to process Win32 filenames under a UNIX-like OS and you need to keep the Win32 separator characters.

For the purposes of the filesys string functions, a pathname is considered to contain up to five components: a UNC prefix, a path component, a filename component, a basename component, and an extension. These components are not necessarily unique (that is, some of them overlap one another). Not all pathnames contain all of these components. Consider the following valid pathname string:

```
fsType://computerName/SharedFolder/path1/path2/base.ext

This example pathname contains the following components:
  UNC:      fsType://computerName/SharedFolder
  path:     fsType://computerName/SharedFolder/path1/path2
  filename:base.ext
  basename:base
  extensionext

UNC (Universal Naming Convention) names take two basic forms:
  //computerName/sharedFolderName
  fsType://computerName/sharedFolderName
```

UNC names contain an optional file system type name followed by a colon. All UNC names contain ‘//’ followed by a computer name which is then followed by a ‘/’ and a shared folder name.

Path components consist of everything to the left of the last ‘/’ appearing in a path name string. Note that a UNC component is also part of a path component. Indeed, if a UNC immediately precedes a filename, then the UNC sequence is the path component (note, however, that a path component may include other subdirectory names in addition to the UNC character sequence). If there is nothing to the left of the last (i.e., only) ‘/’ in a pathname string, then the "/" is the path component.

The filename component is everything appearing to the right of the last ‘/’ character in the pathname string, or the whole pathname string if there is no ‘/’ character in the pathname string.

The basename and extension components are part of the filename. If the filename contains at least one period and that period is not the first character of the filename, then everything to the left of the (last) period is the basename and everything to the right of the (last) period is the extension. If a filename contains multiple periods, then everything up to (but not including) the last period is the basename. If the filename contains only one period and it begins with that period, then the filename has no extension and the basename is equal to the filename. Likewise, if a filename contains no periods at all, it has no extension and the basename is equal to the filename.

```
procedure filesys.hasDriveLetter( pathname:string ); @returns( "@c" );
```

This function returns the Win32 drive letter if the pathname argument begins with a single alphabetic character, immediately followed by a colon (‘:’) and the colon is not immediately followed by a pair of slashes

(indicating a UNC name). Drive letters are Win32-specific, although this function can be called on any pathname string. If a drive letter is present, this function returns the drive letter in AL, converted to uppercase, and also returns with the carry flag set. If there is no drive letter at the beginning of the name (or if it looks like a UNC name), then this function returns with EAX containing zero and the carry flag clear.

Note that the function "returns" value for this function is "@c" (that is, the carry flag) and not "AL". This allows you to use the function call within a boolean expression (e.g., in an "if" statement) and test for true/false return values.

```
HLA high-level calling sequence examples:

if( filesys.hasDriveLetter( somePath )) then

  stdout.put( "Drive is ", (type char al), nl );
  str.delete( somePath, 0, 2 );// Delete the drive letter

endif;


HLA low-level calling sequence example:

  push( somePath );
  call filesys.hasDriveLetter;
  jnc noDriveLetter;

      push( somePath );
      pushd( 0 );
      pushd( 2 );
      call str.delete;

  noDriveLetter:
```

procedure filesys.hasExtension(pathname:string); @returns("@c");

This function returns true if the filename component of the pathname argumen contains an extension. An extension is the last part of a pathname following the last period in the filename. Note that if a filename begins with a period and that is the only period in the filename, then the following characters are not an extension (and the extension is the empty string for such a name). Note that periods found in the path to the filename are not considered when this function searches for the extension. Extensions only belong to filename components, not to path components.

Examples:

```
filesys.hasExtension( "/path/file.ext" );// true, extension = "ext"
filesys.hasExtension( "file" );// false
filesys.hasExtension( ".ext" );// false
filesys.hasExtension( "file.ext" );// true, extension = "ext"
filesys.hasExtension( "file.abc.ext" );// true, extension = "ext"
filesys.hasExtension( "..ext" );// true, extension = "ext"
filesys.hasExtension( "path.ext/file" );// false
```

The true/false result is returned both in the carry flag and in the EAX/AX/AL register. The carry flag is set if the argument has an extension, it is clear otherwise. Similarly, true (1) is returned in EAX/AX/AL if the argument has an extension, false (0) is returned otherwise.

```
HLA high-level calling sequence examples:

filesys.hasExtension( somePath );
mov( al, somePathHasExtension );
if( @c ) then

  << do something if somePath has an extension >>
```

```
endif;


HLA low-level calling sequence example:

  push( somePath );
  call filesys.hasExtension;
  jnc noExtension;

     << Do something if somePath has an extension>>

  noExtension:
```

procedure filesys.hasUncName(pathname:string); @returns("@c");

This function tests the pathname argument to see if it begins with a UNC (universal naming convention) pathname prefix. UNC prefixes take one of two forms (as far as this code is concerned):

```
//computername/sharedfolder/<path>
<type>://computername/sharedfolder/<path>
```

<type> can be any string of filename-compatible characters (length one or greater), such as ‘file’, ‘smb’, and so on. *<path>* may be any OS-compatible pathname of length zero or greater (up to the maximum length supported by the native OS). Portable code should not allow the pathname string (including the UNC) to except about 250 characters.

