The SS Programming Language Appendix 2

Version: June 12, 2019

Table of Contents

Copyright

Directories

Handling directories (aka folders) is done through an object type **_Directory**. This object type defines some functions and methods to create and manipulate directories. The standard library member **std/directory.ss** defines the object type and procedures.

To work with directories, a program must first create the object using the constructor, which has one optional argument **path** with a default value of **'.'**, the current directory.

Table: Built-in Directory Functions

Function:	Result Type:	Description:
changeDir(string path)	_Directory	Change to directory or path (a mode)
clearDir(string path)	bool	Deletes the directory and all of its contents
createDir(string path)	bool	Creates a new directory or subdirectory
currentDir	string	Property, returns the current directory name
dirs	string array	Returns the directories in current directory
files	string array	Returns the files in the current directory
files(string path)	string array	Returns the files in the named directory path
findDir(string path)	bool	Tests if a path is a directory
getPath(string path)	string	Returns full path name of a directory path
renameDir(string name)	bool	Rename/move the current directory

All of the above are based on an object of type _Directory.

Failure when the result is not type bool is a return of **null**.

The meaning of *path* is system-dependent. In the above, *path* can be a single directory or a full path name or a relative path. For consistency, a forward slash is always used.

The function **dirs** returns the subdirectories, without the pseudo-directories ".." or ".".

These names can be obtained:

```
_Directory thisDir .= currentDir
string curDirs[] = thisDir.dirs
```

To change to the parent directory, do this:

```
thisDir.changeDir('..')
```

To see files in the current directory as an array of file names, no directories included:

```
string myFiles[] = thisDir.files
```

It is possible to work with files or directories which are not in the current directory.

Additional functions for security settings are provided.

The File System

SS treats files as objects of a type named _File rather than using file handles or file pointers as in older languages based on C. The definitions are in standard library member **std/files.ss**.

The attributes of a file are defined in object _FileAttribs, also in the same library member. This object is inherited by a _File.

From a directory, we can get file names in that directory. A simple name is in the current directory, and a path name refers to any file.

Table: Built-in File System Functions and Methods

Function:	Base:	Result Type:	Description:
attribs	_File	_FileAttribs	Get the attributes of a file
delete	_File	(none)	Removes a file
extension	_File	string	Returns the extension
file	string	_File	_File object for a path or name
isFile	string	bool	Tests if a path is an existing file
isLegitName	string	bool	Tests if a path or file is a legal name
isWriteOK	string	bool	Tests if a file is writable
name	_File	string	The name part, with extension
path	_File	string	The full path for the file
rename(string name)	_File	_File or **null**	Change the file name
move(string path)	_File	_File or **null**	Move the file and change its name

These _File objects are not prepared ("opened") for input/output. They simply define the files known to the program.

File Operations

File operations means manipulation of the contents of files. In order to read or write, a file must be opened. Unlike using file handles or file pointers as in older languages based on C, file operations use an object.

SS file operations use a hierarchy of object types in opening a file. These ensure that reading a file can only be done on a file open for reading, etc. There are objects for each type of I/O.

The definition of the file operation functions and modes and object types is in standard library member **std/io.ss**.

The hierarchy of object types and what they can do is shown in outline form:

Table: File Operations Object Type Hierarchy

Object Type:	Creating Modes:	Definition:
_File	*string*.file	object identifying the name/path *string*
_TextFile	*_File*.text	object operating on text lines in a file
_ReadTextFile	*_TextFile*.input	object reading text lines, file exists
_WriteTextFile	*_TextFile*.output	object writing text lines, replace a file
_WriteTextFile	*_TextFile*.append	object writing text lines, end of existing file
_BinaryFile	*_File*.binary(object *blk*)	object operating on a binary file of fixed sized blocks, the size of an object *blk*
_ReadBinFile	*_BinaryFile*.input	object reading binary blocks, existing file
_WriteBinFile	*_BinaryFile*.output	object writing binary blocks, replace/new file
_UpdateBinFile	*_BinaryFile*.update	object reading or writing binary blocks

This object type file system limits operations to appropriate functions and values, capable of being checked at compilation/syntax checking time, reducing errors.

Each subsidiary type inherits the prior types, and some of their functions remain available.

From the above hierarchy, we see that a given object type may be only an intermediate step, and we see that the modes input, output, append and update set a working_mode and return a type, as appropriate. Once open, the working_mode cannot be changed.

A binary file is a series of fixed-size blocks which may be buffered or combined. It can also be opened in update mode, which allows randomly positioned reads and writes.

The above uses standard error handling - error condition codes are TBD.

Open Files

An *open_file* is an object of one of these types:

_ReadTextFile, **_WriteTextFile**, **_ReadBinFile**, **_WriteBinFile**, or **_UpdateBinFile**.

An *open_file* has a type which enables the interface **_OpenFile**.

