Harlequin Dylan

System and I/O Reference

Library Reference

Version 2.0 Beta

Copyright and Trademarks

Harlequin Dylan: Library Reference: System and I/O

Version 2.0 Beta

May 1999

Part number: 3DPDTZ-15MD

Companies, names and data used in examples herein are fictitious unless otherwise noted.

The information in this publication is provided for information only and is subject to change without notice. Harlequin Group plc and its affiliates assume no responsibility or liability for any loss or damage that may arise from the use of any information in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

US Government Use

The Harlequin Dylan Software is a computer software program developed at private expense and is subject to the following Restricted Rights Legend: “Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in (i) FAR 52.227-14 Alt III or (ii) FAR 52.227-19, as applicable. Use by agencies of the Department of Defense (DOD) is subject to Harlequin’s customary commercial license as contained in the accompanying license agreement, in accordance with DFAR 227.7202-1(a). For purposes of the FAR, the Software shall be deemed to be `unpublished’ and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Harlequin Incorporated, One Cambridge Center, Cambridge,
Massachusetts 02142.”

http://www.harlequin.com/

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
UK

telephone +44 1223 873 800
fax +44 1223 873 873

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
USA

telephone +1 617 374 2400
fax +1 617 252 6505

Asia Pacific:

Harlequin Australia Pty. Ltd.
Level 12
12 Moore Street
Canberra, ACT 2601
Australia

telephone +61 2 6206 5522
fax +61 2 6206 5525

Contents

1 Introduction

This book provides reference documentation for the Harlequin Dylan libraries and modules that support: printing and output formatting (including a streams module), interfaces to operating system features (such as the file system, time and date information, and the host machine environment), and network protocols and sockets.

In general, each module is documented in a separate chapter.

1.1 The libraries and their modules

The three libraries described here are: the IO library, the System library and the Network library. Each library exports a number of modules designed to provide interfaces and functionality relevant to output formatting, system and networking operations.

The IO library exports:

- the FORMAT module, which extends the functionality of format strings and provides functions for processing the extended format strings
- the FORMAT-OUT module, which repackages the FORMAT module and the STANDARD-IO module for more convenient use
- the PRINT module, which provides interfaces for printing

- the PPRINT module, which provides interfaces for pretty printing
- the STREAMS module, which allows you to establish and control input to and output from aggregates of data
- the STANDARD-IO module, which provides an interface to the standard I/O facility of operating systems like MS-DOS and UNIX

The System library exports:

- the DATE module, which allows for machine-independent representation and manipulation of dates and date/time intervals
- the FILE-SYSTEM module, which provides a generic interface to the file system of the local machine
- the OPERATING-SYSTEM module, which provides an interface to some of the features of the host machine's operating system

The Network library (which is not available in the Personal Edition) exports:

- the SOCKETS module, which provides Internet address protocols, TCP/IP server and client sockets, and UDP sockets

2

The Format Module

2.1 Introduction

This chapter describes the Format module. The Format module is exported from the IO library. This module extends the functionality of the format strings described in Dylan's condition system and provides two new functions for processing the extended format strings. The Format module is a small module, but it uses the printing modules and some of the Streams module. Chapter 4, "The Printing Modules" and Chapter 5, "The Streams Module" give full details of the Print and Streams libraries.

The **`format`** module exports all the identifiers described in this chapter.

2.2 Control strings

The Format module's format strings, or control strings, offer the same directives as Dylan's format strings offer, but Format provides a few more directives, and permits a single argument to all format directives.

The argument is an integer that must appear contiguously between the dispatch character, `%`, and the format directive. The argument indicates a printing field in which to justify the output of the directive. A positive integer indicates that the output should be flush right within the field, and a negative integer indicates the output should be flush left within the field. If the output length is greater than the field’s width, then output occurs as if there were no field specification. The following are examples of valid format directives:

```
%S

%s

%15D

%-10=
```

The directives are:

`%S` Prints the next format argument as a message by calling the function **`print-message`** on the format argument and the stream. This directive is the same as Dylan’s **`%S`** format-string directive except for two features: (i) this module’s **`%S`** directive outputs character objects, and (ii) you can extend the **`%S`** functionality by adding methods to **`print-message`**.

`%=` Prints the next format argument by calling the **`print`** function from the Print module on the format argument and the stream. You can extend the **`%=`** functionality by adding methods to the **`print-object`** function from the Print module.

`%C` Print the next format argument, which must be a character, according to Dylan’s **`%S`** format-string directive. This module’s **`%C`** directive is the same as this module’s **`%S`** directive.

`D` Prints a decimal representation of the next format argument, which must be an integer.

`%B` Prints a binary representation of the next format argument, which must be an integer.

`%O` Prints an octal representation of the next format argument, which must be an integer.

`%X` Prints a hexadecimal representation of the next format argument, which must be an integer.

`%M`	Invokes the next format argument, which must be a function, on the stream passed to **`format`**.
`%%`	Outputs a single `%` character.

2.3 The FORMAT module

This section contains a reference entry for each item exported from the Format module.

format *Function*

Summary
Outputs a control string to a stream.

Signature
`format` ***stream control-string arguments*** **`=> ()`**

Arguments

stream	An instance of **`<stream>`**. The stream to which formatted output should be sent.
control-string	An instance of **`<string>`**. A string containing format directives.
arguments	Instances of **`<object>`**.

Values
None.

Description
Sends output to *stream* according to the format directives in *control-string*. Each directive consumes one argument from *arguments*. See Section 2.2 on page 1 for a description of the control strings that can be used.

The *control-string* contents that are not part of any directive are output directly to *stream*, as if by the Streams module's **`write`** function.

format *G.f. method*

Summary
: Outputs a control string to a stream.

Arguments

stream	An instance of **`<stream>`**.
control-string	An instance of **`<byte-string>`**.
arguments	Instances of **`<object>`**.

Values
: None.

Description
: There is one method for **`format`**, and it is specialized to **`<byte-string>`**.

format-to-string *Function*

Summary
: Returns a formatted string based on a format control string.

Arguments

control-string	An instance of **`<string>`**.
arguments	Instances of **`<object>`**.

Values

result	An instance of **`<string>`**.

Description
: Calls **`format`** to produce output according to *control-string* and returns the output as a string.

format-to-string *G.f. method*

Summary
: Returns a formatted string based on a format control string.

Arguments

control-string	An instance of **`<byte-string>`**.
arguments	Instances of **`<object>`**.

Values

result	An instance of **`<byte-string>`**.

Description

There is one method for **`format-to-string`**. The *control-string* argument must be a **`<byte-string>`**. Result is a **`<byte-string>`**.

print-message *Function*

Summary

Prints an object to a stream.

Arguments

object	An instance of **`<object>`**.
stream	An instance of **`<stream>`**.

Values

None.

Description

Prints *object* to *stream*.

Methods for this function should print objects as a message, as opposed to printing them in any form intending to represent Dylan data, literal syntax, and so on.

For example, printing a condition object with this function presents the condition as an error message, but printing the condition object with the **`print`** function from the Print module prints the condition in some form such as

```
{Simple-error}
```

See the individual methods for the details of how this function prints various objects. This function exists to define the behavior of the **`%S`** format directive and to allow users the ability to extend the **`%S`** directive. Users should have little need to call this function directly.

print-message *Sealed g.f method*

Summary

Prints a condition to a stream as an error message.

Arguments	*condition*	An instance of `<condition>`.
	stream	An instance of `<stream>`.

Values None.

Description Prints *condition* as an error message, as described for the Dylan `%S` format directive. You should not specialize the `print-message` protocol for subclasses of `<condition>`, but instead extend the `print-message` protocol to new condition objects by specializing methods on `report-condition`.

print-message *Sealed g.f. method*

Summary Prints a symbol to a stream.

Signature `print-message` ***symbol stream*** `=> ()`

Arguments	*symbol*	An instance of `<symbol>`.
	stream	An instance of `<stream>`.

Values None.

Description Prints *symbol* to *stream* by converting it to a string with the `as` function and then writing the string with the `write` function from the Streams module.

print-message *Sealed g.f. method*

Summary Prints an object to a stream.

Signature `print-message` ***object stream*** `=> ()`

Arguments	*object*	An instance of `type-union(<string>, <character>)`.

stream An instance of **`<stream>`**.

Values None.

Description Prints *object* to *stream* by calling the **`write`** function from the **`streams`** module.

3

The Format-Out Module

3.1 Introduction

The Format-Out module is a convenient repackaging of two libraries that provides a simple way to send text to the platform's standard output stream. For this purpose, Format-Out uses the Format module and the Standard-IO module and defines a function **`format-out`**. The Format-Out module exports all the identifiers described in this document. The Format-Out module re-exports two modules, **`format`** from the Format library and **`standard-io`** from the Standard-IO library.

Chapter 2, "The Format Module", and Chapter 6, "The Standard-IO Module" give full details of the Format and Standard-IO libraries.

3.2 The FORMAT-OUT module

This section contains a reference entry for each item exported from the **`format-out`** module.

format-out *Function*

Summary Formats its arguments on the standard output.

Signature **format-out** ***control-string*** **#rest** ***arguments*** **=> ()**

Arguments		
	control-string	An instance of **<string>**.
	arguments	Instances of **<object>**.

Values None.

Description Calls the **format** function from the **format** module on ***standard-output*** from the **standard-io** module, *control-string*, and *arguments*.

See also **format**, page 3

standard-output, page 117

format-out *Sealed g.f. method*

Summary Formats its arguments on the standard output.

Signature **format-out** ***control-string*** **#rest** ***arguments*** **=> ()**

Arguments		
	control-string	An instance of **<byte-string>**.
	arguments	Instances of **<object>**.

Values None.

Description Formats its arguments on the standard output. There is one method for **format-out**, and it is specialized to instances of **<byte-string>**.

4

The Printing Modules

4.1 Introduction

The IO library's printing modules provide an interface that outputs an object in Dylan literal syntax if the object can be represented as a Dylan literal, and otherwise, outputs the object in an implementation-dependent manner. There are two functions, **`print`** and **`print-object`**. The **`print`** function accepts keyword arguments that form a print request, controlling features such as circular printing, how deep within a data structure to print, how many elements in long sequences to print before using an ellipsis notation, whether pretty printing is desired, and so on. Users extend **`print`**'s ability to print various objects by adding methods to the **`print-object`** function. The **`print`** function handles most of the overhead to satisfy special print requests, outputting any special notations required, and it only calls **`print-object`** when it is necessary to print objects. Users should always call the **`print`** function to output objects, especially recursively from within **`print-object`** methods to output an object's components. Users should never call **`print-object`** directly.

The IO library exports two modules for use with printing, **`print`** and **`pprint`**. Reference entries for the interfaces exported from the **`print`** module can be found in Section 4.4 on page 18, and reference entries for interfaces exported from the **`pprint`** module are in Section 4.5 on page 23.

These modules uses the Streams module. See Chapter 5 for full details of the Streams module.

4.2 Print functions

The Print module offers two functions for users to call to print objects, **`print`** and **`print-to-string`**.

print *Function*

`print` ***object stream*** **`#key`** ***level length circle? pretty?*** **`=> ()`**

Prints *object* to *stream* according to the print request formed by the keyword arguments. A first call to **`print`** creates a printing stream to represent the print request, and recursive calls to **`print`** on this printing stream process the keyword arguments differently (see below). There are inspection functions for querying the print request. When **`print`** actually prints an object, it calls **`print-object`**. Though the inspection functions for querying the print request allow you to inspect any parameter of the print request, **`print-object`** methods should only need to call **`print-length`**. All other aspects of the print request are handled by **`print`**. There is one exception which is described in Section 4.3.

The *level* keyword controls how deep into a nested data structure to print. The value **`#f`** indicates that there is no limit. The default, **`*print-level*`**, has no effect on recursive calls to **`print`**. Recursive calls to **`print`** may change the value of **`print-level`** explicitly, but **`print`** always uses a value to ensure the print request formed by the first call to **`print`** is never exceeded. For example, if a first call to **`print`** set the level to 5, and while at a depth of 3, a recursive call specified a level of 4, the recursive call would only descend 2 more levels, not 4.

The *length* keyword controls how many elements of a sequence to print before printing ellipsis notation (**`...`**). The value **`#f`** indicates that there is no limit. The **`print-length`** control can be interpreted loosely by some **`print-object`** methods to control how many *elements* of any kind of object to print; for example, the default **`<object>`** method might regard **`print-length`** to determine how many slot-name/value pairs to print. The default, **`*print-length*`**, has no effect on recursive calls to **`print`**.

Recursive calls to **`print`** may change the value of **`print-length`** explicitly, but they may only decrease the value, never increase it.

The *circle?* keyword indicates whether printing should check all subcomponent references to make sure the printing process does not infinitely recurse through a data structure. Circular printing also tags objects that occur more than once when they are first printed, and later occurrences are printed as a reference to the previously emitted tag. The default, **`*print-circle?*`**, has no effect on recursive calls to **`print`**. If **`print-circle?`** is already **`#t`**, then it remains **`#t`** throughout all recursive calls. If **`print-circle?`** is **`#f`**, then recursive calls to **`print`** can change the value to **`#t`**; however, when printing exits the dynamic scope of the call that changed the value to **`#t`**, the value reverts back to **`#f`**. If the original call to **`print`** specifies *circle?* as **`#f`**, and dynamically distinct recursive calls turn circular printing on and off, all output generated while circular printing was on shares the same tagging space; that is, if **`#1#`** is printed twice, once from each of two distinct recursive calls to print, then each **`#1#`** is guaranteed to signify the same **`==`** object.

The *pretty?* keyword indicates whether printing should attempt to insert line breaks and indentation to format objects according to how programmers tend to find it easier to read data. The default, **`*print-pretty?*`**, has no effect on recursive calls to **`print`**. If **`print-pretty?`** is already **`#t`**, then it remains **`#t`** throughout all recursive calls. If **`print-pretty?`** is **`#f`**, then recursive calls to **`print`** can change the value to **`#t`**; however, when printing exits the dynamic scope of the call that changed the value to **`#t`**, the value reverts back to **`#f`**.

print-to-string *Function*

`print-to-string` ***object*** **`#key`** ***level length circle? pretty?*** **`=>`** ***result***

Calls **`print`** to produce output according to the print request formed by the keyword arguments and returns the result as a string. The *level, length, circle?,* and *pretty?* keywords are as for **`print`**.

print-object *Open generic function*

`print-object` ***object stream*** **`=> ()`**

Prints an *object* to a *stream*. You should extend the ability of **`print`** to print various objects by adding methods to the **`print-object`** function. When **`print`** actually prints an object, it calls **`print-object`**. You should never call **`print-object`** directly.

The Print module exports the following variables which provide default values for calls to the print function. Their values are implementation-dependent.

print-level *Variable*

This is an **`<integer>`** that controls how deeply into a nested expression to print.

print-length *Variable*

This is an **`<integer>`** that controls how many elements at a given level to print.

print-circle? *Variable*

A boolean that controls whether or not to print recursively. When **`*print-circle*`** is **`#f`**, printing proceeds recursively and attempts to print a circular structure results in failure to terminate.

print-pretty *Variable*

A boolean that controls whether or not print does *pretty-printing*.

4.3 Pretty printing

When writing **`print-object`** methods, you can ignore whether pretty printing is in effect. If you write your **`print-object`** method using pretty printing functions, then when pretty printing is in effect, the output is pretty printed. When pretty printing is not in effect, your method produces output as though you had not written it to use pretty printing. All **`print-object`** methods that are written to do pretty printing must call the pretty printing functions within the dynamic scope of a call to **`pprint-logical-block`**; otherwise, the pretty printing functions are no-ops.

The following interfaces are exported from the **`pprint`** module:

default-line-length *Variable*

An integer that controls the line length used by the pretty printer to determine how much output will fit on a single line. The value must be an integer. The default is 80.

print-miser-width *Variable*

An integer that controls *miser mode*. Whenever a logical block (see **`pprint-logical-block`**) begins in a column of output that is greater than **`*default-line-length* - *print-miser-width*`**, then pretty printing is in miser mode. The value must be an integer or **`#f`** (the default). **`#f`** indicates that the pretty printer should never enter miser mode.

pprint-logical-block *Function*

`pprint-logical-block` ***stream*** **`#key`** ***prefix per-line-prefix body suffix column*** **`=> ()`**

Groups printing into a logical block. The logical block provides boundaries for new levels of indentation, affects **`#"linear"`** newlines, and so on. The *prefix* keyword is a string to print at the beginning of the logical block. The blocks indentation is automatically set to be one character position greater than the column in which *prefix* ends. Alternatively, *per-line-prefix* is a string to print on every line of the logical block. The **`pprint-logical-block`** function signals an error if it is called with both *prefix* and *per-line-prefix* supplied as non-**`#f`**.

The *suffix* keyword is a string to print at the end of the logical block.

The *column* keyword advises the pretty printer as to the current column of the output stream (the default is zero). This keyword may be ignored entirely by some methods, and it may be ignored in some cases by methods that can better determine the stream's current output column.

The *body* keyword must be a function that can take one argument, and this argument is a stream. The function specified by *body* should use the *stream* argument passed to it; the *body* function should not close over the *stream* argument to **`pprint-logical-block`**. The function **`pprint-logical-block`** wraps *stream* with a pretty printing stream when *stream* is any other kind of stream. If *stream* is already a pretty printing stream, then the *body* function is called on *stream*.

All **`print-object`** methods that are written to do pretty printing must call the other pretty printing functions within the dynamic scope of a call to **`pprint-logical-block`**; otherwise, the pretty printing functions are no-ops.

pprint-newline *Function*

`pprint-newline` ***kind stream*** **`=> ()`**

Announces a conditional newline to the pretty printer. The pretty printer emits a newline depending on the *kind* and the state of the pretty printer's current line buffer. The *kind* argument can be one of the following:

`#"fill"` Emit a newline if the current *section* of output does not fit on one line.

`#"linear"` Emit a newline if any **`#"linear"`** newline in the current *section* needs to be emitted. That is, if a current *section* of output cannot fit on one line, and any one of the **`#"linear"`** newlines in the section needs to be emitted, then emit them all.

`#"miser"` Emit a newline as if it were a **`#"linear"`** newline, but only when *miser mode* is in effect. Miser style is in effect when a logical block starts past a particular column of output.

#"mandatory" Emit a newline always. Establish that any containing *sections* cannot be printed on a single line so that **#"linear"** and **#"miser"** newlines will be emitted as appropriate.

pprint-indent

Function

pprint-indent ***relative-to n stream*** **=> ()**

Specifies the indentation to use within the current logical block. When *relative-to* is **#"block"**, then **pprint-indent** sets the indentation to the column of the first character of the logical block plus *n*. When *relative-to* is **#"current"**, then **pprint-indent** sets the indentation to the current column plus *n*. In both cases, *n* is a **<fixed-integer>**.

pprint-tab

Function

pprint-tab ***kind colnum colinc stream*** **=> ()**

kind One of **#"line"**, **#"line-relative"**, **#"section"**, **#"section-relative"**.

colnum An instance of **<fixed-integer>**.

colinc An instance of **<fixed-integer>**.

stream An instance of **<stream>**.

Announces a tab to the pretty printer. *Colnum* and *colinc* have meaning based on the value of *kind*, which can be one of the following:

#"line" Tab to output column *colnum*. If the output is already at or beyond *colnum*, then add *colinc* to *colnum* until printing can continue at a column beyond the end of the output already on the line.

#"line-relative"

Output *colnum* spaces. Then output enough spaces to tab to a column that is a multiple of *colinc* from the beginning of the line.

`#"section"` This is similar to **`#"line"`**, but column counting is relative to the beginning of the current *section* rather than the beginning of the line.

`#"section-relative"`

This is similar to **`#"line-relative"`**, but column counting is relative to the beginning of the current *section* rather than the beginning of the line.

In all cases, *colnum* and *colinc* are instances of **`<fixed-integer>`**.

4.4 The PRINT module

This section contains a reference entry for each item exported from the IO library's **`print`** module.

print *Function*

Summary Prints *object* to the specified stream.

Signature **`print`** ***object stream*** **`#key`** ***level length circle? pretty?*** **`=> ()`**

Arguments

object	An instance of **`<object>`**.
stream	An instance of **`<stream>`**.
level	**`#f`** or an instance of **`<fixed-integer>`**. Default value: **`*print-level*`**.
length	**`#f`** or an instance of **`<fixed-integer>`**. Default value: **`*print-length*`**.
circle?	An instance of **`<boolean>`**. Default value: **`*print-circle?*`**.
pretty?	An instance of **`<boolean>`**. Default value: **`*print-pretty?*`**.