This function returns true or false in the carry flag indicating whether a UNC, if present, is syntactically correct. That is, this function returns carry clear if there is something that looks like a UNC but is syntactically illegal. Note that a pathname, without an explicit "//computername/sharedfolder/" prefix is still a syntactically correct pathname as far as this function is concerned. That is, this function returns true for pathnames like "name" or "/path/name" even though an explicit UNC item is not present.

This function returns information about the UNC prefix in the EAX register. If this function returns with EAX equal to zero and the carry set, then there is no UNC present in the pathname. If this function returns with EAX containing a value other than zero (and the carry flag set), then a UNC is present and EAX contains an offset into the string that is the start of the pathname just beyond the end of the UNC sequence (i.e., beyond the ‘/’ or ‘\’ that marks the end of the UNC name).

Note: on failure (carry = 0), EAX will be returned containing zero.

```
HLA high-level calling sequence examples:

if( filesys.hasUncName( somePath ) && eax <> 0 ) then

  str.delete( somePath, 0, eax );// Delete the UNC prefix

endif;


HLA low-level calling sequence example:

  push( somePath );
  call filesys.hasUncName;
  jnc noUNC;
  test( eax, eax );
  jz noUNC;

     push( somePath );
     pushd( 0 );
     push( eax );
     call str.delete;
```

```
    noUNC:
```

procedure filesys.hasPath(pathname:string); @returns("@c");

This function returns true if the pathname/filename argument contains a path component. Specifically, this function returns true if the pathname string contains any directory separator characters ('/' or '\'). True is returned in the carry flag (set) and in the EAX/AX/AL register (1). False is carry = 0 or the EAX/AX/AL register equals 0. Note that UNC prefixes immediately before a filename are considered 'paths' and this function will return true if a UNC is present.

Examples:

```
filesys.hasPath( "/path/file.ext" );// true
filesys.hasPath( "file" ); // false
filesys.hasPath( ".ext" ); // false
filesys.hasPath( "file.ext" );// false
filesys.hasPath( "file.abc.ext" );// false
filesys.hasPath( "//machine/folder/file" );// true
filesys.hasPath( "/" );       // true
```

HLA high-level calling sequence examples:

```
filesys.hasPath( somePath );
mov( al, somePathHasAPathComponent );
if( @c ) then

  << do something if somePath has a path component>>

endif;
```

HLA low-level calling sequence example:

```
  push( somePath );
  call filesys.hasPath;
  jnc noPath;

     << Do something if somePath has a path component>>

  noPath:
```

procedure filesys.a_extractBase(pathname:string); @returns("c");

This function extracts and returns the base component of a filename. This function allocates storage for the returned basename on the heap and returns a pointer to that string in the EAX register. If a basename exists, this function returns true in the carry flag (set). If no basename exists (e.g., when the filename ends with '/' so it contains no filename component or if the pathname argument is the empty string), then this function returns a pointer to an empty string allocated on the heap in EAX and it returns with the carry flag clear. It is the caller's responsibility to free the storage associated with the string when the caller is done using that string.

Note that this function's "returns" value is "@c", not EAX. This allows you to use this function in an HLA boolean expression (e.g., in an "if" statement) to test whether the basename is actually valid (that is, it's not an empty string). Don't forget that you still have to free the storage associated with the string, even if it is the empty string.

HLA high-level calling sequence examples:

```
filesys.a_extractBase( somePath );
```

```
mov( eax, basePtr );
if( @c ) then

  <<do something with the basename pointed at by basePtr>>

endif;
str.free( basePtr );


HLA low-level calling sequence example:

  push( somePath );
  call filesys.a_extractBase;
  mov( eax, basePtr );
  jnc noPath;

     << Do something if somePath has a path component>>

  noPath:
  str.free( basePtr );
```

procedure filesys.extractBase(pathname:string; base:string);
@returns("c");

This function extracts and returns the base component of a filename. It extracts the base filename from the *pathname* argument and stores the result into the string storage pointed at by the *base* argument. The *base* argument must point at allocated storage sufficient to hold the base name string or this function will raise an exception. This function returns with the carry flag set if it finds (and copies) a basename component of the pathname. It returns with the carry flag clear if there is no base name component (which implies that pathname ends wth a '/' character or is the empty string).

```
HLA high-level calling sequence examples:

filesys.extractBase( somePath, baseName );
if( @c ) then

  <<do something with the basename held in baseName>>

endif;


HLA low-level calling sequence example:

  push( somePath );
  push( baseName );
  call filesys.extractBase;
  jnc noPath;

     << Do something if somePath has a path component>>

  noPath:
```

procedure filesys.a_extractExt(pathname:string); @returns("c");

This function extracts and returns the extension component of a filename. This function allocates storage for the returned extension on the heap and returns a pointer to that string in the EAX register. If an extension exists, this function returns true in the carry flag (set). If no extension exists (e.g., when the filename component contains no periods), then this function returns a pointer to an empty string allocated on the heap in EAX and it returns with the carry flag clear. It is the caller's responsibility to free the storage associated with the string when the caller is done using that string.