Table: Interface _OpenFile Procedures

Name:	Action:
close	Closes the file

Table: Operations on an Open File

Object Type:	Procedure:	Description:
_ReadTextFile	read First	returns a string (a line) or **null** mode, sets position to first line
_WriteTextFile	write(string *str*) writeLine(string *line*) writeLine(string lines[])	function, writes str, returns bool function, writes line with \n added, returns bool function, writes all lines with \n added, returns bool
_ReadBinFile	First read(ulong *blknum*)	mode, sets position to first byte (number 0) returns a string, one block, at data block number *blknum*, or **null**
_UpdateBinFile	read(ulong *blknum*) write(ulong *blknum*, object *blk*)	returns a block, at block number *blknum*, or **null** function, writes a block, to block number *blknum*, returns bool
_WriteBinFile	write(struct *blk*)	function, writes a block, to next block number, returns bool

Table: Additional File Function

Function:	Base Type:	Result:	Description:
getLines	_TextFile	string array	opens, then reads a text file and returns the lines as an array of strings with end-of-line characters trimmed off, then closes the file

Example of Text File Reading

```
# Open and read a file named 'data' and stops if the word 'cloud' appears - -
_ReadTextFile inp = 'data'.file.text.input
string word = 'cloud'
{ string line = inp.read
    {if line.isNull  # stops on null return
        @ "Word '{word}' not found."
        break
    else if line.find(word) >= 0
        @ "Word '{word}' found in line {line}."
        break
    }
}
```

Another solution:

```
# this is an alternate solution to the problem - -
{if 'data'.file.getLines.join('*').find('cloud') >= 0
    @ "Word 'cloud' found"
else
    @ "Word 'cloud' not found"
}
```


Formatting Strings

A set of built-in functions, modes, and properties are defined in standard library member **std/fmt.ss**. These are used to format strings for printing/output.

These modes are (mostly) based on object type **_Fmt**. A new text string is started with the function 'Fmt', no base object, creating an empty string, adding text elements by appending. Each of these functions and modes begin with (or are a single character) an upper case letter. Then at the end, a last property is 'P', which yields the string result.

Some of the modes do not add text; they set parameters for a following value. These are the *parameter modes* W, D, Z, C, R, and L. A value mode must follow one or more of these modes. Mode D also works for integers, in effect dividing by 100.0 for 2 decimal points, etc.

Modes R and L establish justification. Right (R) is default for numeric values, left for strings.

After any value item, the modes W, A, P, D, E, Z, C, R, and L are reset. Modes P and Z are mutually exclusive.

Table: Built-in Formatting Functions, Properties and Modes

Procedure:	On Type:	Description:
Fmt	(none)	Mode, establishes a new _Fmt object
P	_Fmt	Property, ("Print" or "Produce") returns the produced string
X(int n)	_Fmt	mode, adds n blanks
W(int width)	_Fmt	mode, establishes width of next item
O	_Fmt	mode, resets all options to default
C	_Fmt	mode, shows thousands commas on N following
M	_Fmt	mode, ("minus") shows negative numbers in parentheses, and adds a blank on the right for plus values. Cancels Z
L	_Fmt	mode, sets left justification (default)
R	_Fmt	mode, sets right justification
Z	_Fmt	mode, sets leading zeros for N(int) item following, cancels M
EOL	_Fmt	mode, adds end-of-line characters
D(int d)	_Fmt	mode, sets **d** decimal places for numeric item following
E	_Fmt	mode, toggles mode for N (float) following to either display exponent style rather than fixed point or resume fixed
S(string str)	_Fmt	mode, inserts the string value str, obeys W, R, L modes
N(n)	_Fmt	mode, inserts the numeric value n, obeys W, R, L, D, C, M, and Z modes. Ignores E mode if not float. Disallows M with Z mode. Float n ignores Z mode and C mode, and E mode implies a request for scientific mode

All these modes return a _Fmt object reference. This object maintains the current settings and the current output string.

The D mode sets a number of decimals shown on a float or integer value.

Scientific mode has an adjusted exponent, to a multiple of 3, and it adjusts the number of digits before the period to 1 or 2 nonzero values as needed. Applies to float only.

If a value cannot be converted, a string of '#' characters is output.

This formatting system is designed to optimize output and also enable compiler error checking.

The value of a stack of modes on Fmt ending with P is the formatted output as a string.

Examples of Formatting

```
int tests[], ic = tests.Count   # assume the tests array has been filled somehow
@ Fmt.S('For ').C.N(ic).S('tests, the average is ').W(12).R.D(3).N(tests.sum/ic).P
int Nused = -3547822
int plusV = 78845
string disp1 = Fmt.W(12).C.M.N(Nused).P     # sets value of disp1 to: " (3,547,822)"
string disp2 = Fmt.W(12).C.M.N(plusV).P     # disp2 is set to:        "     78,845 "
string disp3 = Fmt.W(8).Z.N(plusV).P        # disp3 gets value:       "00078845"
```

Localization of Formatting

Default localization is in the **std/config.ss** file.

The pragma statement can control some options in formatting:

pragma locale *option value*, ...

where *option* (not reserved names) and *value* can be:

Table: Locale Options

Option:	Value:	Description:
thousands	',' (default), '' or ' '	separate every 3 digits in integer part of float output if requested
decimal_point	'.' (default) or ','	decimal point in input/output