Values None.

Description

Prints *object* to *stream* according to the print request formed by the keyword arguments. A first call to **print** creates a printing stream to represent the print request, and recursive calls to **print** on this printing stream process the keyword arguments differently (see below). There are inspection functions for querying the print request. When **print** actually prints an object, it calls **print-object**. Though the inspection functions for querying the print request allow you to inspect any parameter of the print request, **print-object** methods should only need to call **print-length**. All other aspects of the print request are handled by **print**. There is one exception, which is described in Section 4.3 on page 14.

The *level* keyword controls how deep into a nested data structure to print. The value **#f** indicates that there is no limit. The default, ***print-level***, has no effect on recursive calls to **print**. Recursive calls to **print** may change the value of **print-level** explicitly, but **print** always uses a value to ensure the print request formed by the first call to **print** is never exceeded. For example, if a first call to **print** set the level to 5, and while at a depth of 3, a recursive call specified a level of 4, the recursive call would only descend 2 more levels, not 4.

The *length* keyword controls how many elements of a sequence to print before printing ellipsis notation (**...**). The value **#f** indicates that there is no limit. The **print-length** control can be interpreted loosely by some **print-object** methods to control how many *elements* of any kind of object to print; for example, the default **<object>** method might regard **print-length** to determine how many slot-name/value pairs to print. The default, ***print-length***, has no effect on recursive calls to **print**. Recursive calls to **print** may change the value of **print-length** explicitly, but they may only decrease the value, never increase it.

The *circle?* keyword indicates whether printing should check all subcomponent references to make sure the printing process does not infinitely recurse through a data structure. Circular printing also tags objects that occur more than once when they are first printed, and later occurrences are printed as a reference to the previously emitted tag. The default, **`*print-circle?*`**, has no effect on recursive calls to **`print`**. If **`print-circle?`** is already **`#t`**, then it remains **`#t`** throughout all recursive calls. If **`print-circle?`** is **`#f`**, then recursive calls to **`print`** can change the value to **`#t`**; however, when printing exits the dynamic scope of the call that changed the value to **`#t`**, the value reverts back to **`#f`**. If the original call to **`print`** specifies *circle?* as **`#f`**, and dynamically distinct recursive calls turn circular printing on and off, all output generated while circular printing was on shares the same tagging space; that is, if **`#1#`** is printed twice, once from each of two distinct recursive calls to print, then each **`#1#`** is guaranteed to signify the same **`==`** object.

The *pretty?* keyword indicates whether printing should attempt to insert line breaks and indentation to format objects according to how programmers tend to find it easier to read data. The default, **`*print-pretty?*`**, has no effect on recursive calls to **`print`**. If **`print-pretty?`** is already **`#t`**, then it remains **`#t`** throughout all recursive calls. If **`print-pretty?`** is **`#f`**, then recursive calls to **`print`** can change the value to **`#t`**; however, when printing exits the dynamic scope of the call that changed the value to **`#t`**, the value reverts back to **`#f`**.

print-circle? *Variable*

Summary Controls whether or not to print recursively.

Type **`<boolean>`**

Initial value None.

Description — Controls whether or not to print recursively. When **`*print-circle*`** is **`#f`**, printing proceeds recursively and attempts to print a circular structure results in failure to terminate.

print-length — *Variable*

Summary — Controls the number of elements of an expression to print.

Type — **`false-or(<integer>)`**

Initial value — None.

Description — Controls how many elements to print at a given level of a nested expression.

print-level — *Variable*

Summary — Controls how deeply into a nested expression to print.

Type — **`false-or(<integer>)`**

Initial value — None.

Description — Controls how many levels of a nested expression to print.

print-object — *Open generic function*

Summary — Prints an object to a stream.

Signature — **`print-object`** ***object stream*** **`=> ()`**

Arguments

object	An instance of **`<object>`**.
stream	An instance of **`<stream>`**.

Values None.

Description Prints an object to a stream. You should extend the ability of **`print`** to print various objects by adding methods to the **`print-object`** function. When **`print`** actually prints an object, it calls **`print-object`**. You should never call **`print-object`** directly.

print-pretty *Variable*

Summary Controls whether or not pretty printing is used.

Type **`<boolean>`**

Initial value None.

Description Controls whether or not **`print`** does pretty printing.

print-to-string *Function*

Summary Calls **`print`** on *object*.and returns the result as a string.

Signature **`print-to-string`** ***object*** **`#key`** ***level length circle? pretty?*** **`=>`** ***result***

Arguments

object	An instance of **`<object>`**.
level	**`#f`** or an instance of **`<fixed-integer>`**. Default value: **`*print-level*`**.
length	**`#f`** or an instance of **`<fixed-integer>`**. Default value: **`*print-length*.`**
circle?	An instance of **`<boolean>`**. Default value: **`*print-circle?*`**.
pretty?	An instance of **`<boolean>`**. Default value: **`*print-pretty?*.`**

Values *result* An instance of **`<byte-string>`**.

Description Calls **`print`** to produce output according to the print request formed by the keyword arguments and returns the result as a string.

4.5 The PPRINT module

This section contains a reference entry for each item exported from the IO library's **`pprint`** module.

default-line-length *Variable*

Summary Controls the default line length used by the pretty printer.

Type **`<integer>`**

Initial value 80

Description Controls the line length used by the pretty printer to determine how much output will fit on a single line. The value must be an integer.

pprint-indent *Function*

Summary Specifies the indentation to use within the current logical block.

Signature **`pprint-indent`** ***relative-to n stream*** **`=> ()`**

Arguments

relative-to	One of **`#"block"`** or **`#"current"`**.
n	An instance of **`<fixed-integer>`**.
stream	An instance of **`<stream>`**.

Values None.

Description Specifies the indentation to use within the current logical block. When *relative-to* is **#"block"**, then **pprint-indent** sets the indentation to the column of the first character of the logical block plus *n*. When *relative-to* is **#"current"**, then **pprint-indent** sets the indentation to the current column plus *n*.

pprint-logical-block *Function*

Summary Groups printing into a logical block.

Signature **pprint-logical-block** ***stream*** **#key** ***prefix per-line-prefix body suffix column*** **=> ()**

Arguments

stream	An instance of **<stream>**.
prefix	**#f** or an instance of **<byte-string>**.
per-line-prefix	**#f** or an instance of **<byte-string>**.
body	An instance of **<function>**.
suffix	**#f** or an instance of **<byte-string>**.
column	A *limited* instance of **<fixed-integer>**, minimum 0.

Values None.

Description Groups printing into a logical block. The logical block provides boundaries for new levels of indentation, affects **#"linear"** newlines, and so on. *Prefix* is a string to print at the beginning of the logical block. The blocks indentation is automatically set to be one character position greater than the column in which *prefix* ends. Alternatively, *per-line-prefix* is a string to print on every line of the logical block. This function signals an error if it is called with both *prefix* and *per-line-pre-*

fix supplied as non-**#f**. *Suffix* is a string to print at the end of the logical block. *Column* advises the pretty printer as to the current column of the output stream (the default is zero). The *column* argument may be ignored entirely by some methods, and it may be ignored in some cases by methods that can better determine the stream's current output column.

The *body* keyword must be a function that can take one argument, and this argument is a stream. The *body* function should use the stream argument passed to it; the *body* function should not close over the stream argument to **pprint-logical-block**. **Pprint-logical-block** wraps *stream* with a pretty printing stream when *stream* is any other kind of stream. If *stream* is already a pretty printing stream, then the *body* function is called on *stream*.

All **print-object** methods that are written to do pretty printing must call the other pretty printing functions within the dynamic scope of a call to **pprint-logical-block**; otherwise, the pretty printing functions are no-ops.

pprint-newline *Function*

Summary

Announces a conditional newline to the pretty printer.

Signature

pprint-newline ***kind stream*** **=> ()**

Arguments

kind	One of **#"fill"**, **#"linear"**, **#"miser"**, **#"mandatory"**.
stream	An instance of **<stream>**.

Values

None.

Description

Announces a conditional newline to the pretty printer. The pretty printer emits a newline depending on the *kind* and the state of the pretty printer's current line buffer. The *kind* argument has roughly the following meanings:

#"fill"	Emit a newline if the current *section* of output does not fit on one line.
#"linear"	Emit a newline if any **#"linear"** newline in the current *section* needs to be emitted. That is, if a current *section* of output cannot fit on one line, and any one of the **#"linear"** newlines in the section needs to be emitted, then emit them all.
#"miser"	Emit a newline as if it were a **#"linear"** newline, but only when *miser mode* is in effect. Miser style is in effect when a logical block starts past a particular column of output.
#"mandatory"	Emit a newline always. Establish that any containing *sections* cannot be printed on a single line so that **#"linear"** and **#"miser"** newlines will be emitted as appropriate.

pprint-tab *Function*

Summary

Announces a tab to the pretty printer.

Signature

pprint-tab ***kind colnum colinc stream*** **=> ()**

Arguments

kind	One of **#"line"**, **#"line-relative"**, **#"section"**, **#"section-relative"**.
colnum	An instance of **<fixed-integer>**.
colinc	An instance of **<fixed-integer>**.
stream	An instance of **<stream>**.

Values

None.

Description

Announces a tab to the pretty printer. The *colnum* and *colinc* arguments have meaning based on the value of *kind*:

#"line"
: Tab to output column *colnum*. If the output is already at or beyond *colnum*, then add *colinc* to *colnum* until printing can continue at a column beyond the end of the output already on the line.

#"line-relative"
: Output *colnum* spaces. Then output enough spaces to tab to a column that is a multiple of *colinc* from the beginning of the line.

#"section"
: Similar to **#"line"**, but column counting is relative to the beginning of the current *section* rather than the beginning of the line.

#"section-relative"
: Similar to **#"line-relative"**, but column counting is relative to the beginning of the current *section* rather than the beginning of the line.

print-miser-width *Variable*

Summary

Controls miser mode.

Type

false-or(<integer>)

Initial value

None.

Description

Controls *miser mode*. Pretty printing is in miser mode whenever a logical block (see **pprint-logical-block**) begins in a column of output that is greater than

```
*default-line-length* - *print-miser-width*
```

The value must be an integer or **`#f`** (the default); **`#f`** indicates that the pretty printer should never enter miser mode.

5

The Streams Module

5.1 Introduction

This chapter describes the Streams module, which allows you to establish and control input to and output from aggregates of data, such as files on disk, or sequences. This module, together with the Standard-IO module, provides similar functionality to the **`Java.io`** package in Java. See Chapter 6, "The Standard-IO Module", for details about the Standard-IO module in Dylan.

Section 5.4 on page 31 discusses the basic concepts involved in streaming over data. Section 5.5 on page 35 describes the different classes of stream available, and how to create them, and Section 5.6 on page 44 describes how to read from and write to them.

More specialized subjects are covered next: Section 5.5.3 on page 42 discusses locking streams while they are in use; Section 5.7 on page 51 describes using buffered streams; Section 5.8 on page 52 describes wrapper streams; Section 5.9 on page 55 the different stream-specific error conditions that can be raised.For the most part, you do not have to worry about the information in these later sections when using streams.

Finally, Section 5.11 on page 59 gives complete details on all interfaces in the Streams module. Each entry in this section is arranged in alphabetical order.

5.2 Discussing error conditions

This chapter uses two special terms in discussions of error conditions.

When it notes that something *is an error*, this means that the result is undefined. In particular, it does not *necessarily* mean that an error condition will be signalled. So, for instance, the following example text means only that the result of using **`pull-stream-element`** in the case described is undefined:

> It is an error to apply **`pull-stream-element`** to an element that has already been read from the stream.

A given function is only guaranteed to raise an exception in response to an error if the documentation for that function specifically states that it will signal an error. Note that the specific error condition that is signaled may depend on the program state; in such situations, the specific error condition is not stated in the documentation. Consider the following hypothetical example, which states that an implementation must signal an error, but does not say what error must be signaled:

> When *index* is a **`<stream-index>`**, if it is invalid for some reason, this function signals an error.

By contrast, the following example names the class of which the condition signaled is guaranteed to be a general instance:

> If the end of the stream is encountered and no value was supplied for *on-end-of-stream*, **`read-last-element`** signals an **`<end-of-stream-error>`** condition.

If the name of the condition class is given, applications are permitted to specialize error handlers on that class.

5.3 Goals of the module

The Streams module provides:

- A generic, easy-to-use interface for streaming over sequences and files. The same high-level interface for consuming or producing is available irrespective of the type of stream, or the types of the elements being streamed over.

- Efficiency, especially for the common case of file I/O.
- Access to an underlying buffer management protocol.

The Streams module does not address a number of related issues, including:

- A standard object-printing package such as Smalltalk's **`printOn:`** or Lisp's **`print-object`**, or a formatted printing facility such as Lisp's **`format`**. These facilities are provided by the Print, Format, and Format-out libraries. For convenience, the Harlequin-Extensions library also provides simple formatting capabilities.
- General object dumping and loading.
- A comprehensive range of I/O facilities for using memory-mapped files, network connections, and so on.
- An interface for naming files. The Locators module provides such an interface.
- An interface to operating system functionality, such as file renaming or deleting operations. The File-System module provides such an interface.

5.4 Concepts

A *stream* provides sequential access to an aggregate of data, such as a Dylan sequence or a disk file. Streams grant this access according to a metaphor of *reading* and *writing*: elements can be read from streams or written to them.

Streams are represented as Dylan objects, and all are general instances of the class **`<stream>`**, which the Streams module defines.

It is usual to say that a stream is established *over* the data aggregate. Hence, a stream providing access to the string **`"hello world"`** is said to be a stream over the string **`"hello world"`**.

Streams permitting reading operations are called *input* streams. Input streams allow elements from the underlying data aggregate to be consumed. Conversely, streams permitting writing operations are called *output* streams. Output streams allow elements to be written to the underlying data aggregate. Streams permitting both kinds of operations are called *input-output* streams.

The Streams module provides a set of functions for reading elements from an input stream. These functions hide the details of indexing, buffering, and so on. For instance, the function **`read-element`** reads a single data element from an input stream.

The following expression binds **`stream`** to an input stream over the string **`"hello world"`**:

```
let stream = make(<string-stream>, contents: "hello world");
```

The first invocation of **`read-element`** on **`stream`** returns the character “h”, the next invocation “e”, and so on. Once a stream has been used to consume all the elements of the data, the stream is said to be at its end. This condition can be tested with the function **`stream-at-end?`**. The following code fragment applies **`my-function`** to all elements of the sequence:

```
let stream = make(<sequence-stream>, contents: seq);
while (~stream-at-end?(stream))
  my-function(read-element(stream));
end;
```

When all elements of a stream have been read, further calls to **`read-element`** result in the **`<end-of-stream-error>`** condition being signaled. An alternative end-of-stream behavior is to have a distinguished end-of-stream value returned. You can supply such an end-of-stream value as a keyword argument to the various read functions; the value can be any object. Supplying an end-of-stream value to a read function is more concise than asking whether a stream is at its end on every iteration of a loop.

The Streams module also provides a set of functions for writing data elements to an output stream. Like the functions that operate upon input streams, these functions hide the details of indexing, growing an underlying sequence, buffering for a file, and so on. For instance, the function **`write-element`** writes a single data element to an output stream.

The following forms bind **`stream`** to an output stream over an empty string and create the string “I see!”, using the function **`stream-contents`** to access all of the stream’s elements.

```
let stream = make(<byte-string-stream>, direction: #"output");
write(stream, "I see!");
stream-contents(stream);
```

Calling **write** on a sequence has the same effect as calling **write-element** on all the elements of the sequence. For more information about writing to streams, see Section 5.6.3 on page 46.

Some streams are *positionable*; that is, any element of the stream can be accessed at any time. Positionable streams allow you to set the position at which the stream is accessed by the next operation. The following example uses positioning to return the character "w" from a stream over the string **"hello world"**:

```
let stream = make(<string-stream>, contents: "hello world");
stream-position(stream) := 6;
read-element(stream);
```

The following example returns a string. The first ten characters are the fill characters for the underlying sequence of the stream. The fill character for **<string>** is " " (the space character), so in the example below, the first ten characters are spaces.

```
let stream = make(<string-stream>, direction: #"output");
adjust-stream-position(stream, 10);
write(stream, "whoa!");
stream-contents(stream);
```

You can request a sequence containing all of the elements of a positionable stream by calling **stream-contents** on it. If the positionable stream is a **<file-stream>**, then it must be readable. Otherwise, it must be a sequence stream. The sequence returned never shares structure with any underlying sequence that might be used in the future by the stream. For instance, the string returned by calling **stream-contents** on an output **<string-stream>** will not be the same string as that being used to represent the string stream.

When making an input **<string-stream>**, you can cause the stream to produce elements from any subsequence of the supplied string. For example:

```
read-to-end(make(<string-stream>,
                 contents: "hello there, world",
                 start: 6,
                 end: 11));
```

This example evaluates to **"there"**. The interval (*start*, *end*) includes the index *start* but excludes the index *end*. This is consistent with standard Dylan functions over sequences, such as **copy-sequence**. The **read-to-end** function is one of a number of convenient utility functions for operating on streams and returns all the elements up to the end of the stream from the stream's current position.

5.4.1 Streams, growing sequences, and object identity

When writing to output streams over sequences, Dylan may from time to time need to grow the underlying sequence that it is using to represent the stream data.

Consider the example of an output stream instantiated over an empty string. As soon as a write operation is performed on the stream, it is necessary to replace the string object used in the representation of the string stream. As well as incurring the cost of creating a new string, references to the string within the program after the replacement operation has occurred will still refer to the *original* string, and this may not be what the user intended.

To guarantee that other references to a sequence used in an output **`<sequence-stream>`** will have access to any elements written to the sequence via the stream, supply a stretchy collection (such as a **`<stretchy-vector>`**) to **`make`**. A stream over a stretchy vector will use the same stretchy vector throughout the stream's existence.

For example:

```
let sv = make(<stretchy-vector>);

let stream = make(<sequence-stream>,
                  contents: sv,
                  direction: #"output");

write(stream,#(1, 2, 3, 4, 5, 6, 7, 8, 9));

write(stream,"ABCDEF");

values(sv, stream-contents(stream));
```

The example returns two values. Each value is the same (**`==`**) stretchy vector:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, ’A’, ’B’, ’C’, ’D’, ’E’, ’F’)
```

If a stretchy vector is not supplied, the result is different:

```
let v = make(<vector>, size: 5);
let stream = make(<sequence-stream>,
                  contents: v,
                  direction: #"output");

write(stream,#(1, 2, 3, 4, 5, 6, 7, 8, 9));

write(stream,"ABCDEF");

values(v, stream-contents(stream));
```

This example returns as its first value the original vector, whose contents are unchanged, but the second value is a new vector:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, ’A’, ’B’, ’C’, ’D’, ’E’, ’F’)
```

This difference arises because the output stream in the second example does not use a stretchy vector to hold the stream data. A vector of at least 15 elements is necessary to accommodate the elements written to the stream, but the vector supplied, **v**, can hold only 5. Since the stream cannot change **v**’s size, it must allocate a new vector each time it grows.

5.5 Stream classes

The exported streams class heterarchy includes the classes shown in Figure 5.1. Classes shown in bold are all instantiable.

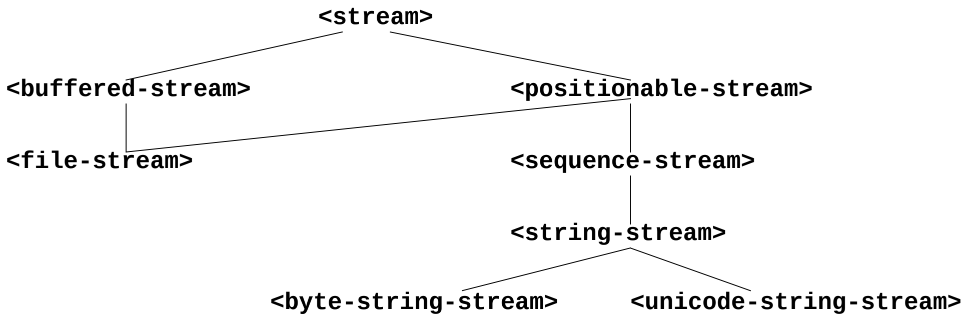

Figure 5.1 Streams module classes.

<stream> *Open abstract class*

The superclass of all stream classes and a direct subclass of **<object>**.