Note that this function's "returns" value is "@c", not EAX. This allows you to use this function in an HLA boolean expression (e.g., in an "if" statement) to test whether the extension is actually valid (that is, it's not an empty string). Don't forget that you still have to free the storage associated with the string, even if it is the empty string.

```
HLA high-level calling sequence examples:

filesys.a_extractExt( somePath );
mov( eax, extPtr );
if( @c ) then

  <<do something with the extension pointed at by extPtr>>

endif;
str.free( extPtr );


HLA low-level calling sequence example:

  push( somePath );
  call filesys.a_extractExt;
  mov( eax, extPtr );
  jnc noPath;

     << Do something if somePath has an extension component>>

  noPath:
  str.free( extPtr );
```

procedure filesys.extractExt(pathname:string; ext:string); @returns("c");

This function extracts and returns the extension component of a filename. It extracts the extension from the *pathname* argument and stores the result into the string storage pointed at by the *ext* argument. The *ext* argument must point at allocated storage sufficient to hold the extension string or this function will raise an exception. This function returns with the carry flag set if it finds (and copies) an extension component of the pathname. It returns with the carry flag clear if there is no extension component (which implies that filename component contains no periods or is the empty string).

```
HLA high-level calling sequence examples:

filesys.extractExt( somePath, extName );
if( @c ) then

  <<do something with the extension held in extName>>

endif;


HLA low-level calling sequence example:

  push( somePath );
```

```
    push( extName );
    call filesys.extractExt;
    jnc noPath;

        << Do something if somePath has an extension component>>

    noPath:
```

procedure filesys.a_extractFilename(pathname:string); @returns("c");

This function extracts and returns the filename component of a pathname. This function allocates storage for the returned filename on the heap and returns a pointer to that string in the EAX register. If a filename component exists, this function returns true in the carry flag (set). If no filename exists (e.g., when the pathname ends with a '/' or is the empty string), then this function returns a pointer to an empty string allocated on the heap in EAX and it returns with the carry flag clear. It is the caller's responsibility to free the storage associated with the string when the caller is done using that string.

Note that this function's "returns" value is "@c", not EAX. This allows you to use this function in an HLA boolean expression (e.g., in an "if" statement) to test whether the filename is actually valid (that is, it's not an empty string). Don't forget that you still have to free the storage associated with the string, even if it is the empty string.

```
HLA high-level calling sequence examples:

filesys.a_extractFilename( somePath );
mov( eax, fnPtr );
if( @c ) then

  <<do something with the filename pointed at by fnPtr>>

endif;
str.free( fnPtr );


HLA low-level calling sequence example:

  push( somePath );
  call filesys.a_extractFilename;
  mov( eax, fnPtr );
  jnc noPath;

      << Do something if somePath has a filename component>>

  noPath:
  str.free( fnPtr );
```

procedure filesys.extractFilename(pathname:string; filename:string); @returns("c");

This function extracts and returns the filename component of a pathname. It extracts the filename from the *pathname* argument and stores the result into the string storage pointed at by the *filename* argument. The *filename* argument must point at allocated storage sufficient to hold the filename string or this function will raise an exception. This function returns with the carry flag set if it finds (and copies) a filename component of the pathname. It returns with the carry flag clear if there is no filename component (which implies that pathname component ends with a '/' or is the empty string).

```
HLA high-level calling sequence examples:
```

```
filesys.extractFilename( somePath, filename );
if( @c ) then

  <<do something with the string held in filename>>

endif;


HLA low-level calling sequence example:

  push( somePath );
  push( filename );
  call filesys.extractFilename;
  jnc noPath;

     << Do something if somePath has a filename component>>

  noPath:
```

procedure filesys.a_extractPath(pathname:string); @returns("c");

This function extracts and returns the path component of a pathname string. This function allocates storage for the returned path on the heap and returns a pointer to that string in the EAX register. If a path component exists, this function returns true in the carry flag (set). If no path component exists (e.g., when the pathname argument contains no '/' characters or is the empty string), then this function returns a pointer to an empty string allocated on the heap in EAX and it returns with the carry flag clear. It is the caller's responsibility to free the storage associated with the string when the caller is done using that string.

Note that this function's "returns" value is "@c", not EAX. This allows you to use this function in an HLA boolean expression (e.g., in an "if" statement) to test whether the path is actually valid (that is, it's not an empty string). Don't forget that you still have to free the storage associated with the string, even if it is the empty string.

```
HLA high-level calling sequence examples:

filesys.a_extractPath( somePath );
mov( eax, pathPtr );
if( @c ) then

  <<do something with the path pointed at by pathPtr>>

endif;
str.free( pathPtr );


HLA low-level calling sequence example:

  push( somePath );
  call filesys.a_extractPath;
  mov( eax, pathPtr );
  jnc noPath;

     << Do something if somePath has a path component>>

  noPath:
  str.free( pathPtr );
```

```
procedure filesys.extractPath( pathname:string; path:string );
    @returns( "c" );
```

This function extracts and returns the path component of a pathname string. It extracts the path from the *pathname* argument and stores the result into the string storage pointed at by the *path* argument. The *path* argument must point at allocated storage sufficient to hold the string or this function will raise an exception. This function returns with the carry flag set if it finds (and copies) a path component of the pathname. It returns with the carry flag clear if there is no path component (which implies that pathname component contains no '/' characters or is the empty string).

Note that the path string that this function returns will include any UNC prefixes and any Win32 drive letters that are present in the original pathname. If you need a path result that doesn't contain any drive letter prefixes, call *filesys.hasDriveLetter* on the result and delete the first two character of the string if there is a drive letter present. If you need a string without a UNC prefix present, then call *filesys.hasUncName* on the path result and delete the first EAX characters from the path string if *filesys.hasUncName* reports that a UNC name is present.

```
HLA high-level calling sequence examples:

filesys.extractPath( somePath, pathComponent );
if( @c ) then

  <<do something with the string held in pathComponent >>

endif;


HLA low-level calling sequence example:

  push( somePath );
  push( pathComponent );
  call filesys.extractPath;
  jnc noPath;

     << Do something if somePath has a path component>>

  noPath:
```

```
procedure filesys.a_joinPaths( leftPath:string; rightPath:string );
    @returns( "eax" );
```

This function concatenates two path strings, adding a directory separator character between them (if necessary). Because of the wide variety of special cases that can occur when concatenating two paths, this function is fairly complex. The operation is described in the following table.

If leftPath…	If rightPath…	Then the resulting path string…
Is empty	Is empty	Is empty.
Ends with '/'	Does not begin with '/'	Is just the concatenation of the two strings.

Does not end with '/'	Begins with '/'	Is just the concatenation of the two strings.
Ends with '/'	Begins with '/'	Is the concatenation of the two strings with one of the '/' characters removed.
Does not end with '/'	Does not begin with '/'	Is the concatenation of the leftPath with '/' and then the rightPath string.

Note that, unlike many of the other *filesys* file/path string functions, this function does not return a failure/success status in the carry flag. This function always succeeds (or it raises an exception if an exceptional condition exists).

```
HLA high-level calling sequence examples:

filesys.a_joinPaths( leftPath, RightPath );
mov( eax, pathPtr );

  <<do something with the path pointed at by pathPtr>>

str.free( pathPtr );


HLA low-level calling sequence example:

  push( leftPath );
  push( rightPath
  call filesys.a_joinPaths;
  mov( eax, pathPtr );

     << Do something with the string pointed at by pathPtr>>

  str.free( pathPtr );
```

```
procedure filesys.joinPaths
(
     leftPath:string;
     rightPath:string;
     joinedPath:string
);
```

This function combines the *leftPath* and *rightPath* strings to form a *joinedPath* string. For the exact details on the concatenation operation, please see the table appearing in the description of the *filesys.a_joinPaths* function. This function will raise an exception if the string storage pointed at by *joinedPaths* is not large enough to hold the result. Note that the calculation for string overflow is computed as the sum of the lengths of leftPath and rightPath plus one, even though (in some cases) the required length might need only be the sum of the lengths of the *leftPath* and *rightPath* strings. Therefore, you should ensure that the storage allocated for the *joinedPath* string is at least one character larger than the sum of the two substrings or this function may raise an exception, even if *joinedPath* turns out to be large enough to hold the actual result.

```
HLA high-level calling sequence examples:

filesys.joinPaths( leftPath, rightPath, newPath );

  <<do something with the string held in newPath >>


HLA low-level calling sequence example:

  push( leftPath );
  push( rightPath );
  push( newPath
  call filesys.joinPaths;

     << Do something with newPath>>
```

procedure filesys.a_normalize(pathname:string); @returns("c");

This function normalizes the pathname passed as the argument and returns a pointer to the normalized pathname in EAX. This function allocates storage for the normalized pathname on the heap; it is the caller's responsibility to free that storage when the application is done using the string.

A normalized pathname is one that has all the directory separators converted to the native format, has all path components of the form "./" deleted from the path string, and has all path components of the form "path/../" deleted from the path string. Note, however, that if "../" appears at the beginning of a path string, or "../" appears immediately after a UNC path sequence, then the normalized result still contains the "../". Likewise, if there are multiple "../" sequences within a path string and deleting the previous paths would delete a UNC component or would attempt to delete a path sequence before any appearing in the path string, then this function leaves the "../" component present (e.g., "path/../../name" produces the string "../name").

This function returns the carry flag set if it was able to produce a correct normalized string. This function returns with the carry flag clear if the *pathname* argument contained an unparseable UNC name prefix or other syntax error (in which case the function ignores the UNC prefix and treats the UNC like any other path sequence).

```
HLA high-level calling sequence examples:

filesys.a_normalize( somePath );
mov( eax, pathPtr );
if( @c ) then

  <<do something with the path pointed at by pathPtr>>

endif;
str.free( pathPtr );


HLA low-level calling sequence example:

  push( somePath );
  call filesys.a_extractPath;
  mov( eax, pathPtr );
  jnc noPath;

     << Do something if somePath has a path component>>

  noPath:
  str.free( pathPtr );
```

procedure filesys.normalize1(pathname:string);
**   @returns("c");**

This function nornalizes, in place, the pathname string passed as an argument. On returns, the pathname string contains the normalized result of the original string passed into this function. Note that normalized strings are always the same length or shorter than the original string, so there is no chance of string overflow occuring when normalizing a path string. For details on the normalization process, see the description of the *filesys.a_normalize* function.

This function returns with the carry flag set if it successfully normalizes the *pathname* argument and there are no UNC parse errors or other problems. If this function cannot parse a string that looks like it has a UNC prefix, it will treat *pathname* as though it has no UNC prefix, normalize that, and return with the carry flag clear.