<positionable-stream> *Open abstract class*

A subclass of **`<stream>`** supporting the Positionable Stream Protocol.

<buffered-stream> *Open abstract class*

A subclass of **`<stream>`** supporting the Stream Extension Protocol and the Buffer Access Protocol.

Buffered streams support the **`buffer-size:`** init-keyword, which can be used to suggest the size of the stream's buffer. However, the instantiated stream might not use this value: it is taken purely as a suggested value.

<file-stream> *Open abstract instantiable class*

The class of single-buffered streams over disk files. The class supports several init-keywords: **`locator:`**, **`direction:`**, **`if-exists:`**, and **`if-does-not-exist:`**.

When you instantiate this class, an indirect instance of it is created. The file being streamed over is opened immediately upon creating the stream.

<sequence-stream> *Open class*

The class of streams over sequences. The class supports several init-keywords: **`contents:`**, **`direction:`**, **`start:`**, and **`end:`**.

This class can be used for streaming over all sequences, but there are also subclasses that are specialized for streaming over strings: see **`<string-stream>`**, page 102, **`<byte-string-stream>`**, page 63, and **`<unicode-string-stream>`**, page 105 for full details.

5.5.1 Creating streams

This section describes how to create and manage different types of file stream and sequence stream.

5.5.1.1 File streams

File streams are intended only for accessing the contents of files. More general file handling facilities, such as renaming, deleting, moving, and parsing directory names, are provided by the File-System module: see Chapter 8, "The File-System Module" for details. The make method on **`<file-stream>`** does not create direct instances of **`<file-stream>`**, but instead an instance of a subclass determined by **`type-for-file-stream`**.

make *file-stream-class* *G.f method*

```
make <file-stream> #key locator: direction: if-exists:
      if-does-not-exist: buffer-size: element-type:
      asynchronous?: share-mode => file-stream-instance
```

Creates and opens a stream over a file, and returns a new instance of a concrete subclass of **`<file-stream>`** that streams over the contents of the file referenced by *filename*. To determine the concrete subclass to be instantiated, this method calls the generic function **`type-for-file-stream`**.

The **`locator:`** init-keyword should be a string naming a file. If the Locators library is in use, *filename* should be an instance of **`<locator>`** or a string that can be coerced to one.

The **`direction:`** init-keyword specifies the direction of the stream. This can be one of **`#"input"`**, **`#"output"`**, or **`#"input-output"`**. The default is **`#"input"`**.

The **`if-exists:`** and **`if-does-not-exist:`** init-keywords specify actions to take if the file named by *filename* does or does not already exist when the stream is created. These init-keywords are discussed in more detail in Section 5.5.1.2 on page 38.

The **`buffer-size:`** init-keyword can be used to suggest the size of a stream's buffer. See **`<buffered-stream>`**, page 62.

The **`element-type:`** init-keyword specifies the type of the elements in the file named by *filename*. See Section 5.5.1.2 on page 38 for more details.

5.5.1.2 Options when creating file streams

When creating file streams, you can supply the following init-keywords to **make** in addition to those described in Section 5.5.1.1 on page 37:

if-exists: An action to take if the file already exists.

if-does-not-exist:

An action to take if the file does not already exist.

element-type: How the elements of the underlying file are accessed.

asynchronous?: Allows asynchronous writing of stream data to disk.

share-mode: How the file can be accessed while the stream is operating on it.

The **if-exists:** init-keyword allows you to specify an action to take if the file named by *filename* already exists. The options are:

#f The file is opened with the stream position at the beginning. This is the default when the stream's direction is **#"input"** or **#"input-output"**.

#"new-version"

If the underlying file system supports file versioning, a new version of the file is created. This is the default when the stream's direction is **#"output"**.

If the file system does not support file versioning, the default is **#"replace"** when the direction of the stream is **#"output"**.

#"overwrite" Set the stream's position to the beginning of the file, but preserve the current contents of the file. This is useful when the direction is **#"input-output"** or **#"output"** and you want to overwrite an existing file.

#"replace" Delete the existing file and create a new file.

#"append" Set the stream's initial position to the end of the existing file so that all new output occurs at the end of the file. This option is only useful if the file is writeable.

#"truncate" If the file exists, it is truncated, setting the size of the file to 0. If the file does not exist, create a new file.

#"signal" Signal a **<file-exists-error>** condition.

The **if-does-not-exist:** init-keyword allows you to specify an action to take if the file named by *filename* does not exist. The options are:

#f No action.

#"signal" Signal a **<file-does-not-exist-error>** condition. This is the default when the stream's direction is **#"input"**.

#"create" Create a new zero-length file. This is the default when the stream's direction is **#"output"** or **#"input-output"**.

Because creating a file stream *always* involves an attempt to open the underlying file, the aforementioned error conditions will occur during file stream instance initialization.

File permissions are checked when creating and opening file streams, and if the user attempts to open a file for input, and has no read permission, or to open a file for output, and has no write permission, then an **<invalid-file-permissions-error>** condition is signalled at the time the file stream is created.

The **element-type:** init-keyword controls how the elements of the underlying file are accessed. This allows file elements to be represented abstractly; for instance, contiguous elements could be treated as a single database record. The three possible element types are:

<byte-character>

The file is accessed as a sequence of 8-bit characters.

<unicode-character>

The file is accessed as a sequence of 16-bit Unicode characters.

<byte> The file is accessed as a sequence of unsigned 8-bit integers.

The **asynchronous?:** init-keyword allows asynchronous writing of stream data to disk. If **#f**, whenever the stream has to write a buffer to disk, the thread which triggered the write must wait for the write to complete. If **asynchronous?** is **#t**, the write proceeds in parallel with the subsequent actions of the thread.

Note that asynchronous writes complicate error handling a bit. Any write error which occurs most likely occurs after the call which triggered the write. If this happens, the error is stored in a queue, and the next operation on that stream signals the error. If you **close** the stream with the *wait?* flag **#f**, the close happens asynchronously (after all queued writes complete) and errors may occur after **close** has returned. A method **wait-for-io-completion** is provided to catch any errors that may occur after **close** is called.

The **share-mode:** keyword determines how a file can be accessed by other streams while the stream has it open. The possible values are:

#"share-read" Allow other streams to be opened to the file for reading but not for writing.

#"share-write" Allow other streams to be opened for writing but not for reading.

#"share-read-write"

Allow other streams to be opened for writing or reading.

#"exclusive" Do not allow other streams to be opened to this file.

5.5.1.3 Sequence streams

There are **make** methods on the following stream classes:

- **<sequence-stream>**
- **<string-stream>**
- **<byte-string-stream>**
- **<unicode-string-stream>**

Rather than creating direct instances of **`<sequence-stream>`** or **`<string-stream>`**, the **`make`** methods for those classes might create an instance of a subclass determined by **`type-for-sequence-stream`**.

make *sequence-stream-class* — *G.f. method*

```
make <sequence-stream> #key contents direction start end
=> sequence-stream-instance
```

Creates and opens a stream over a sequence, and returns a general instance of **`<sequence-stream>`**. To determine the concrete subclass to be instantiated, this method calls the generic function **`type-for-sequence-stream`**.

The **`contents:`** init-keyword is a general instance of **`<sequence>`** which is used as the input for an input stream, and as the initial storage for an output stream. If *contents* is a stretchy sequence (such as an instance of **`<stretchy-vector>`**), then it is the only storage used by the stream.

The **`direction:`** init-keyword specifies the direction of the stream. It must be one of **`#"input"`**, **`#"output"`**, or **`#"input-output"`**; the default is **`#"input"`**.

The **`start:`** and **`end:`** init-keywords are only valid when **`direction:`** is **`#"input"`**. They specify the portion of the sequence to create the stream over: **`start:`** is inclusive and **`end:`** is exclusive. The default is to stream over the entire sequence: **`start:`** is by default 0, and **`end:`** is **`contents.size`**.

make *string-stream-class* — *G.f. method*

```
make <string-stream> #key contents direction start end
=> string-stream-instance
```

Creates and opens a stream over a string, and returns an instance of **`<string-stream>`**.

If supplied, **`contents:`** must be an instance of **`<string>`**. The **`direction:`**, **`start:`**, and **`end:`** init-keywords are as for **`make`** on **`<sequence-stream>`**.

make *byte-string-stream-class* *G.f. method*

make <byte-string-stream #key contents direction start end
=> ***byte-string-stream-instance***

Creates and opens a stream over a byte string, and returns a new instance of **<byte-string-stream>**.

If supplied, **contents:** must be an instance of **<string>**. The **direction:**, **start:**, and **end:** init-keywords are as for **make** on **<sequence-stream>**.

make *unicode-string-stream-class* *G.f. method*

make <unicode-string-stream> #key contents direction start end
=> ***unicode-string-stream-instance***

Creates and opens a stream over a Unicode string, and returns a new instance of **<unicode-string-stream>**.

If supplied, **contents:** must be an instance of **<string>**. The **direction:**, **start:**, and **end:** init-keywords are as for **make** on **<sequence-stream>**.

5.5.2 Closing streams

It is important to call **close** on streams when you have finished with them. Typically, external streams such as **<file-stream>** and **<console-stream>** allocate underlying system resources when they are created, and these resources are not recovered until the stream is closed. The total number of such streams that can be open at one time may be system dependent. It may be possible to add reasonable finalization methods to close streams when they are no longer referenced but these are not added by default. See the *Core Features and Mathematics* manual for full details about finalization.

5.5.3 Locking streams

In an application where more than one control thread may access a common stream, it is important to match the granularity of locking to the transaction model of the application. Ideally, an application should lock a stream which is potentially accessed by multiple threads, only once per transaction. Repeated and unnecessary locking and unlocking can seriously degrade the performance of the Streams module. Thus an application which wishes to write a complex message to a stream that needs to be thread safe should lock the stream, write the message and then unlock the stream after the entire message is written. Locking and unlocking the stream for each character in the message would be a poor match of locking to transaction model. The time required for the lock manipulation would dominate the time required for the stream transactions. Unfortunately this means that there is no way for the Streams module to choose a default locking scheme without the likelihood of seriously degrading streams performance for all applications whose transaction models are different from the model implied by the chosen default locking scheme. Instead, the Streams module provides the user with a single, per instance slot, **`stream-lock:`**, which is inherited by all subclasses of **`<stream>`**. You should use the generic functions **`stream-lock`** and **`stream-lock-setter`**, together with other appropriate functions and macros from the Threads library, to implement a locking strategy appropriate to your application and its stream transaction model. The functions in the Streams module are not of themselves thread safe, and make no guarantees about the atomicity of read and write operations.

stream-lock *Open generic function*

`stream-lock` ***stream*** **`=>`** ***lock***

Returns the *lock* for the specified *stream*, or **`#f`** if no lock has been set. The *lock* argument is of type **`<lock>`**.

stream-lock-setter *Open generic function*

`stream-lock-setter` ***stream lock*** **`=>`** ***lock***

Sets the *lock* for the specified *stream*. The *lock* argument is of type **`<lock>`**, or **`#f`**. If *lock* is **`#f`**, the lock for *stream* is freed.

For full details on the **`<lock>`** class, see the documentation on the Threads library in the *Core Features and Mathematics* manual.

5.6 Reading from and writing to streams

This section describes how you can read from or write to a stream. Note that it is an error to call any of these functions on a buffered stream while its buffer is held by another thread; see Section 5.7 on page 51 for details about buffered streams.

5.6.1 Reading from streams

The following are the basic functions for reading from streams.

read-element *Open generic function*

`read-element` ***input-stream*** **`#key`** ***on-end-of-stream*** **`=>`** ***element-or-eof***

Returns the next element in *input-stream*. If the stream is not at its end, the stream is advanced in preparation for a subsequent read operation.

The *on-end-of-stream* keyword allows you to specify a value to be returned if the stream is at its end. If this is not supplied, **`read-element`** signals an **`<end-of-stream-error>`** condition on reading the end of the stream.

If no input is available and the stream is not at its end, **`read-element`** waits until input becomes available.

See also **`unread-element`**, page 58.

read *Open generic function*

`read` ***input-stream n*** **`#key`** ***on-end-of-stream*** **`=>`** ***sequence-or-eof***

Returns a sequence of the next *n* elements from *input-stream*.

The type of the sequence returned depends on the type of the stream's underlying aggregate. For instances of **`<sequence-stream>`**, the type of the result is given by **`type-for-copy`** of the underlying aggregate. For instances of **`<file-stream>`**, the result is a vector that can contain elements of the type returned by calling **`stream-element-type`** on the stream.

The stream position is advanced so that the next call to any function that reads from or writes to *input-stream* acts on the stream position immediately following the last of the *n* elements read.

If the stream is not at its end, **`read`** waits until input becomes available.

If the end of the stream is reached before all *n* elements have been read, the behavior is as follows.

If *on-end-of-stream* was supplied, it is returned as the value of **`read`**.

If *on-end-of-stream* argument was not supplied, and at least one element was read from the stream, then an **`<incomplete-read-error>`** condition is signalled. When signalling this condition, **`read`** supplies two values: a sequence of the elements that were read successfully, and *n*.

If *on-end-of-stream* was not supplied, and no elements were read from the stream, an **`<end-of-stream-error>`** condition is signalled.

The second of these is in some sense the most general behavior, in that the first and third cases could, in principle, be duplicated by using the second case, handling the signalled **`<incomplete-read-error>`**, and returning appropriate results.

A number of other functions are available for reading from streams. See **`peek`**, page 81, **`read-into!`**, page 85, **`discard-input`**, page 66, and **`stream-input-available?`**, page 96.

5.6.2 Convenience functions for reading from streams

The following is a small set of reading functions that search for particular elements in a stream. These functions behave as though they were implemented in terms of the more primitive functions described in Section 5.6.1.

read-to *Function*

`read-to` ***input-stream element*** **`#key`** ***on-end-of-stream test***
`=>` ***sequence-or-eof found?***

Returns a new sequence containing the elements of *input-stream* from the stream's current position to the first occurrence of *element*, but not *element* itself.

found? is `#t` if the read terminated with *element*, or `#f` if the read terminated by reaching the end of the stream's source. The stream is left positioned after *element*.

See also **`read-through`**, page 88.

read-to-end *Function*

`read-to-end` ***input-stream*** **`=>`** ***sequence***

Returns a sequence of all the elements up to, and including, the last element of *input-stream*, starting from the stream's current position.

skip-through *Function*

`skip-through` ***input-stream element*** `#key` ***test*** **`=>`** ***found?***

Positions *input-stream* after the first occurrence of *element*, starting from the stream's current position. Returns `#t` if the element was found, or `#f` if the end of the stream was encountered. When **`skip-through`** does not find the *element*, it leaves *input-stream* positioned at the end of the stream.

5.6.3 Writing to streams

This section describes the basic functions for writing to streams.

write-element *Open generic function*

`write-element` ***output-stream element*** **`=> ()`**

Writes *element* to *output-stream* at the stream's current position. It is an error if the type of *element* is inappropriate for the stream's underlying aggregate.

If the stream is positionable, and it is not positioned at its end, **`write-element`** overwrites the element at the current position and then advance the stream position.

write *Open generic function*

`write` ***output-stream sequence*** **`#key`** ***start end*** **`=> ()`**

Writes the elements of *sequence* to *output-stream*, starting at the stream's current position.

If supplied, *start* and *end* delimit the portion of *sequence* to write to the stream. The value of *start* is inclusive and that of *end* is exclusive. If *start* and *end* are not supplied, the whole sequence is written.

For positionable streams, if the initial position of the stream is such that writing *sequence* will flow past the current end of the stream, then the stream is extended to accommodate the extra elements. Once the write operation has finished, the stream is positioned one place past the last element written.

See **`force-output`**, page 71, **`synchronize-output`**, page 103, and **`discard-output`**, page 66.

5.6.4 Reading and writing by lines

The following functions provide line-based input and output operations.

The newline sequence for string streams is a sequence comprising the single newline character **`\n`**. For character file streams, the newline sequence is whatever sequence of characters the underlying platform uses to represent a newline. For example, on MS-DOS platforms, the sequence comprises two characters: a carriage return followed by a linefeed.

Note: No other functions in the Streams module do anything to manage the encoding of newlines; calling **`write-element`** on the character **`\n`** does not cause the **`\n`** character to be written as the native newline sequence, unless **`\n`** happens to *be* the native newline sequence.

read-line *Open generic function*

`read-line` ***input-stream*** **`#key`** ***on-end-of-stream*** **`=>`** ***string-or-eof newline?***

Returns a newly allocated **`<string>`** containing all the input in *input-stream* up to the next newline. The string does not contain the newline itself.

newline? is **`#t`** if the read terminated with a newline or **`#f`** if the read terminated because it came to the end of the stream.

The type of the result string is chosen so that the string can contain characters of *input-stream*'s element type. For example, if the element type is **`<byte-character>`**, the string will be a **`<byte-string>`**.

write-line *Open generic function*

`write-line` ***output-stream string*** **`#key`** ***start end*** **`=> ()`**

Writes *string* followed by a newline sequence to *output-stream*.

The default method behaves as though it calls **`write`** on *string* and then calls **`new-line`**, with *output-stream* locked across both calls.

If supplied, *start* and *end* delimit the portion of *string* to write to the stream. They default to 0 and *string***`.size`** respectively.

new-line *Open generic function*

`new-line` ***output-stream*** **`=> ()`**

Writes a newline sequence to *output-stream*.

A method for **`new-line`** is defined on **`<string-stream>`** that writes the character **`\n`** to the string stream.

See also **`read-line-into!`**, page 87.

5.6.5 Querying streams

The following functions can be used to determine various properties of a stream.

stream-open? *Open generic function*

`stream-open?` ***stream*** **`=>`** ***open?***

Returns **`#f`** if the stream has been closed, and **`#t`** otherwise. Note that an input stream which is at its end, but has not been closed, is still open and **`stream-open?`** will return #t.

stream-element-type *Open generic function*

`stream-element-type` ***stream*** **`=>`** ***element-type***

Returns the element type of *stream* as a Dylan **`<type>`**.

stream-at-end? *Open generic function*

`stream-at-end?` ***stream*** **`=>`** ***boolean***

Returns **`#t`** if the stream is at its end and **`#f`** if it is not. For input streams, it returns **`#t`** if a call to **`read-element`** with no supplied keyword arguments would signal an **`<end-of-stream-error>`**. For output streams, this function always returns **`#f`**.

For output streams, note that you can determine if a stream is one place past the last written element by comparing **`stream-position`** to **`stream-size`**.

5.6.6 Using file streams

The following operations can be performed on file streams.

close *G.f. method*

`close` ***file-stream*** **`#key`** ***abort wait?*** **`=> ()`**

Closes a file stream. If the stream is asynchronous and *wait?* is false (its default value is **`#t`**), then a close request is merely enqueued to be performed after all pending write operations; otherwise the file is closed immediately and all underlying system resources held on behalf of the stream are freed.

If *abort?*. is false (the default) all buffered data is written before closing; if *abort?* false, this data is discarded.

If *synchronize?* (default value **`#f`**) is true, the file is flushed to the physical disk before closing — this guarantees that no data is retained in the operating system's write cache. Calling **`close`** with *synchronize?* **`#t`** is equivalent to calling **`force-output`** with *synchronize?* true and then calling **`close`**.

wait-for-io-completion *Statement macro*

wait-for-io-completion ***file-stream*** **=> ()**

If *file-stream* is asynchronous, waits for all pending **write** or **close** operations to complete and signals any queued errors. If *file-stream* is not asynchronous, returns immediately.

with-open-file *Statement macro*

with-open-file (*stream-var* = *filename*, #rest *keys*) *body* end => *values*

This macro provides a safe mechanism for working with file streams. It creates a file stream and binds it to *stream-var*, evaluates a *body* of code within the context of this binding, and then closes the stream. The macro calls **close** upon exiting *body*.

The values of the last expression in *body* are returned.

The *keys* are passed to the **make** method on **<file-stream>**.

For example, the following expression yields the contents of file **foo.text** as a **<byte-vector>**:

```
with-open-file (fs = "foo.text", element-type: <byte>)
  read-to-end(fs)
end;
```

It is roughly equivalent to:

```
begin
  let hidden-fs = #f;     // In case the user bashes fs variable
  block ()
    hidden-fs := make(<file-stream>,
                    locator: "foo.text", element-type: <byte>);
    let fs = hidden-fs;
    read-to-end(fs);
  cleanup
    if (hidden-fs) close(hidden-fs) end;
  end block;
end;
```

5.7 Using buffered streams

The Streams module provides efficient support for general use of buffered I/O. Most ordinary programmers using the module do not need to be concerned with buffering in most cases. When using buffered streams, the buffering is transparent, but programs requiring more control can access buffering functionality when appropriate. This section describes the available buffering functionality.