Because this function normalizes the string in place, the *pathname* argument must point at string data in writeable memory or else this function will raise an exception.

```
HLA high-level calling sequence examples:

filesys.normalize1( somePath );
if( @c ) then

  <<do something with the string held in somePath >>

endif;


HLA low-level calling sequence example:

  push( somePath );
  call filesys.normalize1;
  jnc noPath;

     << Do something with somePath>>

  noPath:
```

procedure filesys.normalize2(pathname:string; path:string);
**   @returns("c");**

This function nornalizes the *pathname* string passed as an argument and stores the normalized result into the string object pointed at by *path*. The *path* object must have at least as much space allocated for it as the length of the *pathname* argument or this function will raise an exception; this exception will be raised even if the actual normalized string would be shorter than the length of *pathname* and *path* is actually large enough to hold the actual normalized string. The test for string overflow takes place before the normalization operation begins because this function first copies *pathname* to *path* and then performs the normalization operation on *path*. For details on the normalization process, see the description of the *filesys.a_normalize* function.

This function returns with the carry flag set if it successfully normalizes the *pathname* argument and there are no UNC parse errors or other problems. If this function cannot parse a string that looks like it has a UNC prefix, it will treat *pathname* as though it has no UNC prefix, normalize that, and return with the carry flag clear.

```
HLA high-level calling sequence examples:

filesys.extractPath( somePath, pathComponent );
if( @c ) then

  <<do something with the string held in pathComponent >>
```

```
endif;


HLA low-level calling sequence example:

  push( somePath );
  push( pathComponent );
  call filesys.extractPath;
  jnc noPath;

     << Do something if somePath has a path component>>

  noPath:
```

procedure filesys.a_toUnixPath(pathname:string); @returns("eax");

This function converts a string to the Unix format. This entails converting all '\' characters to '/' characters in the pathname string. Note that this function does not process any drive letter prefixes. If a drive letter prefix is present in the pathname argument, this function returns that drive letter prefix in the result string.

This function returns a pointer to tne converted string, which is allocated on the heap, in the EAX register. It is the caller's responsibility to free the storage associated with this string when the caller is done using the string data.

```
HLA high-level calling sequence examples:

filesys.a_toUnixPath( somePath );
mov( eax, unixPathPtr );

  <<do something with the path pointed at by unixPathPtr >>

str.free( unixPathPtr );


HLA low-level calling sequence example:

  push( somePath );
  call filesys.a_toUnixPath;
  mov( eax, unixPathPtr );

     << Do something with unixPathPtr>>

  str.free( unixPathPtr );
```

procedure filesys.toUnixPath1(pathname:string);

This function converts the string argument to UNIX format by replacing all '\' characters with '/' ' characters. Note that this function does not process any drive letter prefixes. If a drive letter prefix is present in the pathname argument, this function returns that drive letter prefix in the result string. Converted strings are always the same length as the original string, so there is no chance of string overflow occuring when converting a path string to UNIX format.

Because this function normalizes the string in place, the *pathname* argument must point at string data in writeable memory or else this function will raise an exception.

```
HLA high-level calling sequence examples:
```

```
filesys.toUnixPath1( somePath );

  <<do something with the string held in somePath >>
```

HLA low-level calling sequence example:

```
  push( somePath );
  call filesys.toUnixPath1;

      << Do something with somePath>>
```

procedure filesys.toUnixPath2(pathname:string; unixPath:string);

This function converts the *pathname* string argument to UNIX format by replacing all '\' characters with '/' characters. It stores the resulting string into *unixPath*. Note that this function does not process any drive letter prefixes. If a drive letter prefix is present in the pathname argument, this function returns that drive letter prefix in the result string. Converted strings are always the same length as the original string, so the storage pointed at by the *unixPath* argument must be able to hold at least as many characters as the current length of the *pathname* argument or this function will raise an exception.

HLA high-level calling sequence examples:

```
filesys.toUnixPath2( somePath, unixPath );

  <<do something with the string held in unixPath>>
```

HLA low-level calling sequence example:

```
  push( somePath );
  push( unixPath );
  call filesys.toUnixPath2;

      << Do something with unixPath>>
```

procedure filesys.a_toWin32Path(pathname:string); @returns("eax");

This function converts a pathname string to the Windows format. This entails converting all '/' characters to '\' characters in the *pathname* string.

This function returns a pointer to the converted string, which is allocated on the heap, in the EAX register. It is the caller's responsibility to free the storage associated with this string when the caller is done using the string data.

HLA high-level calling sequence examples:

```
filesys.a_toWin32Path( somePath );
mov( eax, win32PathPtr );

  <<do something with the path pointed at by win32PathPtr >>
```

```
str.free( win32PathPtr );


HLA low-level calling sequence example:

  push( somePath );
  call filesys.a_toWin32Path;
  mov( eax, win32PathPtr );

      << Do something with win32PathPtr >>

  str.free( win32PathPtr );
```

procedure filesys.toWin32Path1(pathname:string);

This function converts the pathname string argument to Windows format by replacing all '/' characters with '\' ' characters. Converted strings are always the same length as the original string, so there is no chance of string overflow occuring when converting a path string to UNIX format.

Because this function normalizes the string in place, the *pathname* argument must point at string data in writeable memory or else this function will raise an exception.

```
HLA high-level calling sequence examples:

filesys.toWin32Path1( somePath );

  <<do something with the string held in somePath >>


HLA low-level calling sequence example:

  push( somePath );
  call filesys.toWin32Path1;

      << Do something with somePath>>
```

procedure filesys.toWin32Path2(pathname:string; windowsPath:string);

This function converts the *pathname* string argument to Windows format by replacing all '/' characters with '\' ' characters. It stores the resulting string into *windowsPath*. Converted strings are always the same length as the original string, so the storage pointed at by the *windowsPath* argument must be able to hold at least as many characters as the current length of the *pathname* argument or this function will raise an exception.

```
HLA high-level calling sequence examples:

filesys.toWin32Path2( somePath, windowsPath );

  <<do something with the string held in windowsPath >>


HLA low-level calling sequence example:
```

```
    push( somePath );
    push( windowsPath );
    call filesys.toWin32Path2;

        << Do something with windowsPath >>
```

procedure filesys.a_toNativePath(pathname:string);
procedure filesys.toNativePath1(pathname:string);
procedure filesys.toNativePath2(pathname:string; windowsPath:string);

These functions are synonyms for either the *toUnix* functions or the *toWin32* functions, depending upon the operating system under which you're compiling them. That is, under Windows, these functions are synonyms for the *filesys.a_toWin32Path, filesys.toWin32Path1*, and *filesys.toWin32Path2* functions. Under other Oses, these functions are synonyms for the *filesys.a_toUnixPath, filesys.toUnixPath1*, and *filesys.toUnixPath2* functions.

Functionally, this procedures convert the *pathname* passed as an argument to the native OS pathname format. Note that these functions ignore drive letters if they are converting pathnames to UNIX format (that is, the drive letters will still be present in the converted string).

procedure filesys.a_getFullPathName(partialPath:string); @returns("eax");

This function takes the *partialPath* passed as a parameter and converts it to a full, absolute, pathname. If *partialPath* begins with a '/' character, this function simply returns *partialPath's* value as its result. If *partialPath* does not begin with a '/' character, then this function determines the working directory path and concatenates (via *filesys.joinPaths*) *partialPath* to the end of the current working directory and returns that full path string.

This function returns a pointer to the full pathname string, which is allocated on the heap, in the EAX register. It is the caller's responsibility to free the storage associated with this string when the caller is done using the string data.

```
HLA high-level calling sequence examples:

filesys.a_getFullPath( somePath );
mov( eax, fullPathPtr );

  <<do something with the path pointed at by fullPathPtr>>

str.free( fullPathPtr );


HLA low-level calling sequence example:

  push( somePath );
  call filesys.a_getFullPath;
  mov( eax, fullPathPtr );

      << Do something with fullPathPtr >>

  str.free( fullPathPtr );
```

procedure filesys.getFullPath(partialPath:string; resultPath:string);

This function takes the *partialPath* passed as a parameter and converts it to a full, absolute, pathname. If *partialPath* begins with a '/' character, this function simply returns *partialPath's* value as its result. If *partialPath* does not begin with a '/' character, then this function determines the working directory path and concatenates (via *filesys.joinPaths*) *partialPath* to the end of the current working directory and returns that full path string. In any case, the resulting fully qualified pathname is stored into the *resultPath* string (raising an exception if *resultPath's* allocation is insufficient to hold the string).

```
HLA high-level calling sequence examples:

filesys.getFullPath( somePath, fullPath );

  <<do something with the string held in fullPath>>



HLA low-level calling sequence example:

  push( somePath );
  push( fullPath );
  call filesys.getFullPath;

     << Do something with fullPath>>
```

16.2 Directory and File Predicates

These functions test some condition about a file or directory and return true or false in the EAX register.

`procedure filesys.exists( pathname:string ); @returns( "eax" );`

This function returns true if the file exists and the application can open the file (at least for reading). It returns false in EAX if the file does not exist, is inaccessible, or there is some other error that occurs when attemting to open the file.

```
HLA high-level calling sequence examples:

filesys.exists( "someFilename" );
mov( al, someFilenameExists );// someFileNameExists:boolean;


HLA low-level calling sequence examples:

someFilename:string := "someFileName";
  .
  .
  .
push( someFileName );
call filesys.exists;
mov( al, someFilenameExists );
```

`procedure filesys.isFile( FileName:string ); @returns( "eax" );`

This function returns true if the file exists and is actually a file (rather than a directory) and the application can open the file (at least for reading). It returns false in EAX if the file does not exist, exists but is a directory, is inaccessible, or there is some other error that occurs when attemting to open the file.

```
HLA high-level calling sequence examples:

filesys.isFile( "someFilename" );
mov( al, someFilenameExists );// someFileNameExists:boolean;


HLA low-level calling sequence examples:

someFilename:string := "someFileName";
  .
  .
```

```
  .
push( someFileName );
call filesys.isFile;
mov( al, someFilenameExists );
```

procedure filesys.isDir(FileName:string); @returns("eax");

This function returns true if the file exists and is a directory (rather than a regular file). It returns false in EAX if the file does not exist, exists but is not a directory, is inaccessible, or there is some other error that occurs when attemting to open the file.

```
HLA high-level calling sequence examples:

filesys.isDir( "someDirName" );
mov( al, someDirNameExists );// someDirNameExists:boolean;


HLA low-level calling sequence examples:

someDirName:string := "someDirName";
  .
  .
  .
push( someDirName );
call filesys.isDir;
mov( al, someDirNameExists );
```

16.3 File Information Functions

These functions return some information about the file.

procedure filesys.size(Handle:dword); @returns("edx:eax");
procedure filesys.size(filename:string); @returns("edx:eax");

These (overloaded) functions return the current size of a file. They return the size in the EDX:EAX register pair. There are two versions of this function – one that accepts a string parameter and one that accepts a dword parameter. The first (dword) form expects you to pass it an open file handle; the second (string) form expects you to pass it a string containing the filename of the file whose size you want to compute.