5.7.1 Overview

A buffered stream maintains some sort of buffer. All buffered streams use the sealed class **`<buffer>`** for their buffers. You can suggest a buffer size when creating buffered streams, but normally you do not need to do so, because a buffer size that is appropriate for the stream's source or destination is chosen for you.

Instances of the class **`<buffer>`** also contain some state information. This state information includes an index where reading or writing should begin, and an index that is the end of input to be read, or the end of space available for writing.

Buffered streams also maintain a *held* state, indicating whether the application has taken the buffer for a stream and has not released it yet. When a thread already holds the buffer for a stream, it is an error to get the buffer again (or any other buffer for the same stream).

5.7.2 Useful types when using buffers

The following types are used in operations that involve buffers.

<byte> *Type*

A type representing limited integers in the range 0 to 255 inclusive.

<byte-character> *Type*

A type representing 8-bit characters that instances of **`<byte-string>`** can contain.

<unicode-character> *Type*

A type representing Unicode characters that instances of **`<unicode-string>`** can contain.

<byte-vector> *Type*

A subtype of **`<vector>`** whose element-type is **`<byte>`**.

5.8 Wrapper streams

Sometimes stream data requires conversion before an application can use it: you might have a stream over a file of EBCDIC characters which you would prefer to handle as their ASCII equivalents, or you might need to encrypt or decrypt file data.

Wrapper streams provide a mechanism for working with streams which require such conversion. Wrapper streams hold on to an underlying stream, delegating to it most streams operations. The wrapper stream carries out appropriate processing in its own implementations of the streaming protocol.

The Streams module includes a base class called **`<wrapper-stream>`** upon which other wrapping streams can be implemented.

A subclass of **`<wrapper-stream>`** can "pass on" functions such as **`read-element`** and **`write-element`** by simply delegating these operations to the inner stream, as shown below:

```
define method read-element (ws :: <io-wrapper-stream>,
                            #key on-end-of-stream)
    => (element)
  read-element(ws.inner-stream,
               on-end-of-stream: on-end-of-stream)
end method;

define method write-element (ws :: <io-wrapper-stream>, element)
    => ()
  write-element(ws.inner-stream, element)
end method;
```

Assuming that **`<io-wrapper-stream>`** delegates all other operations to its inner stream, the following would suffice to implement a 16-bit Unicode character stream wrapping an 8-bit character stream.

```
define class <unicode-stream> (<io-wrapper-stream>) end class;

define method read-element (s :: <unicode-stream>,
                            #key on-end-of-stream)
    => (ch :: <unicode-character>)
  with-stream-locked (s)
    let first-char =  read-element(s.inner-stream,
                                   on-end-of-stream);
    let second-char = read-element(s.inner-stream,
                                   on-end-of-stream)
  end;
  convert-byte-pair-to-unicode(first-char, second-char)
end method;

define method write-element  (s :: <unicode-stream>,
                              c :: <character>)
    => ()
  let (first-char, second-char) =
        convert-unicode-to-byte-pair(c);
  with-stream-locked (s)
    write-element(s.inner-stream, first-char);
    write-element(s.inner-stream, second-char)
  end;
  c
end method;

define method stream-position (s :: <unicode-stream>)
  =>  p :: <integer>;
  truncate/(stream-position(s.inner-stream), 2)
end method;

define method stream-position-setter  (p :: <integer>,
                                       s :: <unicode-stream>);
  stream-position(s.inner-stream) := p * 2
end method;
```

5.8.1 Wrapper streams and delegation

One problem with wrapper streams is the need for a wrapper stream to intercept methods invoked by its inner stream. For example, consider two hypothetical streams, **<interactive-stream>** and **<dialog-stream>**, the latter a subclass of **<wrapper-stream>**. Both of these classes have a method called **prompt**. The **<interactive-stream>** class specializes **read** thus:

```
define method read (s :: <interactive-stream>,
                    n :: <integer>,
                    #key on-end-of-stream);
  prompt(s);
  next-method()
end method;
```

If a **<dialog-stream>** is used to wrap an **<interactive-stream>** then an invocation of **read** on the **<dialog-stream>** will call **prompt** on the inner **<interactive-stream>**, not on the **<dialog-stream>**, as desired. The problem is that the **<dialog-stream>** delegates some tasks to its inner stream, but handles some other tasks itself.

Delegation by inner-streams to outer-streams is implemented by the use of the **outer-stream** function. The **outer-stream** function is used instead of the stream itself whenever a stream invokes one of its other protocol methods.

A correct implementation of the **read** method in the example above would be as follows:

```
define method read (stream :: <interactive-stream>,
                    n :: <integer>,
                    #key on-end-of-stream)
  prompt(s.outer-stream);
  next-method()
end method;
```

The **initialize** method on **<stream>** is defined to set the **outer-stream** slot to be the stream itself. The **initialize** method on **<wrapper-stream>** is specialized to set the **outer-stream** slot to be the “parent” stream:

```
define method initialize (stream :: <wrapper-stream>,
                          #key on, #rest all-keys);
  an-inner-stream.outer-stream := stream;
  next-method()
end method;
```

5.9 Conditions

The following classes are available for error conditions on streams.

```
<end-of-stream-error>
<incomplete-read-error>
<file-error>
<file-exists-error>
<file-does-not-exist-error>
<invalid-file-permissions-error>
```

There is no recovery protocol defined for any of these errors. Every condition that takes an init-keyword has a slot accessor for the value supplied. The name of this accessor function takes the form *class*-*key*, where *class* is the name of the condition class (without the angle brackets) and *key* is the name of the init-keyword. For example, the accessor function for the **locator:** init-keyword for **<file-error>** is **file-error-locator**.

For more information, please refer to the reference entry for the individual conditions.

5.10 Streams protocols

This section describes the protocols for different classes of stream.

5.10.1 Positionable stream protocol

This section describes the protocol for positionable streams.

A stream position can be thought of as a natural number that indicates how many elements into the stream the stream’s current location is. However, it is not always the case that a single integer contains enough information to reposition a stream. Consider the case of an “uncompressing” file stream that requires additional state beyond simply the file position to be able to get the next input character from the compressed file.

The Streams module addresses this problem by introducing the class **`<stream-position>`**, which is subclassed by various kinds of stream implementations that need to maintain additional state. A stream can be repositioned as efficiently as possible when **`stream-position-setter`** is given a value previously returned by **`stream-position`** on that stream.

It is also legal to set the position of a stream to an integer position. However, for some types of streams, to do so might be slow, perhaps requiring the entire contents of the stream up to that point to be read.

<position-type> *Type*

`type-union(<stream-position>, <integer>)`

A type used to represent a position in a stream. In practice, positions within a stream are defined as instances of **`<integer>`**, but this type, together with the **`<stream-position>`** class, allows for cases where this might not be possible.

<stream-position> *Abstract class*

A direct subclass of **`<object>`**. It is used in rare cases to represent positions within streams that cannot be represented as instances of **`<integer>`**, such as a stream that supports compression.

stream-position *Open generic function*

`stream-position` ***positionable-stream*** **`=>`** ***position***

Returns the current position of *positionable-stream* for reading or writing.

stream-position-setter *Open generic function*

`stream-position-setter` ***position positionable-stream*** **`=>`** ***new-position***

Changes the stream's position to *position*, for reading or writing.

The following are all possible values of *position*: an integer between 0 and *positionable-stream*`.`**`stream-size`**, a valid **`<stream-position>`**, **`#"start"`**, or **`#"end"`**.

Note: You cannot use **`stream-position-setter`** to set the position past the current last element of the stream: use **`adjust-stream-position`** instead.

adjust-stream-position *Open generic function*

`adjust-stream-position` ***positionable-stream delta*** **`#key`** ***from***
`=>` ***new-position***

Moves the position of *positionable-stream* to be offset *delta* elements from the position indicated by *from*. The new position is returned. The *delta* offset must be an instance of **`<integer>`**.

The value of *from* can be one of the symbols **`#"current"`**, **`#"start"`**, and **`#"end"`**. The default is **`#"current"`**.

Using **`adjust-stream-position`** to set the position of a stream to be beyond its current last element grows the underlying aggregate to a new size.

as *G.f. method*

`as` ***integer-class stream-position*** **`=>`** ***integer***

Coerces a **`<stream-position>`** to an integer. The *integer-class* argument is the class **`<integer>`**.

stream-size *Open generic function*

`stream-size` ***positionable-stream*** **`=>`** ***size***

Returns the number of elements in *positionable-stream*.

For input streams, this is the number of elements that were available when the stream was created. It is unaffected by any read operations that might have been performed on the stream.

For output and input-output streams, this is the number of elements that were available when the stream was created (just as with input streams), added to the number of elements written past the end of the stream (regardless of any repositioning operations).

stream-contents *Open generic function*

`stream-contents` ***positionable-stream*** **`#key`** ***clear-contents?*** **`=>`** ***sequence***

Returns a sequence that contains all of *positionable-stream*'s elements from its start to its end, regardless of its current position. The type of the returned sequence is as for **`read`**. See page 44.

The *clear-contents?* argument only applies to writeable sequence streams. If *clear-contents?* is **`#t`** (the default for streams to which it is applicable), this function sets the size of the stream to zero, and the position to the stream's start. Thus the next call to **`stream-contents`** will return only the elements written after the previous call to **`stream-contents`**. The *clear-contents?* argument is not defined for file streams, or any other external stream. It is also an error to apply it to input-only streams.

Note: You must use **`read-to-end`** for input streams.

unread-element *Open generic function*

`unread-element` ***positionable-stream element*** **`=>`** ***element***

Returns *element* to *positionable-stream* so that the next call to **`read-element`** returns *element*. It is an error if *element* was not the last element read from the stream. You may not call **`unread-element`** more than once without an intervening read operation (that is, you cannot unread more than one element at a time).

5.10.2 Wrapper stream protocol

This section describes the protocol for implementing wrapper streams. For information on using wrapper streams, see Section 5.8 on page 52.

<wrapper-stream> *Open instantiable class*

The class that implements the basic wrapper-stream functionality. A required init-keyword, **`inner-stream:`**, specifies the wrapped stream.

inner-stream *Open generic function*

`inner-stream` ***wrapper-stream*** **`=>`** ***wrapped-stream***

Returns the stream wrapped by *wrapper-stream*.

inner-stream-setter *Open generic function*

`inner-stream-setter` ***stream wrapper-stream*** **`=>`** ***stream***

Wraps *stream* with *wrapper-stream*. It does so by setting the **`inner-stream`** slot of *wrapper-stream* to *stream*, and the **`outer-stream`** slot of *stream* to *wrapper-stream*.

outer-stream *Open generic function*

`outer-stream` ***stream*** **`=>`** ***wrapping-stream***

Returns the stream that is wrapping *stream*.

outer-stream-setter *Open generic function*

`outer-stream-setter` ***wrapper-stream stream*** **`=>`** ***wrapper-stream***

Sets the **`outer-stream`** slot of *stream* to *wrapper-stream*.

5.11 The STREAMS module

This section includes complete reference entries for all interfaces that are exported from the **`streams`** module.

adjust-stream-position *Open generic function*

Summary

Moves the position of a positionable stream by a specified amount.

Signature

`adjust-stream-position` ***positionable-stream delta*** **`#key`** ***from*** **`=>`** ***new-position***

Arguments

positionable-stream	An instance of **<positionable-stream>**.
delta	An instance of **<integer>**.
from	One of **#"current"**, **#"start"**, or **#"end"**. Default value: **#"current"**.

Values

new-position	An instance of **<stream-position>**.

Description

Moves the position of *positionable-stream* to be offset *delta* elements from the position indicated by *from*. The new position is returned.

When *from* is **#"start"**, the stream is positioned relative to the beginning of the stream. When *from* is **#"end"**, the stream is positioned relative to its end. When *from* is **#"current"**, the current position is used.

Using **adjust-stream-position** to set the position of a stream to be beyond its current last element causes the underlying aggregate to be grown to a new size. When extending the underlying aggregate for a stream, the contents of the unwritten elements are the fill character for the underlying sequence.

Example

The following example returns a string, the first ten characters of which are the space character, which is the fill character for the sequence **<string>**.

```
let stream = make(<string-stream>,
                  direction: #"output");
adjust-stream-position(stream, 10);
write(stream, "whoa!");
stream-contents(stream);
```

See also

stream-position-setter, page 56

as *G.f. method*

Summary
: Coerces a **`<stream-position>`** to an integer.

Signature
: **`as`** ***integer-class stream-position*** **`=>`** ***integer***

Arguments

integer-class	The class **`<integer>`**.
stream-position	An instance of **`<stream-position>`**.

Values

integer	An instance of **`<integer>`**.

Description
: Coerces a **`<stream-position>`** to an integer. The *integer-class* argument is the class **`<integer>`**.

See also
: **`as`**, page 61

<buffer> *Sealed instantiable class*

Summary
: A subclass of **`<vector>`** whose **`element-type`** is **`<byte>`**.

Superclasses
: **`<vector>`**

Init-keywords

`size:`	An instance of **`<integer>`** specifying the size of the buffer. Default value: 0.
`next:`	An instance of **`<integer>`**. For an input buffer, this is where the next input byte can be found. For an output buffer, this is where the next output byte should be written to. Default value: 0.
`end:`	An instance of **`<integer>`**. The value of this is one more than the last valid index in a buffer. For an input buffer, this represents the number of bytes read.

Description
: A subclass of **`<vector>`** whose **`element-type`** is **`<byte>`**.

Instances of **`<buffer>`** contain a data vector and two indices: the inclusive start and the exclusive end of valid data in the buffer. The accessors for these indexes are called **`buffer-next`** and **`buffer-end`**.

Note that **`size:`** is not taken as a suggestion of the size the user would like, as with the value passed with **`buffer-size:`** to **`make`** on **`<buffered-stream>`**; if you supply a value with the **`size:`** init-keyword, that size is allocated, or, if that is not possible, an error is signalled, as with making any vector.

<buffered-stream> *Open abstract class*

Summary

A subclass of **`<stream>`** supporting the Stream Extension and Buffer Access protocols.

Superclasses

`<stream>`

Init-keywords

`buffer-size:` An instance of **`<integer>`**. This is the size of the buffer in bytes.

Description

A subclass of **`<stream>`** supporting the Stream Extension Protocol and the Buffer Access Protocol. It is not instantiable.

Streams of this class support the **`buffer-size:`** init-keyword, which can be used to suggest the size of the stream's buffer. However, the instantiated stream might not use this value: it is taken purely as a suggested value. For example, a stream that uses a specific device's hardware buffer might use a fixed buffer size regardless of the value passed with the **`buffer-size:`** init-keyword.

In general, it should not be necessary to supply a value for the **`buffer-size:`** init-keyword.

<byte> *Type*

Summary A type representing limited integers in the range 0 to 255 inclusive.

Supertypes **`<integer>`**

Init-keywords None.

Description A type representing limited integers in the range 0 to 255 inclusive.

Operations **`type-for-file-stream`**

<byte-character> *Type*

Summary A type representing 8-bit characters that instances of **`<byte-string>`** can contain.

Supertypes **`<character>`**

Init-keywords None.

Description A type representing 8-bit characters that instances of **`<byte-string>`** can contain.

Operations **`type-for-file-stream`**

<byte-string-stream> *Open instantiable class*

Summary The class of streams over byte strings.

Superclasses **`<string-stream>`**

Init-keywords **`contents:`** A general instance of **`<sequence>`**.

direction:	Specifies the direction of the stream. It must be one of **#"input"**, **#"output"**, or **#"input-output"**. Default value: **#"input"**.
start:	An instance of **<integer>**. This specifies the start position of the byte string to be streamed over. Only valid when **direction:** is **#"input"**. Default value: 0.
end:	An instance of **<integer>**. This specifies the sequence position immediately after the portion of the byte string to stream over. Only valid when **direction:** is **#"input"**. Default value: **contents.size**.

Description

The class of streams over byte strings. It is a subclass of **<string-stream>**.

The class supports the same init-keywords as **<sequence-stream>**.

The **contents:** init-keyword is used as the input for an input stream, and as the initial storage for an output stream.

The **start:** and **end:** init-keywords specify the portion of the byte string to create the stream over: **start:** is inclusive and **end:** is exclusive. The default is to stream over the entire byte string.

Operations

make byte-string-stream-class

See also

make byte-string-stream-class, page 74

<sequence-stream>, page 91

<byte-vector> *Sealed class*

Summary

A subtype of **<vector>** whose element-type is **<byte>**.

Superclasses `<vector>`

Init-keywords See Superclasses.

Description A subclass of `<vector>` whose element-type is `<byte>`.

Operations None.

See also `<byte>`, page 63

close *Open generic function*

Summary Closes a stream.

Signature **`close`** ***stream*** **`#key #all-keys => ()`**

Arguments *stream* An instance of `<stream>`.

Values None.

Description Closes *stream*, an instance of `<stream>`.

close *G.f. method*

Summary Closes a file stream.

Signature **`close`** ***file-stream*** **`#key`** ***abort? wait?*** **`=> ()`**

Arguments

file-stream	An instance of `<file-stream>`.
abort?	An instance of `<boolean>`. Default value: `#f`.
wait?	An instance of `<boolean>`.

Values None.

Description

Closes a file stream. This method frees whatever it can of any underlying system resources held on behalf of the stream.

If *abort* is false, any pending data is forced out and synchronized with the file's destination. If *abort* is true, then any errors caused by closing the file are ignored.

discard-input *Open generic function*

Summary

Discards input from an input stream.

Signature

`discard-input` ***input-stream*** **`=> ()`**

Arguments

input-stream An instance of **`<stream>`**.

Values

None.

Description

Discards any pending input from *input-stream*, both buffered input and, if possible, any input that might be at the stream's source.

This operation is principally useful for "interactive" streams, such as TTY streams, to discard unwanted input after an error condition arises. There is a default method on **`<stream>`** so that applications can call this function on any kind of stream. The default method does nothing.

See also

`discard-output`, page 66

discard-output *Open generic function*

Summary

Discards output to an output stream.

Signature

`discard-output` ***output-stream*** **`=> ()`**

Arguments

output-stream An instance of **`<stream>`**.

Values None.

Description Attempts to abort any pending output for *output-stream*.

A default method on **`<stream>`** is defined, so that applications can call this function on any sort of stream. The default method does nothing.

See also **`discard-input`**, page 66

<end-of-stream-error> *Error*

Summary Error type signaled on reaching the end of an input stream.

Superclasses **`<error>`**

Init-keywords **`stream:`** An instance of **`<stream>`**.

Description Signalled when one of the read functions reaches the end of an input stream. It is a subclass of **`<error>`**.

The **`stream:`** init-keyword has the value of the stream that caused the error to be signaled. Its accessor is **`end-of-stream-error-stream`**.

Operations None.

See also **`<file-does-not-exist-error>`**, page 68

`<file-error>`, page 68

`<file-exists-error>`, page 69

`<incomplete-read-error>`, page 71

`<invalid-file-permissions-error>`, page 73

<file-does-not-exist-error> *Error*

Summary
Error type signaled when attempting to read a file that does not exist.

Superclasses
`<file-error>`

Init-keywords
See Superclasses.

Description
Signaled when an input file stream creation function tries to read a file that does not exist. It is a subclass of **`<file-error>`**.

Operations
None.

See also
`<end-of-stream-error>`, page 67

`<file-error>`, page 68

`<file-exists-error>`, page 69

`<incomplete-read-error>`, page 71

`<invalid-file-permissions-error>`, page 73

<file-error> *Error*

Summary
The base class for all errors related to file I/O.

Superclasses
`<error>`

Init-keywords
`locator:` An instance of **`<locator>`**.

Description
The base class for all errors related to file I/O. It is a subclass of **`<error>`**.

The **`locator:`** init-keyword indicates the locator of the file that caused the error to be signalled. Its accessor is **`file-error-locator`**.

Operations None.

See also **`<end-of-stream-error>`**, page 67

`<file-does-not-exist-error>`, page 68

`<file-exists-error>`, page 69

`<incomplete-read-error>`, page 71

`<invalid-file-permissions-error>`, page 73

<file-exists-error> *Error*

Summary Error type signaled when trying to create a file that already exists.

Superclasses **`<file-error>`**

Init-keywords See Superclasses.

Description Signalled when an output file stream creation function tries to create a file that already exists. It is a subclass of **`<file-error>`**.

Operations None.

See also **`<end-of-stream-error>`**, page 67

`<file-does-not-exist-error>`, page 68

`<file-error>`, page 68

`<incomplete-read-error>`, page 71

`<invalid-file-permissions-error>`, page 73

<file-stream> *Open abstract instantiable class*

Summary

The class of single-buffered streams over disk files.

Superclasses

<buffered-stream> <positionable-stream>

Init-keywords

locator: An instance of **<string>** or **<locator>**. This specifies the file over which to stream.

direction: Specifies the direction of the stream. It must be one of **#"input"**, **#"output"**, or **#"input-output"**. Default value: **#"input"**.

if-exists: One of **#f**, **#"new-version"**, **#"overwrite"**, **#"replace"**, **#"append"**, **#"truncate"**, **#"signal"**. Default value: **#f**.

if-does-not-exist:

One of **#f**, **#"signal"**, or **#"create"**. Default value: depends on the value of **direction:**.

asynchronous?: If **#t,** all writes on this stream are performed asynchronously. Default value: **#f**.

Description

The class of single-buffered streams over disk files. It is a subclass of **<positionable-stream>** and **<buffered-stream>**.

When you instantiate this class, an indirect instance of it is created. The file being streamed over is opened immediately upon creating the stream.

The class supports several init-keywords: **locator:**, **direction:**, **if-exists:**, and **if-does-not-exist:**.

Operations

close make file-stream-class

See also

make file-stream-class, page 75

force-output *Open generic function*

Summary
Forces pending output from an output stream buffer to its destination.

Signature
`force-output` ***output-stream*** **`#key`** ***synchroniz?*** **`e=> ()`**

Arguments

output-stream	An instance of **`<stream>`**.
synchronize?	An instance of **`<boolean>`**. Default value: **`#f`**.

Values
None.

Description
Forces any pending output from *output-stream*'s buffers to its destination. Even if the stream is asynchronous, this call waits for all writes to complete. If *synchronize?* is true, also flushes the operating system's write cache for the file so that all data is physically written to disk. This should only be needed if you're concerned about system failure causing loss of data.

See also
`synchronize-output`, page 103

<incomplete-read-error> *Error*

Summary
Error type signaled on encountering the end of a stream before reading the required number of elements.

Superclasses
`<end-of-stream-error>`

Init-keywords

`sequence:`	An instance of **`<sequence>`**.
`count:`	An instance of **`<integer>`**.

Description
This error is signaled when input functions are reading a required number of elements, but the end of the stream is read before completing the required read.

The **sequence:** init-keyword contains the input that was read before reaching the end of the stream. Its accessor is **incomplete-read-error-sequence**.

The **count:** init-keyword contains the number of elements that were requested to be read. Its accessor is **incomplete-read-error-count**.

Operations

None.

See also

<end-of-stream-error>, page 67

<file-does-not-exist-error>, page 68

<file-error>, page 68

<file-exists-error>, page 69

<invalid-file-permissions-error>, page 73

inner-stream *Open generic function*

Summary

Returns the stream being wrapped.

Signature

inner-stream ***wrapper-stream*** **=>** ***wrapped-stream***

Arguments

wrapper-stream An instance of **<wrapper-stream>**.

Values

wrapped-stream An instance of **<stream>**.

Description

Returns the stream wrapped by *wrapper-stream*.

See also

inner-stream-setter, page 73

outer-stream, page 80

<wrapper-stream>, page 109

inner-stream-setter *Open generic function*

Summary

Wraps a stream with a wrapper stream.

Signature

`inner-stream-setter` ***stream wrapper-stream*** **`=>`** ***stream***

Arguments

stream	An instance of **`<stream>`**.
wrapper-stream	An instance of **`<wrapper-stream>`**.

Values

stream	An instance of **`<stream>`**.

Description

Wraps *stream* with *wrapper-stream*. It does so by setting the **`inner-stream`** slot of *wrapper-stream* to *stream*, and the **`outer-stream`** slot of *stream* to *wrapper-stream*.

Note: Applications should not set **`inner-stream`** and **`outer-stream`** slots directly. The **`inner-stream-setter`** function is for use only when implementing stream classes.

See also

`inner-stream`, page 72

`outer-stream-setter`, page 81

<invalid-file-permissions-error> *Error*

Summary

Error type signalled when accessing a file in a way that conflicts with the permissions of the file.

Superclasses

`<file-error>`

Init-keywords

See Superclasses.

Description

Signalled when one of the file stream creation functions tries to access a file in a manner for which the user does not have permission. It is a subclass of **`<file-error>`**.

Operations

None.

See also

<end-of-stream-error>, page 67

<file-does-not-exist-error>, page 68

<file-error>, page 68

<file-exists-error>, page 69

<incomplete-read-error>, page 71

make *byte-string-stream-class* *G.f. method*

Summary

Creates and opens a stream over a byte string.

Signature

make ***byte-string-stream-class*** **#key** ***contents direction start end***
=> ***byte-string-stream-instance***

Arguments

byte-string-stream-class

The class **<byte-string-stream>**.

contents An instance of **<byte-string>**.

direction One of **#"input"**, **#"output"**, or **#"input-output"**. Default value: **#"input"**.

start An instance of **<integer>**. Default value: 0.

end An instance of **<integer>**. Default value: *contents***.size**.

Values

byte-string-stream-instance

An instance of **<byte-string-stream>**.

Description

Creates and opens a stream over a byte string.

This method returns a new instance of **<byte-string-stream>**.

If supplied, *contents* describes the contents of the stream, and must be an instance of **<byte-string>**. The *direction*, *start*, and *end* init-keywords are as for **make** on **<sequence-stream>**.

Example

```
let stream = make(<byte-string-stream>,
                  direction: #"output");
```

See also

`<byte-string-stream>`, page 63

`make sequence-stream-class`, page 76

make *file-stream-class* — *G.f method*

Summary

Creates and opens a stream over a file.

Signature

`make` ***file-stream-class*** **`#key`** ***filename direction***
if-exists if-does-not-exist
buffer-size element-type
`=>` ***file-stream-instance***

Arguments

file-stream-class	The class **`<file-stream>`**.
filename	An instance of **`<object>`**.
direction	One of **`#"input"`**, **`#"output"`**, or **`#"input-output"`**. The default is **`#"input"`**.
if-exists	One of **`#f`**, **`#"new-version"`**, **`#"overwrite"`**, **`#"replace"`**, **`#"append"`**, **`#"truncate"`**, **`#"signal"`**. Default value: **`#f`**.
if-does-not-exist	One of **`#f`**, **`#"signal"`**, or **`#"create"`**. Default value: depends on the value of *direction*.
buffer-size	An instance of **`<integer>`**.
element-type	One of **`<byte-character>`**, **`<unicode-character>`**, or **`<byte>`**, or **`#f`**.

Values

file-stream-instance

An instance of **`<file-stream>`**.

Description

Creates and opens a stream over a file.

Returns a new instance of a concrete subclass of **`<file-stream>`** that streams over the contents of the file referenced by *filename*. To determine the concrete subclass to be instantiated, this method calls the generic function **`type-for-file-stream`**.

The *filename* init-keyword should be a string naming a file. If the Locators library is in use, *filename* should be an instance of **`<locator>`** or a string that can be coerced to one.

The *direction* init-keyword specifies the direction of the stream.

The *if-exists* and *if-does-not-exist* init-keywords specify actions to take if the file named by *filename* does or does not already exist when the stream is created. These init-keywords are discussed in more detail in Section 5.5.1.2 on page 38.

The *buffer-size* init-keyword is explained in **`<buffered-stream>`**, page 36.

The *element-type* init-keyword specifies the type of the elements in the file named by *filename*. This allows file elements to be represented abstractly; for instance, contiguous elements could be treated as a single database record. This init-keyword defaults to something useful, potentially based on the properties of the file; **`<byte-character>`** and **`<unicode-character>`** are likely choices. See Section 5.5.1.2 on page 38.

See also

`<buffered-stream>`, page 36

`<file-stream>`, page 70

`type-for-file-stream`, page 103

make *sequence-stream-class* — *G.f. method*

Summary

Creates and opens a stream over a sequence.

Signature

`make` ***sequence-stream-class*** **`#key`** ***contents direction start end***
`=>` ***sequence-stream-instance***

Arguments

sequence-stream-class

The class **`<sequence-stream>`**.

contents An instance of **`<sequence>`**.

direction One of **`#"input"`**, **`#"output"`**, or **`#"input-output"`**. Default value: **`#"input"`**.

start An instance of **`<integer>`**. Default value: 0.

end An instance of **`<integer>`**. Default value: *contents*`.size`.

Values

sequence-stream-instance

An instance of **`<sequence-stream>`**.

Description

Creates and opens a stream over a sequence.

This method returns a general instance of **`<sequence-stream>`**. To determine the concrete subclass to be instantiated, this method calls the generic function **`type-for-sequence-stream`**.

The *contents* init-keyword is a general instance of **`<sequence>`** which is used as the input for input streams, and as the initial storage for an output stream. If *contents* is a stretchy vector, then it is the only storage used by the stream.

The *direction* init-keyword specifies the direction of the stream.

The *start* and *end* init-keywords are only valid when *direction* is **`#"input"`**. They specify the portion of the sequence to create the stream over: *start* is inclusive and *end* is exclusive. The default is to stream over the entire sequence.

Example

```
let sv = make(<stretchy-vector>);
let stream = make(<sequence-stream>,
                  contents: sv,
                  direction: #"output");

write(stream,#(1, 2, 3, 4, 5, 6, 7, 8, 9));

write(stream,"ABCDEF");

values(sv, stream-contents(stream));
```

See also

`<sequence-stream>`, page 91

`type-for-sequence-stream`, page 104

make *string-stream-class* — *G.f. method*

Summary

Creates and opens a stream over a string.

Signature

`make` ***string-stream-class*** **`#key`** ***contents direction start end***
`=>` ***string-stream-instance***

Arguments

string-stream-class

The class **`<string-stream>`**.

contents — An instance of **`<string>`**.

direction — One of **`#"input"`**, **`#"output"`**, or **`#"input-output"`**. Default value: **`#"input"`**.

start — An instance of **`<integer>`**. Default value: 0.

end — An instance of **`<integer>`**. Default value: *contents***`.size`**.

Values

string-stream-instance

An instance of **`<string-stream>`**.

Description

Creates and opens a stream over a string.

This method returns an instance of **`<string-stream>`**. If supplied, *contents* describes the contents of the stream. The *direc-*

tion, *start*, and *end* init-keywords are as for **make** on **<sequence-stream>**.

Example

```
let stream = make(<string-stream>,
                  contents: "here is a sequence");
```

See also

make sequence-stream-class, page 76

<string-stream>, page 102

make *unicode-string-stream-class* — *G.f. method*

Summary

Creates and opens a stream over a Unicode string.

Signature

make ***unicode-string-stream-class*** **#key** ***contents direction start end*** **=>** ***unicode-string-stream-instance***

Arguments

unicode-string-stream-class

The class **<unicode-string-stream>**.

contents An instance of **<unicode-string>**.

direction One of **#"input"**, **#"output"**, or **#"input-output"**. Default value: **#"input"**.

start An instance of **<integer>**. Default value: 0.

end An instance of **<integer>**. Default value: *contents***.size**.

Values

unicode-string-stream-instance

An instance of **<unicode-string-stream>**.

Description

Creates and opens a stream over a Unicode string.

This method returns a new instance of **<unicode-string-stream>**. If supplied, *contents* describes the contents of the stream, and must be an instance of **<unicode-string>**. The *direction*, *start*, and *end* init-keywords are as for **`make`** on **`<sequence-stream>`**.

See also

`make sequence-stream-class`, page 76

`<unicode-string-stream>`, page 105

new-line *Open generic function*

Summary

Writes a newline sequence to an output stream.

Signature

`new-line` ***output-stream*** **`=> ()`**

Arguments

output-stream An instance of **`<stream>`**.

Values

None.

Description

Writes a newline sequence to *output-stream*.

A method for **`new-line`** is defined on **`<string-stream>`** that writes the character **`\n`** to the string stream.

outer-stream *Open generic function*

Summary

Returns a stream's wrapper stream.

Signature

`outer-stream` ***stream*** **`=>`** ***wrapping-stream***

Arguments

stream An instance of **`<stream>`**.

Values

wrapping-stream An instance of **`<wrapper-stream>`**.

Description

Returns the stream that is wrapping *stream*.

See also

`inner-stream`, page 72

`outer-stream-setter`, page 81

`<wrapper-stream>`, page 109

outer-stream-setter *Open generic function*

Summary
Sets a stream's wrapper stream.

Signature
`outer-stream-setter` ***wrapper-stream stream*** **`=>`** ***wrapper-stream***

Arguments

wrapper-stream	An instance of **`<wrapper-stream>`**.
stream	An instance of **`<stream>`**.

Values

wrapper-stream	An instance of **`<wrapper-stream>`**.

Description
Sets the **`outer-stream`** slot of *stream* to *wrapper-stream*.

Note: Applications should not set **`inner-stream`** and **`outer-stream`** slots directly. The **`outer-stream-setter`** function is for use only when implementing stream classes.

See also
`inner-stream-setter`, page 73

`outer-stream`, page 80

peek *Open generic function*

Summary
Returns the next element of a stream without advancing the stream position.

Signature
`peek` ***input-stream*** **`#key`** ***on-end-of-stream*** **`=>`** ***element-or-eof***

Arguments

input-stream	An instance of **`<stream>`**.
on-end-of-stream	An instance of **`<object>`**.

Values

element-or-eof	An instance of **`<object>`**, or **`#f`**.

Description This function behaves as `read-element` does, but the stream position is not advanced.

See also `read-element`, page 44

<positionable-stream> *Open abstract class*

Summary The class of positionable streams.

Superclasses `<stream>`

Init-keywords See Superclasses.

Description A subclass of `<stream>` supporting the Positionable Stream Protocol. It is not instantiable.

Operations `adjust-stream-position` `stream-contents` `stream-position` `stream-position-setter` `unread-element`

<position-type> *Type*

Summary A type representing positions in a stream.

Equivalent `type-union(<stream-position>, <integer>)`

Supertypes None.

Init-keywords None.

Description A type used to represent a position in a stream. In practice, positions within a stream are defined as instances of `<integer>`, but this type, together with the `<stream-position>` class, allows for cases where this might not be possible.

See also `<stream-position>`, page 99

read *Open generic function*

Summary Reads a number of elements from an input stream.

Signature **`read`** ***input-stream n*** **`#key`** ***on-end-of-stream*** **`=>`** ***sequence-or-eof***

Arguments

input-stream	An instance of **`<stream>`**.
n	An instance of **`<integer>`**.
on-end-of-stream	An instance of **`<object>`**.

Values

sequence-or-eof	An instance of **`<sequence>`**, or an instance of **`<object>`** if the end of stream is reached.

Description Returns a sequence of the next *n* elements from *input-stream*.

The type of the sequence returned depends on the type of the stream's underlying aggregate. For instances of **`<sequence-stream>`**, the type of the result is given by **`type-for-copy`** of the underlying aggregate. For instances of **`<file-stream>`**, the result is a vector that can contain elements of the type returned by calling **`stream-element-type`** on the stream.

The stream position is advanced so that subsequent reads start after the *n* elements.

If the stream is not at its end, **`read`** waits until input becomes available.

If the end of the stream is reached before all *n* elements have been read, the behavior is as follows.

- If a value for the *on-end-of-stream* argument was supplied, it is returned as the value of **`read`**.
- If a value for the *on-end-of-stream* argument was not supplied, and at least one element was read from the stream,

then an **`<incomplete-read-error>`** condition is signaled. When signaling this condition, **`read`** supplies two values: a sequence of the elements that were read successfully, and *n*.

- If the *on-end-of-stream* argument was not supplied, and no elements were read from the stream, an **`<end-of-stream-error>`** condition is signalled.

See also

`<end-of-stream-error>`, page 67

`<incomplete-read-error>`, page 71

`stream-element-type`, page 96

read-element *Open generic function*

Summary

Reads the next element in a stream.

Signature

`read-element` ***input-stream*** **`#key`** ***on-end-of-stream***
`=>` ***element-or-eof***

Arguments

input-stream	An instance of **`<stream>`**.
on-end-of-stream	An instance of **`<object>`**.

Values

element-or-eof	An instance of **`<object>`**.

Description

Returns the next element in the stream. If the stream is not at its end, the stream is advanced so that the next call to **`read-element`** returns the next element along in *input-stream*.

The *on-end-of-stream* keyword allows you to specify a value to be returned if the stream is at its end. If the stream is at its end and no value was supplied for *on-end-of-stream*, **`read-element`** signals an **`<end-of-stream-error>`** condition.

If no input is available and the stream is not at its end, **`read-element`** waits until input becomes available.

Example

The following piece of code applies **`function`** to all the elements of a sequence:

```
let stream = make(<sequence-stream>, contents: seq);
while (~stream-at-end?(stream))
  function(read-element(stream));
end;
```

See also

`<end-of-stream-error>`, page 67

`peek`, page 81

`unread-element`, page 58

read-into! *Open generic function*

Summary

Reads a number of elements from a stream into a sequence.

Signature

`read-into!` ***input-stream n sequence*** **`#key`** ***start on-end-of-stream*** **`=>`** ***count-or-eof***

Arguments

input-stream	An instance of **`<stream>`**.
n	An instance of **`<integer>`**.
sequence	An instance of **`<mutable-sequence>`**.
start	An instance of **`<integer>`**.
on-end-of-stream	An instance of **`<object>`**.

Values

count-or-eof	An instance of **`<integer>`**, or an instance of **`<object>`** if the end of stream is reached..

Description

Reads the next *n* elements from *input-stream*, and inserts them into a mutable sequence starting at the position *start*. Returns the number of elements actually inserted into *sequence* unless the end of the stream is reached, in which case the *on-end-of-stream* behavior is as for **`read`**.

If the sum of *start* and *n* is greater than the size of *sequence*, **`read-into!`** reads only enough elements to fill sequence up to the end. If *sequence* is a stretchy vector, no attempt is made to grow it.

If the stream is not at its end, **`read-into!`** waits until input becomes available.

See also **`read`**, page 44

read-line *Open generic function*

Summary Reads a stream up to the next newline.

Signature **`read-line`** ***input-stream*** **`#key`** ***on-end-of-stream***
`=>` ***string-or-eof newline?***

Arguments

	input-stream	An instance of **`<stream>`**.
	on-end-of-stream	An instance of **`<object>`**.

Values

	string-or-eof	An instance of **`<string>`**, or an instance of **`<object>`** if the end of the stream is reached.
	newline?	An instance of **`<boolean>`**.

`io`Description Returns a new string containing all the input in *input-stream* up to the next newline sequence.

The resulting string does not contain the newline sequence. The second value returned is **`#t`** if the read terminated with a newline or **`#f`** if the read terminated because it came to the end of the stream.

The type of the result string is chosen so that the string can contain characters of *input-stream*'s element type. For example, if the element type is **`<byte-character>`**, the string will be a **`<byte-string>`**.

If *input-stream* is at its end immediately upon calling **`read-line`** (that is, the end of stream appears to be at the end of an empty line), then the end-of-stream behavior and the interpretation of *on-end-of-stream* is as for **`read-element`**.

Example

See also **`read-element`**, page 44

read-line-into! *Open generic function*

Summary Reads a stream up to the next newline into a string.

Signature

```
read-line-into! input-stream string
                      #key start on-end-of-stream grow?
=> string-or-eof newline?
```

Arguments

input-stream	An instance of **`<stream>`**.
string	An instance of **`<string>`**.
start	An instance of **`<integer>`**. Default value: 0.
on-end-of-stream	An instance of **`<object>`**.
grow?	An instance of **`<boolean>`**. Default value: **`#f`**.

Values

string-or-eof	An instance of **`<string>`**, or an instance of **`<object>`** if the end of the stream is reached.
newline?	An instance of **`<boolean>`**.

Description

Fills *string* with all the input from *input-stream* up to the next newline sequence. The *string* must be a general instance of **`<string>`** that can hold elements of the stream's element type.

The input is written into *string* starting at the position *start*. By default, *start* is the start of the stream.

The second return value is **#t** if the read terminated with a newline, or **#f** if the read completed by getting to the end of the input stream.