Note: HLA uses macros to implement overloading. If you really must make a low-level call to one of these functions (or if you need to take the address of one of these functions), then you will need to refer to the actual procedure names: *filesys._sizeh_* is the name of the function expecting a file handle, *filesys._sizen_* is the name of the function expecting a filename string parameter.

```
HLA high-level calling sequence examples:

filesys.size( "someFileName" );
mov( eax, sizeOfSomeFileName );
filesys._sizen_( "someOtherFileName" );
mov( eax, sizeOfSomeOtherFile );

fileio.open( "anotherFile", fileio.r );
mov( eax, fileHandle );
filesys.size( fileHandle );
mov( eax, sizeOfAnotherFile );

fileio.open( "anotherFile2", fileio.r );
mov( eax, fileHandle2 );
filesys._sizeh_( fileHandle2 );
mov( eax, sizeOfAnotherFile2 );
```

```
HLA low-level calling sequence examples:

someFileName:string := "someFileName";
anotherFile :string := "anotherFile";
  .
  .
  .
push( someFileName );
callfilesys._sizen_;
mov( eax, sizeOfSomeOtherFile );
  .
  .
  .
push( anotherFile );
pushd( fileio.r );
call fileio.open;
mov( eax, fileHandle );
push( eax );
call filesys._sizeh_;
mov( eax, sizeOfAnotherFile2 );
```

16.4 Directory and File Manipulation Functions

`procedure filesys.delete( filename:string ); @returns( "eax" );`

This function deletes the specified file. Obviously, use this function with care. If this function fails, it raises the *ex.CannotRemoveFile* exception. Note that this function will only delete regular files. It will fail, and raise an exception if you attempt to delete a directory.

```
HLA high-level calling sequence examples:

filesys.delete( "someFileName" );


HLA low-level calling sequence examples:

someFileName:string := "someFileName";
  .
  .
  .
push( someFileName);
call filesys.delete;
```

`procedure filesys.mkdir( dirname:string ); @returns( "eax" );`

This function creates a directory using the pathname you supply as a parameter. It returns the error status in EAX. If this function fails, it raises the *ex.CannotCreateDir* exception.

```
HLA high-level calling sequence examples:

filesys.mkdir( "newDirName" );


HLA low-level calling sequence examples:

newDirName:string := "newDirName";
  .
```

```
    .
    .
  push( newDirName);
  call filesys.mkdir;
```

procedure filesys.cd(dirname:string);

This function sets the current working directory to the filename you pass as a parameter. If this function fails, it raises the *ex.CDFailed* exception.

```
  HLA high-level calling sequence examples:

  filesys.cd( "newWorkingDirectory" );


  HLA low-level calling sequence examples:

  newWorkingDirectory:string := "newWorkingDirectory";
    .
    .
    .
  push( newWorkingDirectory);
  call filesys.cd;
```

procedure filesys.gwd(dest:string);

This function returns a string containing the current working directory's pathname in the string you pass as a parameter. The string must have storage allocated for it and it must be large enough to hold the pathname or HLA will raise a string overflow exception.

```
  HLA high-level calling sequence examples:

  filesys.gwd( allocatedStringVar );


  HLA low-level calling sequence examples:

  push( allocatedStringVar);
  call filesys.gwd;
```

procedure filesys.rename(fromPath:string; toPath:string);

This function renames one file to another. The *fromPath* parameter specifies the file to rename, the *toPath* parameter specifies the new name for the file. If the rename operation is unsuccessful, this function raises the *ex.CannotRenameFile* exception.

```
  HLA high-level calling sequence examples:

  filesys.rename( "oldName", "newName" );


  HLA low-level calling sequence examples:

  oldName:string := "oldName";
  newName:string := "newName";
    .
    .
```

```
    .
    .
    .
    push( oldName );
    push( newName );
    call filesys.rename;
```

procedure filesys.rmdir(directory:string);

This function deletes the specified directory. The directory must be empty before you attempt to delete it or this function will raise an exception. If this function fails, it raises the *ex.CannotRemoveDir* exception. Note that this function will only delete directories. It will fail, and raise an exception if you attempt to delete a regular file.

```
HLA high-level calling sequence examples:

filesys.rmdir( "dirToRemove" );


HLA low-level calling sequence examples:

dirToRemove:string := "dirToRemove";
    .
    .
    .
push( dirToRemove);
call filesys.rmdir;
```

iterator filesys.fileWithSuffix(directory:string; suffix:string);