If *grow?* is **#t**, and *string* is not large enough to hold all of the input, **read-line-into!** creates a new string which it writes to and returns instead. The resulting string holds all the original elements of *string*, except where **read-line-into!** overwrites them with input from *input-stream*.

In a manner consistent with the intended semantics of *grow?*, when *grow?* is **#t** and *start* is greater than or equal to *string*.**size**, **read-line-into!** grows *string* to accommodate the *start* index and the new input.

If *grow?* is **#f** and *string* is not large enough to hold the input, the function signals an error.

The end-of-stream behavior and the interpretation of *on-end-of-stream* is the same as that of **read-line**.

See also **read-line**, page 86

read-through *Function*

Summary Returns a sequence containing the elements of the stream up to, and including, the first occurrence of a given element.

Signature **read-through** ***input-stream element*** **#key** ***on-end-of-stream test***
=> ***sequence-or-eof found?***

Arguments

input-stream	An instance of **<stream>**.
element	An instance of **<object>**.
on-end-of-stream	An instance of **<object>**.
test	An instance of **<function>**. Default value: **==**.

Values

sequence-or-eof	An instance of **`<sequence>`**, or an instance of **`<object>`** if the end of the stream is reached.
found?	An instance of **`<boolean>`**.

Description

This function is the same as **`read-to`**, except that *element* is included in the resulting sequence.

If the *element* is not found, the result does not contain it. The stream is left positioned after *element*.

See also

`read-to`, page 89

read-to *Function*

Summary

Returns a sequence containing the elements of the stream up to, but not including, the first occurrence of a given element.

Signature

`read-to` ***input-stream element*** **`#key`** ***on-end-of-stream test*** **`=>`** ***sequence-or-eof found?***

Arguments

input-stream	An instance of **`<stream>`**.
element	An instance of **`<object>`**.
on-end-of-stream	An instance of **`<object>`**.
test	An instance of **`<function>`**. Default value: **`==`**.

Values

sequence-or-eof	An instance of **`<sequence>`**, or an instance of **`<object>`** if the end of the stream is reached.
found?	An instance of **`<boolean>`**.

Description

Returns a new sequence containing the elements of *input-stream* from the stream's current position to the first occurrence of *element*. The result does not contain *element*.

The second return value is **#t** if the read terminated with *element*, or **#f** if the read terminated by reaching the end of the stream's source. The "boundary" element is consumed, that is, the stream is left positioned after *element*.

The **read-to** function determines whether the element occurred by calling the function *test*. This function must accept two arguments, the first of which is the element retrieved from the stream first and the second of which is *element*.

The type of the sequence returned is the same that returned by **read**. The end-of-stream behavior is the same as that of **read-element**.

See also **read-element**, page 44

read-to-end *Function*

Summary Returns a sequence containing all the elements up to, and including, the last element of the stream.

Signature **read-to-end** ***input-stream*** **=>** ***sequence***

Arguments	*input-stream*	An instance of **<stream>**.
Values	*sequence*	An instance of **<sequence>**.

Description Returns a sequence of all the elements up to, and including, the last element of *input-stream*, starting from the stream's current position.

The type of the result sequence is as described for **read**. There is no special end-of-stream behavior; if the stream is already at its end, an empty collection is returned.

Example

```
read-to-end(make(<string-stream>,
                 contents: "hello there, world",
                 start: 6,
                 end: 11));
```

See also **read**, page 44

<sequence-stream> *Open instantiable class*

Summary The class of streams over sequences.

Superclasses **<positionable-stream>**

Init-keywords

contents: A general instance of **<sequence>** which is used as the input for an input stream, and as the initial storage for an output stream.

direction: Specifies the direction of the stream. It must be one of **#"input"**, **#"output"**, or **#"input-output"**. Default value: **#"input"**.

start: An instance of **<integer>**. This specifies the start position of the sequence to be streamed over. Only valid when **direction:** is **#"input"**. Default value: 0.

end: An instance of **<integer>**. This specifies the sequence position immediately after the portion of the sequence to stream over. Only valid when **direction:** is **#"input"**. Default value: **contents.size**.

Description The class of streams over sequences. It is a subclass of **<positionable-stream>**.

If **contents:** is a stretchy vector, then it is the only storage used by the stream.

The **<sequence-stream>** class can be used for streaming over all sequences, but there are also subclasses **<string-stream>**, **<byte-string-stream>**, and **<unicode-string-stream>**, which are specialized for streaming over strings.

The **start:** and **end:** init-keywords specify the portion of the sequence to create the stream over: **start:** is inclusive and **end:** is exclusive. The default is to stream over the entire sequence.

Operations

make sequence-stream-class

See also

<byte-string-stream>, page 63

make sequence-stream-class, page 76

<string-stream>, page 102

<unicode-string-stream>, page 105

skip-through *Function*

Summary

Skips through an input stream past the first occurrence of a given element.

Signature

skip-through ***input-stream element*** **#key** ***test*** **=>** ***found?***

Arguments

input-stream	An instance of **<stream>**.
element	An instance of **<object>**.
test	An instance of **<function>**. Default value: **==**.

Values

found?	An instance of **<boolean>**.

Description

Positions *input-stream* after the first occurrence of *element*, starting from the stream’s current position. Returns **#t** if the element was found, or **#f** if the end of the stream was encountered. When **skip-through** does not find *element*, it leaves *input-stream* positioned at the end.

The **skip-through** function determines whether the element occurred by calling the test function *test*. The test function must accept two arguments. The order of the arguments is the element retrieved from the stream first and element second.

<stream> *Open abstract class*

Summary

The superclass of all stream classes.

Superclasses

<object>

Init-keywords

outer-stream: The name of the stream wrapping the stream. Default value: the stream itself (that is, the stream is not wrapped).

Description

The superclass of all stream classes and a direct subclass of **<object>**. It is not instantiable.

The **outer-stream:** init-keyword should be used to delegate a task to its wrapper stream. See Section 5.8.1 on page 53 for more information.

Operations

close **discard-input** **discard-output** **force-output** **new-line** **outer-stream** **outer-stream-setter** **peek** **read** **read-element** **read-into!** **read-line** **read-line-into!** **read-through** **read-to** **read-to-end** **skip-through** **stream-at-end?** **stream-element-type** **stream-input-available?** **stream-lock** **stream-lock-setter** **stream-open?** **synchronize-output** **write** **write-element**

stream-at-end? *Open generic function*

Summary

Tests whether a stream is at its end.

Signature

`stream-at-end?` ***stream*** **`=>`** ***at-end?***

Arguments

stream	An instance of **`<stream>`**.

Values

at-end?	An instance of **`<boolean>`**.

Description

Returns **`#t`** if the stream is at its end and **`#f`** if it is not. For input streams, it returns **`#t`** if a call to **`read-element`** with no supplied keyword arguments would signal an **`<end-of-stream-error>`**.

This function differs from **`stream-input-available?`**, which tests whether the stream can be read.

For output-only streams, this function always returns **`#f`**.

For output streams, note that you can determine if a stream is one place past the last written element by comparing **`stream-position`** to **`stream-size`**.

Example

The following piece of code applies **`function`** to all the elements of a sequence:

```
let stream = make(<sequence-stream>, contents: seq);
while (~stream-at-end?(stream))
  function(read-element(stream));
end;
```

See also

`<end-of-stream-error>`, page 67

`read-element`, page 44

`stream-input-available?`, page 96

stream-contents *Open generic function*

Summary

Returns a sequence containing all the elements of a positionable stream.

Signature

`stream-contents` ***positionable-stream*** **`#key`** ***clear-contents?***
`=>` ***sequence***

Arguments

positionable-stream	An instance of **`<positionable-stream>`**.
clear-contents?	An instance of **`<boolean>`**. Default value: **`#t`**.

Values

sequence	An instance of **`<sequence>`**.

Description

Returns a sequence that contains all of *positionable-stream*'s elements from its start to its end, regardless of its current position. The type of the returned sequence is as for **`read`**. See page 44.

The *clear-contents?* argument is only applicable to writeable sequence streams, and is not defined for file-streams or any other external stream. It returns an error if applied to an input only stream. If *clear-contents?* is **`#t`** (the default for cases where the argument is defined), this function sets the size of the stream to zero, and the position to the stream's start. Thus the next call to **`stream-contents`** will return only the elements written after the previous call to **`stream-contents`**.

Note that the sequence returned never shares structure with any underlying sequence that might be used in the future by the stream. For instance, the string returned by calling **`stream-contents`** on an output **`<string-stream>`** will not be the same string as that being used to represent the string stream.

Example

The following forms bind **`stream`** to an output stream over an empty string and create the string “I see!”, using the function **`stream-contents`** to access all of the stream's elements.

```
let stream = make(<byte-string-stream>,
                  direction: #"output");
write-element(stream, 'I');
write-element(stream, ' ');
write(stream, "see");
write-element(stream, '!');
stream-contents(stream);
```

See also **`read-to-end`**, page 46

`stream-contents`, page 95

stream-element-type *Open generic function*

Summary Returns the element-type of a stream.

Signature **`stream-element-type`** ***stream*** **`=>`** ***element-type***

Arguments *stream* An instance of **`<stream>`**.

Values *element-type* An instance of **`<type>`**.

Description Returns the element type of *stream* as a Dylan **`<type>`**.

stream-input-available? *Open generic function*

Summary Tests if an input stream can be read.

Signature **`stream-input-available?`** ***input-stream*** **`=>`** ***available?***

Arguments *input-stream* An instance of **`<stream>`**.

Values *available?* An instance of **`<boolean>`**.

Description Returns **`#t`** if *input-stream* would not block on **`read-element`**, otherwise it returns **`#f`**.

This function differs from **`stream-at-end?`**. When **`stream-input-available?`** returns **`#t`**, **`read-element`** will not block, but it may detect that it is at the end of the stream's source, and consequently inspect the *on-end-of-stream* argument to determine how to handle the end of stream.

See also **`read-element`**, page 44

`stream-at-end?`, page 94

stream-lock *Open generic function*

Summary Returns the lock for a stream.

Signature **`stream-lock`** ***stream*** **`=>`** ***lock***

Arguments *stream* An instance of **`<stream>`**.

Values *lock* An instance of **`<lock>`**, or **`#f`**.

Description Returns *lock* for the specified *stream*. You can use this function, in conjunction with **`stream-lock-setter`** to implement a basic stream locking facility. For full details on the **`<lock>`** class, see the documentation on the Threads library in the *Core Features and Mathematics* manual.

See also **`stream-lock-setter`**, page 97

stream-lock-setter *Open generic function*

Summary Sets a lock on a stream.

Signature **`stream-lock-setter`** ***stream lock*** **`=>`** ***lock***

Arguments *stream* An instance of **`<stream>`**.

lock An instance of **`<lock>`**, or **`#f`**.

Values *lock* An instance of **`<lock>`**, or **`#f`**.

Description Sets *lock* for the specified *stream*. If *lock* is **`#f`**, then the lock on *stream* is freed. You can use this function in conjunction with **`stream-lock`** to implement a basic stream locking facility. For full details on the **`<lock>`** class, see the documentation on the Threads library in the *Core Features and Mathematics* manual.

See also **`stream-lock`**, page 97

stream-open? *Open generic function*

Summary Generic function for testing whether a stream is open.

Signature **`stream-open?`** ***stream*** **`=>`** ***open?***

Arguments *stream* An instance of **`<stream>`**.

Values *open?* An instance of **`<boolean>`**.

Description Returns **`#t`** if *stream* is open and **`#f`** if it is not.

See also **`close`**, page 65

stream-position *Open generic function*

Summary Finds the current position of a positionable stream.

Signature **`stream-position`** ***positionable-stream*** **`=>`** ***position***

Arguments *positionable-stream*

An instance of **`<positionable-stream>`**.

Values *position* An instance of **`<position-type>`**.

Description Returns the current position of *positionable-stream* for reading or writing.

The value returned can be either an instance of **`<stream-position>`** or an integer. When the value is an integer, it is an offset from position zero, and is in terms of the stream's element type. For instance, in a Unicode stream, a position of four means that four Unicode characters have been read.

Example The following example uses positioning to return the character "w" from a stream over the string **`"hello world"`**:

```
let stream = make(<string-stream>,
                  contents: "hello world");
stream-position(stream) := 6;
read-element(stream);
```

See also **`<position-type>`**, page 82

<stream-position> *Abstract class*

Summary The class representing non-integer stream positions.

Superclasses **`<object>`**

Init-keywords None.

Description A direct subclass of **`<object>`**. It is used in rare cases to represent positions within streams that cannot be represented by instances of **`<integer>`**. For example, a stream that supports compression will have some state associated with each position in the stream that a single integer is not sufficient to represent.

The **`<stream-position>`** class is disjoint from the class **`<integer>`**.

Operations **as** **stream-position-setter** **stream-size**

See also **<position-type>**, page 82

stream-position-setter *Open generic function*

Summary Sets the position of a stream.

Signature **stream-position-setter** ***position positionable-stream***
=> ***new-position***

Arguments

position	An instance of **<position-type>**.
positionable-stream	An instance of **<positionable-stream>**.

Values

new-position	An instance of **<stream-position>**, or an instance of **<integer>**.

Description Changes the stream’s position for reading or writing to *position*.

When it is an integer, if it is less than zero or greater than *positionable-stream***.stream-size** this function signals an error. For file streams, a **<stream-position-error>** is signalled. For other types of stream, the error signalled is **<simple-error>**.

When *position* is a **<stream-position>**, if it is invalid for some reason, this function signals an error. Streams are permitted to restrict the *position* to being a member of the set of values previously returned by calls to **stream-position** on the same stream.

The *position* may also be **#"start"**, meaning that the stream should be positioned at its start, or **#"end"**, meaning that the stream should be positioned at its end.

Note: You cannot use **`stream-position-setter`** to set the position past the current last element of the stream: use **`adjust-stream-position`** instead.

See also

`adjust-stream-position`, page 59

`<stream-position>`, page 99

stream-size *Open generic function*

Summary

Finds the number of elements in a stream.

Signature

`stream-size` ***positionable-stream*** **`=>`** ***size***

Arguments

positionable-stream

An instance of **`<positionable-stream>`**.

Values

size An instance of **`<integer>`**, or **`#f`**.

Description

Returns the number of elements in *positionable-stream*.

For input streams, this is the number of elements that were available when the stream was created. It is unaffected by any read operations that might have been performed on the stream.

For output and input-output streams, this is the number of elements that were available when the stream was created (just as with input streams), added to the number of elements written past the end of the stream (regardless of any repositioning operations).

It is assumed that:

- There is no more than one stream open on the same source or destination at a time.
- There are no shared references to files by other processes.

- There are no alias references to underlying sequences, or any other such situations.

In such situations, the behavior of **stream-size** is undefined.

<string-stream> *Open instantiable class*

Summary

The class of streams over strings.

Superclasses

<sequence-stream>

Init-keywords

contents:	A general instance of **<sequence>**.
direction:	Specifies the direction of the stream. It must be one of **#"input"**, **#"output"**, or **#"input-output"**; Default value: **#"input"**.
start:	An instance of **<integer>**. Only valid when **direction:** is **#"input"**. Default value: 0.
end:	An instance of **<integer>**. This specifies the string position immediately after the portion of the string to stream over. Only valid when **direction:** is **#"input"**. Default value: **contents.size**.

Description

The class of streams over strings.

The **contents:** init-keyword is used as the input for an input stream, and as the initial storage for an output stream.

The **start:** init-keyword specifies the start position of the string to be streamed over.

The class supports the same init-keywords as **<sequence-stream>**.

The **start:** and **end:** init-keywords specify the portion of the string to create the stream over: **start:** is inclusive and **end:** is exclusive. The default is to stream over the entire string.

Example **make string-stream-class**

See also **make string-stream-class**, page 78

<sequence-stream>, page 91

synchronize-output *Open generic function*

Summary Synchronizes an output stream with the application state.

Signature **synchronize-output** ***output-stream*** **=> ()**

Arguments *output-stream* An instance of **<stream>**.

Values None.

Description Forces any pending output from *output-stream*'s buffers to its destination. Before returning to its caller, **synchronize-output** also attempts to ensure that the output reaches the stream's destination before, thereby synchronizing the output destination with the application state.

When creating new stream classes it may be necessary to add a method to the **synchronize-output** function, even though it is not part of the Stream Extension Protocol.

See also **force-output**, page 71

type-for-file-stream *Open generic function*

Summary Finds the type of file-stream class that needs to be instantiated for a given file.

Signature **type-for-file-stream** ***filename element-type*** **#rest #all-keys**
=> ***file-stream-type***

Arguments *filename* An instance of **`<object>`**.

element-type One of **`<byte-character>`**, **`<unicode-character>`**, or **`<byte>`**, or **`#f`**.

Values *file-stream-type* An instance of **`<type>`**.

Description Returns the kind of file-stream class to instantiate for a given file. The method for **`make`** on **`<file-stream>`** calls this function to determine the class of which it should create an instance.

See also **`<file-stream>`**, page 70

`make file-stream-class`, page 75

type-for-sequence-stream *Open generic function*

Summary Finds the type of sequence-stream class that needs to be instantiated for a given sequence.

Signature **`type-for-sequence-stream`** ***sequence*** **`=>`** ***sequence-stream-type***

Arguments *sequence* An instance of **`<sequence>`**.

Values *sequence-stream-type*

An instance of **`<type>`**.

Description Returns the sequence-stream class to instantiate over a given sequence object. The method for **`make`** on **`<sequence-stream>`** calls this function to determine the concrete subclass of **`<sequence-stream>`** that it should instantiate.

There are **`type-for-sequence-stream`** methods for each of the string object classes. These methods return a stream class object that the Streams module considers appropriate.

See also **make sequence-stream-class**, page 76

<sequence-stream>, page 91

<unicode-character> *Type*

Summary The type that represents Unicode characters.

Supertypes **<character>**

Init-keywords None.

Description A type representing Unicode characters that instances of **<unicode-string>** can contain.

Operations **type-for-file-stream**

<unicode-string-stream> *Open instantiable class*

Summary The class of streams over Unicode strings.

Superclasses **<string-stream>**

Init-keywords

contents: A general instance of **<sequence>**.

direction: Specifies the direction of the stream. It must be one of **#"input"**, **#"output"**, or **#"input-output"**. Default value: **#"input"**.

start: An instance of **<integer>**. This specifies the start position of the Unicode string to be streamed over. Only valid when **direction:** is **#"input"**. Default value: 0.

end: An instance of **<integer>**. This specifies the sequence position immediately after the portion of the Unicode string to stream over. Only valid when **direction:** is **#"input"**. Default value: **contents.size**.

Description

The class of streams over Unicode strings. It is a subclass of **<string-stream>**.

The **contents:** init-keyword is used as the input for an input stream, and as the initial storage for an output stream. If it is a stretchy vector, then it is the only storage used by the stream.

The class supports the same init-keywords as **<sequence-stream>**.

The **start:** and **end:** init-keywords specify the portion of the Unicode string to create the stream over: **start:** is inclusive and **end:** is exclusive. The default is to stream over the entire Unicode string.

Operations

make unicode-string-stream-class

See also

make unicode-string-stream-class, page 79

<sequence-stream>, page 91

unread-element *Open generic function*

Summary

Returns an element that has been read back to a positionable stream.

Signature

unread-element ***positionable-stream element*** **=>** ***element***

Arguments

positionable-stream

An instance of **<positionable-stream>**.

element An instance of **<object>**.

Values *element* An instance of **`<object>`**.

Description "Unreads" the last element from *positionable-stream*. That is, it returns *element* to the stream so that the next call to **`read-element`** will return *element*. The stream must be a **`<positionable-stream>`**.

It is an error to do any of the following:

- To apply **`unread-element`** to an element that is not the element most recently read from the stream.
- To call **`unread-element`** twice in succession.
- To unread an element if the stream is at its initial position.
- To unread an element after explicitly setting the stream's position.

See also **`read-element`**, page 44

wait-for-io-completion *G. f. method*

Summary Waits for all pending operations on a stream to complete.

Signature **`wait-for-io-completion`** ***file-stream*** **`=> ()`**

Arguments *file-stream* An instance of **`<stream>`**.

Description If is asynchronous, waits for all pending write or close operations to complete and signals any queued errors. If *file-stream* is not asynchronous, returns immediately.

with-open-file *Statement macro*

Summary Runs a body of code within the context of a file stream.