This iterator, used within a foreach loop, will repeat once for each file in the directory specified by the *directory* parameter that ends with the string specified by *suffix*. On each iteration of the foreach loop, this function will provide a pointer to the full filename in the EAX register. Note that this function only iterates on regular files, it will not iterate (nor return the string name) of any directory entries. The *filesys.fileWithSuffix* function will allocate storage for the filename string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

```
HLA high-level calling sequence examples:

foreach filesys.fileWithSuffix( "dirToSearch", ".hla" ) do

  // At this point, EAX points at a filename ending with ".hla"

      mov( eax, filename );// filename:string

      // Do something with that string…

      // When we're done with the string, free it.

      str.free( filename );

  endfor;
```

iterator filesys.fileIn(directory:string);

This iterator, used within a foreach loop, will repeat once for each file in the directory specified by the *directory* parameter. On each iteration of the foreach loop, this function will provide a pointer to the full filename in the EAX register. Note that this function only iterates on regular files, it will not iterate (nor return the string name) of any directory entries. The *filesys.fileIn* function will allocate storage for the filename string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

```
HLA high-level calling sequence examples:
```

```
foreach filesys.fileIn( "dirToSearch" ) do

     // At this point, EAX points at a filename of a file
     // found in the "dirToSearch" directory.

     mov( eax, filename );// filename:string

     // Do something with the string…

     // When we're done with the string, free it.

     str.free( filename );

  endfor;
```

iterator filesys.fileInCwd;

This iterator, used within a foreach loop, will repeat once for each file in the current directory. On each iteration of the foreach loop, this function will provide a pointer to the full filename in the EAX register. Note that this function only iterates on regular files, it will not iterate (nor return the string name) of any directory entries. The *filesys.fileInCwd* function will allocate storage for the filename string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

```
HLA high-level calling sequence examples:

foreach filesys.fileInCwd() do

     // At this point, EAX points at a filename of a file
     // found in the current working directory.

     mov( eax, filename );// filename:string

     // Do something with the string…

     // When we're done with the string, free it.

     str.free( filename );

  endfor;
```

iterato

This iterator, used within a foreach loop, will repeat once for each directory entry, whose name ends with *suffix*, in the directory specified by the *directory* parameter. On each iteration of the foreach loop, this function will provide a pointer to the full directory name in the EAX register. Note that this function only iterates on directory files, it will not iterate (nor return the string name) of any regular file entries. The *filesys.dirWithSuffix* function will allocate storage for the directory name string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

```
HLA high-level calling sequence examples:

foreach filesys.dirWithSuffix( "dirToSearch", "suffix" ) do

     // At this point, EAX points at name of a directory
     // that ends with "suffix" that was found in the "dirToSearch"
     // directory.

     mov( eax, dirname );// dirname :string

     // Do something with the string…
```

```
        // When we're done with the string, free it.

        str.free( dirname );

    endfor;
```

iterator filesys.dirIn(directory:string);

This iterator, used within a foreach loop, will repeat once for each directory entry in the directory specified by the *directory* parameter. On each iteration of the foreach loop, this function will provide a pointer to the full directory name in the EAX register. Note that this function only iterates on directory files, it will not iterate (nor return the string name) of any regular file entries. The *filesys.dirIn* function will allocate storage for the directory string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

```
HLA high-level calling sequence examples:

foreach filesys.dirIn( "dirToSearch" ) do

        // At this point, EAX points at name of a directory
        // that was found in the "dirToSearch" directory.

        mov( eax, dirname );// dirname :string

        // Do something with the string…

        // When we're done with the string, free it.

        str.free( dirname );

    endfor;
```

iterator filesys.dirInCwd;

This iterator, used within a foreach loop, will repeat once for each directory entry in the current directory. On each iteration of the foreach loop, this function will provide a pointer to the full directory name in the EAX register. Note that this function only iterates on directory files, it will not iterate (nor return the string name) of any regular file entries. The *filesys.dirInCwd* function will allocate storage for the directory string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

```
HLA high-level calling sequence examples:

foreach filesys.dirInCwd() do

        // At this point, EAX points at name of a directory
        // that was found in the current working directory.

        mov( eax, dirname );// dirname :string

        // Do something with the string…

        // When we're done with the string, free it.

        str.free( dirname );

    endfor;
```

iterator filesys.itemWithSuffix(directory:string; suffix:string);

This iterator, used within a foreach loop, will repeat once for each entry, whose name ends with *suffix*, in the directory specified by the *directory* parameter. On each iteration of the foreach loop, this function will provide a pointer to the name in the EAX register. Note that this function iterates on both directory and regular file entries. The *filesys.itemWithSuffix* function will allocate storage for the result string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

```
HLA high-level calling sequence examples:

foreach filesys.itemWithSuffix( "dirToSearch", "suffix" ) do

  // At this point, EAX points at a filename or a directory
  // name ending with "suffix"

      mov( eax, entryname );// entryname :string

      // Do something with that string…

      // When we're done with the string, free it.

      str.free( entryname );

  endfor;
```

iterator filesys.itemInCwd;

This iterator, used within a foreach loop, will repeat once for each entry in the current directory. On each iteration of the foreach loop, this function will provide a pointer to the name in the EAX register. Note that this function iterates on both directory and regular file entries. The *filesys.itemInCwd* function will allocate storage for the result string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

```
HLA high-level calling sequence examples:

foreach filesys.itemInCwd( ) do

  // At this point, EAX points at a filename or a directory
  // name from the current working directory.

      mov( eax, entryname );// entryname :string

      // Do something with that string…

      // When we're done with the string, free it.

      str.free( entryname );

  endfor;
```