Macro call **with-open-file (*stream-var* = *filename*, #rest *keys*)**
body end => _values_

Arguments

stream-var An Dylan variable-name$_{bnf}$.

filename An instance of **<string>**.

keys Instances of **<object>**.

body A Dylan body$_{bnf}$.

Values

values Instances of **<object>**.

Description

Provides a safe mechanism for working with file streams. The macro creates a file stream and binds it to *stream-var*, evaluates a *body* of code within the context of this binding, and then closes the stream. The macro calls **close** upon exiting *body*.

The values of the last expression in *body* are returned.

Any *keys* are passed to the **make** method on **<file-stream>**.

Example

The following expression yields the contents of file **foo.text** as a **<byte-vector>**:

```
with-open-file (fs = "foo.text", element-type: <byte>)
  read-to-end(fs)
end;
```

It is roughly equivalent to:

```
begin
  let hidden-
fs = #f;     // In case the user bashes fs variable
  block ()
    hidden-fs := make(<file-stream>,
                        locator: "foo.text", element-
type: <byte>);
    let fs = hidden-fs;
    read-to-end(fs);
  cleanup
    if (hidden-fs) close(hidden-fs) end;
  end block;
end;
```

See also

`close`, page 65

`<file-stream>`, page 70

`make file-stream-class`, page 75

<wrapper-stream> *Open instantiable class*

Summary

The class of wrapper-streams.

Superclasses

`<stream>`

Init-keywords

`inner-stream:` An instance of **`<stream>`**.

Description

The class that implements the basic wrapper-stream functionality.

It takes a required init-keyword **`inner-stream:`**, which is used to specify the wrapped stream.

The **`<wrapper-stream>`** class implements default methods for all of the stream protocol functions described in this document. Each default method on **`<wrapper-stream>`** simply “trampolines” to its inner stream.

Operations

`inner-stream`

`inner-stream-setter`

`outer-stream-setter`

Example

In the example below, **`<io-wrapper-stream>`**, a subclass of **`<wrapper-stream>`**, “passes on” functions such as **`read-element`** and **`write-element`** by simply delegating these operations to the inner stream:

```
define method read-element (ws :: <io-wrapper-stream>,
                            #key on-end-of-stream)
    => (element)
  read-element(ws.inner-stream)
end method;

define method write-element (ws :: <io-wrapper-stream>,
                             element)
    => ()
  write-element(ws.inner-stream,element)
end method;
```

Assuming that **<io-wrapper-stream>** delegates all other operations to its inner stream, the following is sufficient to implement a 16-bit Unicode character stream wrapping an 8-bit character stream.

```
define class <unicode-stream> (<io-wrapper-stream>)
end class;

define method read-element (s :: <unicode-stream>,
                            #key on-end-of-stream)
    => (ch :: <unicode-character>)
  with-stream-locked (s)
    let first-char =  read-element(s.inner-stream,
                                   on-end-of-stream);
    let second-char = read-element(s.inner-stream,
                                   on-end-of-stream);
  end;
  convert-byte-pair-to-unicode(first-char, second-char)
end method;

define method write-element  (s :: <unicode-stream>,
                              c :: <character>)
    => ()
  let (first-char, second-char)
      = convert-unicode-to-byte-pair(c);
  with-stream-locked (s)
    write-element(s.inner-stream, first-char);
    write-element(s.inner-stream, second-char)
  end;
  c
end method;
```

```
define method stream-position (s :: <unicode-stream>)
  =>  (p :: <integer>)
  truncate/(stream-position(s.inner-stream), 2)
end method;

define method stream-position-setter  (p :: <integer>,
                                       s :: <unicode-stream>);
  stream-position(s.inner-stream) := p * 2
end method;
```

write *Open generic function*

Summary

Writes the elements of a sequence to an output stream.

Signature

write ***output-stream sequence*** **#key** ***start end*** **=> ()**

Arguments

output-stream	An instance of **<stream>**.
sequence	An instance of **<sequence>**.
start	An instance of **<integer>**. Default value: 0.
end	An instance of **<integer>**. Default value: *sequence***.size**.

Values

None.

ioDescription

Writes the elements of *sequence* to *output-stream*, starting at the stream’s current position.

The elements in *sequence* are accessed in the order defined by the forward iteration protocol on **<sequence>**. This is effectively the same as the following:

```
do (method (elt) write-element(stream, elt)
    end, sequence);
sequence;
```

If supplied, *start* and *end* delimit the portion of *sequence* to write to the stream. The value of *start* is inclusive and that of *end* is exclusive.

If the stream is positionable, and it is not positioned at its end, **write** overwrites elements in the stream and then advance the stream's position to be beyond the last element written.

Implementation Note: Buffered streams are intended to provide a very efficient implementation of **write**, particularly when *sequence* is an instance of **<byte-string>**, **<unicode-string>**, **<byte-vector>**, or **<buffer>**, and the stream's element type is the same as the element type of *sequence*.

Example

The following forms bind **stream** to an output stream over an empty string and create the string "I see!", using the function **stream-contents** to access all of the stream's elements.

```
let stream = make(<byte-string-stream>,
                  direction: #"output");
write-element(stream, 'I');
write-element(stream, ' ');
write(stream, "see");
write-element(stream, '!');
stream-contents(stream);
```

See also

read, page 44

write-element, page 112

write-line, page 113

write-element *Open generic function*

Summary

Writes an element to an output stream.

Signature

write-element ***output-stream element*** **=> ()**

Arguments

output-stream	An instance of **<stream>**.
element	An instance of **<object>**.

Values

None.

Description

Writes *element* to *output-stream* at the stream's current position. The output-stream must be either **`#"output"`** or **`#"input-output"`**. It is an error if the type of *element* is inappropriate for the stream's underlying aggregate.

If the stream is positionable, and it is not positioned at its end, **`write-element`** overwrites the element at the current position and then advances the stream position.

Example

The following forms bind **`stream`** to an output stream over an empty string and create the string "I see!", using the function **`stream-contents`** to access all of the stream's elements.

```
let stream = make(<byte-string-stream>,
                  direction: #"output");
write-element(stream, 'I');
write-element(stream, ' ');
write-element(stream, 'd');
write-element(stream, 'o');
stream-contents(stream);
```

See also

`read-element`, page 44

`write`, page 111

`write-line`, page 113

write-line *Open generic function*

Summary

Writes a string followed by a newline to an output stream.

Signature

`write-line` ***output-stream string*** **`#key`** ***start end*** **`=> ()`**

Arguments

output-stream	An instance of **`<stream>`**.
string	An instance of **`<string>`**.
start	An instance of **`<integer>`**. Default value: 0.
end	An instance of **`<integer>`**. Default value: *string*.**`size`**.

Values None.

Description Writes *string* followed by a newline sequence to *output-stream*.

The default method behaves as though it calls **`write`** on *string* and then calls **`new-line`**.

If supplied, *start* and *end* delimit the portion of *string* to write to the stream.

See also **`read-line`**, page 86

`write`, page 111

`write-element`, page 112

6

The Standard-IO Module

6.1 Introduction

This document describes the Standard-IO module, which requires the Streams module. All interfaces described in this document are exported from the Standard-IO module. The functionality provided by this module mirrors some of the functionality provided by the **`Java.io`** package in Java.

For convenience, the Standard-IO module, together with the Format module, is repackaged by the Format-out module. See Chapter 3, "The Format-Out Module" for details.

6.2 Handling standard input and output

The Standard-IO module provides a Dylan interface to the standard I/O facility of operating systems such as MS-DOS or UNIX.

The module consists of three variables, each of which is bound to a stream.

standard-input *Variable*

An input stream that reads data from the platform's standard input location. It is equivalent to the Java stream **`java.lang.System.in`**.

standard-output *Variable*

An output stream that sends data to the platform's standard output location. It is equivalent to the Java stream **`java.lang.System.out`**.

standard-error *Variable*

An output stream that sends data to the platform's standard error location. It is equivalent to the Java stream **`java.lang.System.err`**.

For console-based applications (i.e., applications that run in character mode), the three streams just use the console window in which the application was started.

For purely window-based applications, each variable is bound by default to a stream that lazily creates a console window as soon as any input is requested or output is performed. Only one window is created, and this is shared between all three streams. Any subsequent input or output uses the same window. The window that is created uses the standard configuration settings set by the user. For example, the window is only scrollable if all console windows are configured to be scrollable on the machine running the application.

For more information about streams, please refer to Chapter 5, "The Streams Module".

6.3 The STANDARD-IO module

This section contains a complete reference of all the interfaces that are exported from the **`standard-io`** module.

standard-input *Variable*

Summary The standard input stream.

Type **`<stream>`**

Initial value The standard input stream for the platform on which the application is running.

Description

This variable is bound to an input stream that reads data from the standard input location for the platform on which the application is running. It is equivalent to the Java stream **`java.lang.System.in`**.

If the platform has a notion of standard streams, such as MS-DOS, this stream maps onto the platform-specific standard input stream. If the platform has no such convention, such as a platform that is primarily window-based, then a console window is created for this stream if necessary, in order to provide users with a place to provide input.

If a console window has already been created as a result of writing to one of the other variables in the Standard-IO module, then the existing console window is used, and a new one is not created: a single console window is used for all variables in this module.

standard-output *Variable*

Summary

The standard output stream.

Type

`<stream>`

Initial value

The standard output stream for the platform on which the application is running.

Description

This variable is bound to an output stream that sends normal output to the standard output location for the platform on which the application is running. It is equivalent to the Java stream **`java.lang.System.out`**.

If the platform has a notion of standard streams, such as MS-DOS, this stream maps onto the platform-specific standard output stream. If the platform has no such convention, such as a platform that is primarily window-based, a console window is created for this stream if necessary, just to capture output to it.

If a console window has already been created as a result of writing to or reading from one of the other variables in the Standard-IO module, then the existing console window is used, and a new one is not created: a single console window is used for all variables in this module.

standard-error *Variable*

Summary
The standard error stream.

Type
`<stream>`

Initial value
The standard error stream for the platform on which the application is running.

Description
This variable is bound to an output stream that sends error messages to the standard error location for the platform on which the application is running. It is equivalent to the Java stream **`java.lang.System.err`**.

If the platform has a notion of standard streams, such as MS-DOS, this stream maps onto the platform-specific standard error stream. If the platform has no such convention, such as a platform that is primarily window-based, a console window is created for this stream if necessary, just to capture output to it.

If a console window has already been created as a result of writing to or reading from one of the other variables in the Standard-IO module, then the existing console window is used, and a new one is not created: a single console window is used for all variables in this module.

7

The Date Module

7.1 Introduction

The Date module is part of the System library and provides a machine-independent facility for representing and manipulating dates and date/time intervals.

This chapter describes the classes, types, and functions that the Date module contains.

7.2 Representing dates and times

The Date module contains a single class, **`<date>`**, an instance of which can represent any date between 1 Jan 1800 00:00:00 and 31 Dec 2199 23:59:59, Greenwich Mean Time. You can create a date object by calling the function **`encode-date`** or using the **`make`** method for **`<date>`**.

<date> *Sealed class*

The class of all date objects. It is a subclass of **`<number>`**. Note that there is no method for **`make`** defined on **`<date>`**. The function **`encode-date`** should be used instead.

make *date-class* — *G.f. method*

`make` ***date-class*** **`#key`** ***iso8601-string year month day hours minutes seconds microseconds time-zone-offset***
`=>` ***date-instance***

Creates an instance of the **`<date>`** class.

encode-date — *Function*

`encode-date` ***year month day hours minutes seconds***
`#key` ***microseconds time-zone-offset*** **`=>`** ***date***

Creates a **`<date>`** object from a set of **`<integer>`** values.

Each of the arguments to **`encode-date`** and to the **`make`** method on **`<date>`** is an instance of **`<integer>`** (except for the *iso8601-string* keyword for the **`make`** method, which is a string) that is passed as an init-keyword value to the **`<date>`** object. Each init-keyword takes an instance of **`<integer>`**, limited to the natural range for that attribute. For example, **`month:`** can only take values between 1 and 12.

You must specify values, via **`encode-date`**, for at least the **`year:`**, **`month:`**, and **`day:`** init-keywords. In addition, you can also specify values for **`hours:`**, **`minutes:`**, **`seconds:`**, **`microseconds:`**, and **`time-zone-offset:`**. If not supplied, the default value for each of these init-keywords is 0.

The **`time-zone-offset:`** init-keyword is used to represent time zones in the Date module as **`<integer>`** values representing the offset in minutes from Greenwich Mean Time (GMT). Positive values are used for time zones East of Greenwich; negative values represent time zones to the west of Greenwich.

For example, the value -300 (-5 hours) is U.S. Eastern Standard Time and the value -240 (-4 hours) is U.S. Eastern Daylight Savings Time.

If you wish, a **`<date>`** can be specified completely by using the **`iso8601-string:`** init-keyword. This init-keyword takes an instance of **`<string>`**, which should be a valid ISO8601 format date. If you use the **`iso8601-string:`** init-keyword, there is no need to specify any other init-keywords to a call to **`make`** on **`<date>`**.

current-date *Function*

```
current-date () => date
```

Returns the current date on your local machine as a **<date>** object.

<day-of-week> *Type*

```
one-of(#"Sunday", #"Monday", #"Tuesday", #"Wednesday",
       #"Thursday", #"Friday", #"Saturday")
```

Days of the week can be represented using the **<day-of-week>** type.

You can extract the day of the week of a specified date using **date-day-of-week**. See Section 7.5 for details.

7.3 Representing durations

Date / time intervals, called durations, are modeled in a style quite similar to that of SQL. There are two, effectively disjoint, classes of duration: one with a resolution of months (for example, 3 years, 2 months) and the other with a resolution of microseconds (for example, 50 days, 6 hours, 23 minutes). The first is **<year/month-duration>** and the second **<day/time-duration>**.

An important distinction between **<day/time-duration>** and **<year/month-duration>** is that a given instance of **<day/time-duration>** is always a fixed unit of a fixed length, whereas a **<year/month-duration>** follows the vagaries of the calendar. So if you have a **<date>** that represents, for example, the 5th of some month, adding a **<year/month-duration>** of 1 month to that will always take you to the 5th of the following month, whether that is an interval of 28, 29, 30, or 31 days.

<duration> *Sealed abstract instantiable class*

This class is the used to represent durations. It is a subclass of **<number>**, and it has two subclasses, described below.

<year/month-duration> *Sealed class*

Represents durations in units of calendar months. It is a subclass of **`<duration>`**.

<day/time-duration> *Sealed class*

Represents durations in units of microseconds. It is a subclass of **`<duration>`**.

The following functions and methods are available for creating durations, and decoding them into their constituent integer parts.

encode-year/month-duration *Function*

`encode-year/month-duration` ***years months*** **`=>`** ***duration***

Creates an instance of **`<year/month-duration>`**.

encode-day/time-duration *Function*

`encode-day/time-duration` ***days hours minutes seconds microseconds***
`=>` ***duration***

Creates an instance of **`<day/time-duration>`**.

decode-duration *Sealed generic function*

`decode-duration` ***duration*** **`=> #rest`** ***components***

Decodes an instance of **`<duration>`** into its constituent parts. There are methods for this generic function that specialize on **`<year/month-duration>`** and **`<day/time-duration>`** respectively, as described below.

decode-duration *Sealed method*

`decode-duration` ***duration*** **`=>`** ***years months***

Decodes an instance of **`<year/month-duration>`** into its constituent parts.

decode-duration *Sealed method*

`decode-duration` ***duration* => *days hours minutes seconds microseconds***

Decodes an instance of **`<day/time-duration>`** into its constituent parts.

7.4 Performing operations on dates and durations

A number of interfaces are exported from the Date module that let you perform other operations on dates and durations, and extract date-specific information from your local machine.

7.4.1 Comparing dates

The following operations are exported from the Date module.

= *Sealed method*

< *Sealed method*

date1* = *date2* => *equal?
date1* < *date2* => *before?

These methods let you perform arithmetic-like operations on dates to test for equality, or to test whether one date occurred before another.

7.4.2 Comparing durations

The following operations are exported from the Date module.

= *Sealed method*

< *Sealed method*

duration1* = *duration2* => *equal?
duration1* < *duration2* => *less-than?

As with dates, you can perform arithmetic-like operations on durations to test for equality, or to test whether one duration is shorter than another.

7.4.3 Performing arithmetic operations

You can add, subtract, multiply, and divide dates and durations in a number of ways to produce a variety of date or duration information. Methods are defined for any combination of date and duration, with any operation that makes sense, and the return value is of the appropriate type.

For example, a method is defined that subtracts one date from another, and returns a duration, but there is no method for adding two dates together, since dates cannot be summed in any sensible way. However, there are methods for adding dates and durations which return dates.

Note that some addition and subtraction operations involving dates and instances of **`<year/month-duration>`** can cause errors where the result is a date that does not exist in the calendar. For example, adding one month to January 30th.

The table below summarizes the methods defined for each arithmetic operation, for different combinations of date and duration arguments, together with their return values.

Table 7.1 Methods defined for arithmetic operations on dates and durations

Op	Argument 1	Argument 2	Return value
+	**`<duration>`**	**`<duration>`**	**`<duration>`**
+	**`<year/month-duration>`**	**`<year/month-duration>`**	**`<year/month-duration>`**
+	**`<day/time-duration>`**	**`<day/time-duration>`**	**`<day/time-duration>`**
+	**`<date>`**	**`<duration>`**	**`<date>`**
+	**`<duration>`**	**`<date>`**	**`<date>`**
-	**`<duration>`**	**`<duration>`**	**`<duration>`**
-	**`<year/month-duration>`**	**`<year/month-duration>`**	**`<year/month-duration>`**
-	**`<day/time-duration>`**	**`<day/time-duration>`**	**`<day/time-duration>`**
-	**`<date>`**	**`<duration>`**	**`<date>`**
-	**`<date>`**	**`<date>`**	**`<day/time-duration>`**

Table 7.1 Methods defined for arithmetic operations on dates and durations

Op	Argument 1	Argument 2	Return value
`*`	`<duration>`	`<real>`	`<duration>`
`*`	`<real>`	`<duration>`	`<duration>`
`/`	`<duration>`	`<real>`	`<duration>`

7.4.4 Dealing with time-zones

The following functions return information about the time-zone that the host machine is in.

local-time-zone-name *Function*

`local-time-zone-name () =>` ***time-zone-name***

Returns the name of the time-zone that the local computer is in. The name is returned as a string (for example, **`"EST"`**).

local-time-zone-offset *Function*

`local-time-zone-offset () =>` ***time-zone-offset***

Returns the offset of the time-zone from Greenwich Mean Time, expressed as a number of minutes. A positive number represents an offset ahead of GMT, and a negative number represents an offset behind GMT. The return value is an instance of **`<integer>`** (for example, -300 represents the offset for EST, which is 5 hours behind GMT). The return value incorporates daylight savings time when necessary.

local-daylight-savings-time? *Function*

`local-daylight-savings-time? () =>` ***dst?***

Returns **`#t`** if the local computer is using Daylight Savings Time.

7.5 Extracting information from dates

A number of functions are available to return discrete pieces of information from a specified **<date>** object. These are useful to allow you to deconstruct a given date in order to retrieve useful information from it.

The most basic way to extract information from a date is to use the function **decode-date**.

decode-date *Function*

decode-date ***date*** **=>** ***year month day hours minutes seconds day-of-week time-zone-offset***

Decodes a **<date>** into its constituent parts. This function is the companion of **encode-date**, in that it takes a **<date>** object and returns all of its constituent parts. Note, however, that in contrast to **encode-date**, it does not return any millisecond component to the date, but it does return the day of the week of the specified date.

A number of other functions exist to extract individual components from a **<date>** object. Each of these functions is listed below. Each function takes a single argument, a **<date>** object, and returns the component of the date referred to in the function name. For example, **date-month** takes a **<date>** object as an argument, and returns the month that the date refers to.

```
date-year
date-month
date-day
date-day-of-week
date-hours
date-minutes
date-seconds
date-microseconds
date-time-zone-offset
```

For each function except **date-day-of-week**, the value returned is an instance of **<integer>**. The **date-day-of-week** function returns an object of type **<day-of-week>**. For more information, please refer to the reference entries of each function. See also the function **date-time-zone-offset-setter**, which allows you to set the time-zone offset of a **<date>** explicitly.

To return an ISO 8601 format date from a **`<date>`** object, use the function **`as-iso8601-string`**.

as-iso8601-string *Function*

`as-iso8601-string` ***date*** **`#key`** ***precision*** **`=>`** ***iso8601-string***

Returns an instance of **`<string>`** representing a date in ISO 8601 format. The *precision* keyword, if present, is an integer representing the number of decimal places to which the second should be specified in the result.

7.6 The DATE module

This section contains a reference entry for each item exported from the Date module.

= *Sealed method*

Summary — Compares two dates for equality.

Signature — ***date1*** **=** ***date2*** **=>** ***equal?***

Arguments

Arguments	*date1*	An instance of **`<date>`**.
	date2	An instance of **`<date>`**.
Values	*equal?*	An instance of **`<boolean>`**.

Description — This method lets you compare two dates to see if they are equal. Any differences in microseconds between *date1* and *date2* are ignored.

See also — **<**, page 128

= *Sealed method*

Summary

Compares two durations for equality.

Signature

duration1* = *duration2* => *equal?

Arguments

duration1	An instance of **`<duration>`**.
duration2	An instance of **`<duration>`**.

Values

equal?	An instance of **`<boolean>`**.

Description

This method lets you compare two durations to see if they are equal. If the durations are actually instances of **`<day/time-duration>`**, any differences in microseconds between *duration1* and *duration2* are ignored.

See also

<, page 129

< *Sealed method*

Summary

Determines whether one date is earlier than another.

Signature

date1* < *date2* => *before?

Arguments

date1	An instance of **`<date>`**.
date2	An instance of **`<date>`**.

Values

before?	An instance of **`<boolean>`**.

Description

This method determines if *date1* is earlier than *date2*. Any differences in microseconds between *date1* and *date2* are ignored.

See also

=, page 127

< *Sealed method*

Summary
: Determines whether one duration is less than another.

Signature
: ***duration1* < *duration2* => *less-than?***

Arguments

duration1	An instance of **`<duration>`**.
duration2	An instance of **`<duration>`**.

Values

less-than?	An instance of **`<boolean>`**.

Description
: This method determines if *duration1* is less than *duration2*. If the durations are actually instances of **`<day/time-duration>`**, any differences in microseconds between *duration1* and *duration2* are ignored.

See also
: **=**, page 128

+ *Sealed methods*

Summary
: Performs addition on specific combinations of dates and durations.

Signature
: **+ *arg1* *arg2* => *sum***

Arguments

arg1	An instance of **`<date>`** or **`<duration>`**. See description for details.
arg2	An instance of **`<date>`** or **`<duration>`**. See description for details.

Values

sum	An instance of **`<date>`** or **`<duration>`**. See description for details.

Description
: A number of methods are defined for the **+** generic function to allow summing of various combinations of dates and

durations. Note that there is not a method defined for every possible combination of date and duration. Specifically, you cannot sum different types of duration, and you cannot sum two dates. The return value can be either a date or a duration, depending on the arguments supplied. The table below lists the methods that are defined on **+**.

Table 7.2 Methods defined for addition of dates and durations

arg1	*arg2*	*sum*
`<duration>`	**`<duration>`**	**`<duration>`**
`<year/month-duration>`	**`<year/month-duration>`**	**`<year/month-duration>`**
`<day/time-duration>`	**`<day/time-duration>`**	**`<day/time-duration>`**
`<date>`	**`<duration>`**	**`<date>`**
`<duration>`	**`<date>`**	**`<date>`**

See also **-**, page 130

*****, page 131

/, page 132

- *Sealed methods*

Summary Performs subtraction on specific combinations of dates and durations.

Signature **-** ***arg1 arg2*** **=>** ***diff***

Arguments *arg1* An instance of **`<date>`** or **`<duration>`**. See description for details.

arg2 — An instance of **`<duration>`**, or an instance of **`<date>`** if *arg1* is a **`<date>`**. See description for details.

Values — *diff* — An instance of **`<date>`** or **`<duration>`**. See description for details.

Description — A number of methods are defined for the **`-`** generic function to allow subtraction of various combinations of dates and durations. Note that there is not a method defined for every possible combination of date and duration. Specifically, you cannot subtract a date from a duration, and you cannot subtract different types of duration. The return value can be either a date or a duration, depending on the arguments supplied. The table below lists the methods that are defined on **`-`**.

Table 7.3 Methods defined for subtraction of dates and durations

arg1	*arg2*	*diff*
`<year/month-duration>`	**`<year/month-duration>`**	**`<year/month-duration>`**
`<day/time-duration>`	**`<day/time-duration>`**	**`<day/time-duration>`**
`<date>`	**`<duration>`**	**`<date>`**
`<date>`	**`<date>`**	**`<day/time-duration>`**

See also — **`+`**, page 129

`*`, page 131

`/`, page 132

`*` *Sealed methods*

Summary — Multiplies a duration by a scale factor.

Signature **`*`** ***duration scale*** **`=>`** ***new-duration***
`*` ***scale duration*** **`=>`** ***new-duration***

Arguments

duration	An instance of **`<duration>`**.
scale	An instance of **`<real>`**.

Note: These arguments can be expressed in any order.

Values

new-duration	An instance of **`<date>`** or **`<duration>`**. See description for details.

Description Multiples a duration by a scale factor and returns the result. Note that the arguments can be expressed in any order: methods are defined such that the duration can be placed first or second in the list of arguments.

See also **`+`**, page 129

`-`, page 130

`/`, page 132

/ *Sealed methods*

Summary Divides a duration by a scale factor

Signature **`/`** ***duration scale*** **`=>`** ***new-duration***

Arguments

duration	An instance of **`<duration>`**.
scale	An instance of **`<real>`**.

Values

new-duration	An instance of **`<date>`** or **`<duration>`**. See description for details.

Description A number of methods are defined for the + generic function to allow summing of various combinations of dates and durations. Note that there is not a method defined for every possible combination of date and duration. Specifically, you cannot sum different types of duration, and you cannot sum two dates. The return value can be either a date or a duration, depending on the arguments supplied. The table below lists the methods that are defined on +.

See also

+, page 129

-, page 130

*****, page 131

as-iso8601-string *Function*

Summary

Returns a string representation of a date, conforming to the ISO 8601 standard.

Signature

as-iso8601-string ***date*** **#key** ***precision*** **=>** ***iso8601-string***

Arguments

date	An instance of **<date>**.
precision	An instance of **<integer>**. Default value: 0.

Values

iso8601-string	An instance of **<string>**.

Description

Returns a string representation of *date* using the format identified by International Standard ISO 8601 (for example, **"19960418T210634Z"**). If *precision* is non-zero, the specified number of digits of a fraction of a second are included in the string (for example, **"19960418T210634.0034Z"**).

The returned string always expresses the time in Greenwich Mean Time. The *iso8601-string* init-keyword for **<date>**, however, accepts ISO 8601 strings with other time zone specifications.

See also

<date>, page 134

current-date *Function*

Summary Returns a date object representing the current date and time.

Signature **`current-date () =>`** ***date***

Arguments None.

Values	*date*	An instance of **`<date>`**.

Description Returns *date* for the current date and time.

<date> *Sealed class*

Summary The class of objects representing dates.

Superclasses **`<number>`**

Init-keywords	*iso8601-string*	An instance of **`false-or(<string>)`**. Default value: **`#f`**.
	year	An instance of **`limited(<integer>, min: 1800, max: 2199)`**.
	month	An instance of **`limited(<integer>, min: 1, max: 12)`**.
	day	An instance of **`limited(<integer>, min: 1, max: 31)`**.
	hours	An instance of **`limited(<integer>, min: 0, max: 23)`**. Default value: 0.
	minutes	An instance of **`limited(<integer>, min: 0, max: 59)`**. Default value: 0.
	seconds	An instance of **`limited(<integer>, min: 0, max: 59)`**. Default value: 0.

microseconds	An instance of **`limited(<integer>, min: 0, max: 999999)`**. Default value: 0.
time-zone-offset	An instance of **`<integer>`**. Default value: 0.

Description

Represents a date and time between 1 Jan 1800 00:00:00 and 31 Dec 2199 23:59:59, Greenwich Mean Time (GMT).

A **`<date>`** can be specified to microsecond precision and includes a time zone indication.

If supplied, the **`iso8601-string:`** init-keyword completely specifies the value of the **`<date>`**. Otherwise, the **`year:`**, **`month:`**, and **`day:`** init-keywords must be supplied. Note that, although you can supply ISO 8601 strings that represent any time zone specification, the related function **`as-iso8601-string`** always returns an ISO 8601 string representing a time in Greenwich Mean Time.

For the *time-zone-offset* init-keyword, a positive number represents an offset ahead of GMT, in minutes, and a negative number represents an offset behind GMT. The value returned is an instance of **`<integer>`** (for example, -300 represents the offset for EST, which is 5 hours behind GMT).

Operations

`= < + - as-iso8601-string current-date date-day date-day-of-week date-hours date-microseconds date-minutes date-month date-seconds date-time-zone-offset date-time-zone-offset-setter date-year decode-date`

See also

`as-iso8601-string`, page 133

`<day-of-week>`, page 143

date-day *Function*

Summary

Returns the day of the month component of a specified date.

Signature **`date-day`** ***date*** **`=>`** ***day***

Arguments *date* An instance of **`<date>`**.

Values *day* An instance of **`<integer>`**.

Description Returns the day of the month component of the specified *date*. For example, if passed a **`<date>`** that represented 16:36 on the 20th June, 1997, **`date-day`** returns the value 20.

See also **`decode-date`**, page 144

`date-month`, page 139

`date-year`, page 142

`date-hours`, page 137

`date-minutes`, page 139

`date-seconds`, page 140

`date-microseconds`, page 138

`date-time-zone-offset`, page 141

`date-day-of-week`, page 136

date-day-of-week *Function*

Summary Returns the day of the week of a specified date.

Signature **`date-day-of-week`** ***date*** **`=>`** ***day-of-week***

Arguments *date* An instance of **`<date>`**.

Values *day-of-week* An object of type **`<day-of-week>`**.

Description Returns the day of the week of the specified *date*.

See also

`decode-date`, page 144

`date-month`, page 139

`date-year`, page 142

`date-hours`, page 137

`date-minutes`, page 139

`date-seconds`, page 140

`date-microseconds`, page 138

`date-time-zone-offset`, page 141

`date-day`, page 135

`<day-of-week>`, page 143

date-hours *Function*

Summary

Returns the hour component of a specified date.

Signature

`date-hours` ***date*** **`=>`** ***hour***

Arguments

date	An instance of **`<date>`**.

Values

hour	An instance of **`<integer>`**.

Description

Returns the hour component of the specified *date*. This component is always expressed in 24 hour format.

See also

`decode-date`, page 144

`date-month`, page 139

`date-day`, page 135

`date-year`, page 142

`date-minutes`, page 139

`date-seconds`, page 140

date-microseconds, page 138

date-time-zone-offset, page 141

date-day-of-week, page 136

date-microseconds *Function*

Summary

Returns the microseconds component of a specified date.

Signature

date-microseconds ***date*** **=>** ***microseconds***

Arguments

date	An instance of **<date>**.

Values

microseconds	An instance of **<integer>**.

Description

Returns the microseconds component of the specified *date*. Note that this does *not* return the entire date object, represented as a number of microseconds; it returns any value assigned to the **microseconds:** init-keyword when the **<date>** object was created.

See also

decode-date, page 144

date-month, page 139

date-day, page 135

date-hours, page 137

date-minutes, page 139

date-seconds, page 140

date-year, page 142

date-time-zone-offset, page 141

date-day-of-week, page 136

date-minutes *Function*

Summary Returns the minutes component of a specified date.

Signature **`date-minutes`** ***date*** **`=>`** ***minutes***

Arguments *date* An instance of **`<date>`**.

Values *minutes* An instance of **`<integer>`**.

Description Returns the minutes component of the specified *date*.

See also **`decode-date`**, page 144

`date-month`, page 139

`date-day`, page 135

`date-hours`, page 137

`date-year`, page 142

`date-seconds`, page 140

`date-microseconds`, page 138

`date-time-zone-offset`, page 141

`date-day-of-week`, page 136

date-month *Function*

Summary Returns the month of a specified date.

Signature **`date-month`** ***date*** **`=>`** ***month***

Arguments *date* An instance of **`<date>`**.

Values *month* An instance of **`<integer>`**.

Description Returns the month of the specified *date*.

See also

decode-date, page 144

date-year, page 142

date-day, page 135

date-hours, page 137

date-minutes, page 139

date-seconds, page 140

date-microseconds, page 138

date-time-zone-offset, page 141

date-day-of-week, page 136

date-seconds *Function*

Summary
Returns the seconds component of a specified date.

Signature
date-seconds ***date*** **=>** ***seconds***

Arguments

date	An instance of **<date>**.

Values

seconds	An instance of **<integer>**.

Description
Returns the seconds component of the specified *date*. Note that this does *not* return the entire date object, represented as a number of seconds; it returns any value assigned to the **seconds:** init-keyword when the **<date>** object was created.

See also

decode-date, page 144

date-month, page 139

date-day, page 135

date-hours, page 137

date-minutes, page 139

date-year, page 142

date-microseconds, page 138

date-time-zone-offset, page 141

date-day-of-week, page 136

date-time-zone-offset *Function*

Summary Returns the time zone offset of a specified date.

Signature `date-time-zone-offset` ***date*** `=>` ***time-zone-offset***

Arguments *date* An instance of **`<date>`**.

Values *time-zone-offset* An instance of **`<integer>`**.

Description Returns the time zone offset of the specified *date*. The values of the other components of *date* reflect this time zone.

A positive number represents an offset ahead of GMT, in minutes, and a negative number represents an offset behind GMT. The value returned is an instance of **`<integer>`** (for example, -300 represents the offset for EST, which is 5 hours behind GMT).

See also **decode-date**, page 144

date-month, page 139

date-day, page 135

date-hours, page 137

date-minutes, page 139

date-seconds, page 140

date-year, page 142

date-microseconds, page 138

date-time-zone-offset-setter, page 142

date-day-of-week, page 136

date-time-zone-offset-setter *Function*

Summary

Change the time zone offset of a specified date, while maintaining the same point in time.

Signature

date-time-zone-offset-setter ***new-time-zone-offset date***
=> ***new-time-zone-offset***

Arguments

new-time-zone-offset

An instance of **<integer>**.

date An instance of **<date>**.

Values

new-time-zone-offset

An instance of **<integer>**.

Description

Changes the time zone offset of *date* without changing the actual point in time identified by the *date*. The values of the other components of *date* are adjusted to reflect the new time zone.

The *new-time-zone-offset* argument should represent the offset from GMT, in minutes. Thus, if you wish to specify a new offset representing EST, which is 5 hours behind GMT, *new-time-zone-offset* should have the value -300.

See also

date-time-zone-offset, page 141

date-year *Function*

Summary

Returns the year of a specified date.

Signature **`date-year`** ***date*** **`=>`** ***year***

Arguments	*date*	An instance of **`<date>`**.
Values	*year*	An instance of **`<integer>`**.

Description Returns the year of the specified *date*.

See also **`decode-date`**, page 144

`date-month`, page 139

`date-day`, page 135

`date-hours`, page 137

`date-minutes`, page 139

`date-seconds`, page 140

`date-microseconds`, page 138

`date-time-zone-offset`, page 141

`date-day-of-week`, page 136

<day-of-week> *Type*

Summary The days of the week.

Equivalent

```
one-of(#"Sunday", #"Monday", #"Tuesday", #"Wednesday",
       #"Thursday", #"Friday", #"Saturday")
```

Supertypes None.

Init-keywords None.

Description The days of the week. This is the type of the return value of the **`date-day-of-week`** function.

Operations **`date-day-of-week`**

See also **`date-day-of-week`**, page 136

<day/time-duration> *Sealed class*

Summary The class of objects representing durations in units of microseconds.

Superclasses **`<duration>`**

Init-keywords

days	An instance of **`<integer>`**.
hours	An instance of **`<integer>`**. Default value: 0.
minutes	An instance of **`<integer>`**. Default value: 0.
seconds	An instance of **`<integer>`**. Default value: 0.
microseconds	An instance of **`<integer>`**. Default value: 0.

Description The class of objects representing durations in units of microseconds. It is a subclass of **`<duration>`**.

Use this class to represent a number of days and fractions thereof. If you need to represent durations in calendar units of months or years, use **`<year/month-duration>`** instead.

Operations **`< + - decode-duration encode-day/time-duration`**

See also **`<duration>`**, page 147

`<year/month-duration>`, page 152

decode-date *Function*

Summary Returns the date and time stored in a date object.

Signature **`decode-date`** ***date***
`=>` ***year month day hours minutes seconds day-of-week time-zone-offset***

Arguments	*date*	An instance of **`<date>`**.
Values	*year*	An instance of **`<integer>`**.
	month	An instance of **`<integer>`**.
	day	An instance of **`<integer>`**.
	hours	An instance of **`<integer>`**.
	minutes	An instance of **`<integer>`**.
	seconds	An instance of **`<integer>`**.
	day-of-week	An instance of **`<day-of-week>`**.
	time-zone-offset	An instance of **`<integer>`**.

Description Returns the date and time stored in *date*. Note that it does not return the millisecond component of a **`<date>`**, but it does return the appropriate **`<day-of-week>`**.

See also **`encode-date`**, page 148

decode-duration *Sealed generic function*

Summary Decodes a duration into its constituent parts.

Signature **`decode-duration`** ***duration*** **`=> #rest`** ***components***

Arguments	*duration*	An instance of **`<duration>`**.
Values	*components*	Instances of **`<integer>`**.

Description Decodes an instance of **`<duration>`** into its constituent parts. There are methods for this generic function that specialize on **`<year/month-duration>`** and **`<day/time-duration>`** respectively, as described below.

See also **`decode-duration`**, page 146

decode-duration, page 146

decode-duration *Sealed method*

Summary Decodes a day / time duration into its constituent parts.

Signature **decode-duration** ***duration***
=> ***days hours minutes seconds microseconds***

Arguments

duration	An instance of **<day/time-duration>**.

Values

days	An instance of **<integer>**.
hours	An instance of **<integer>**.
minutes	An instance of **<integer>**.
seconds	An instance of **<integer>**.
microseconds	An instance of **<integer>**.

Description Decodes an instance of **<day/time-duration>** into its constituent parts.

See also **decode-duration**, page 145

decode-duration, page 146

encode-day/time-duration, page 149

decode-duration *Sealed method*

Summary Decodes a year / month duration into its constituent parts.

Signature **decode-duration** ***duration*** **=>** ***years months***

Arguments

duration	An instance of **<year/month-duration>**.

Values

years	An instance of **<integer>**.

months An instance of **`<integer>`**.

Description Decodes an instance of **`<year/month-duration>`** into its constituent parts.

See also **`decode-duration`**, page 145

`decode-duration`, page 146

`encode-year/month-duration`, page 149

<duration> *Sealed abstract instantiable class*

Summary The class of objects representing durations.

Superclasses **`<number>`**

Init-keywords *iso8601-string*An instance of **`false-or(<string>)`**. Default value: **`#f`**.

year An instance of **`limited(<integer>, min: 1800, max: 2199)`**.

month An instance of **`limited(<integer>, min: 1, max: 12)`**.

day An instance of **`limited(<integer>, min: 1, max: 31)`**.

hours An instance of **`limited(<integer>, min: 0, max: 23)`**. Default value: 0.

minutes An instance of **`limited(<integer>, min: 0, max: 59)`**. Default value: 0.

seconds An instance of **`limited(<integer>, min: 0, max: 59)`**. Default value: 0.

microseconds An instance of **`limited(<integer>, min: 0, max: 999999)`**. Default value: 0.

time-zone-offset An instance of **`<integer>`**. Default value: 0.

Description

This class is the used to represent durations. It is a subclass of **`<number>`**, and it has two subclasses.

Operations

`= < + - * /`

See also

`<day/time-duration>`, page 144

`<year/month-duration>`, page 152

encode-date *Function*

Summary

Creates a date object for the specified date and time.

Signature

`encode-date` ***year month day hours minutes seconds***
`#key` ***microseconds time-zone-offset*** **`=>`** ***date***

Arguments

year	An instance of **`<integer>`**.
month	An instance of **`<integer>`**.
day	An instance of **`<integer>`**.
hours	An instance of **`<integer>`**.
minutes	An instance of **`<integer>`**.
seconds	An instance of **`<integer>`**.
microseconds	An instance of **`<integer>`**. Default value: 0.
time-zone-offset	An instance of **`<integer>`**. Default value: **`local-time-zone-offset()`**.

Values

date	An instance of **`<date>`**.

Description

Creates a **`<date>`** object for the specified date and time.

See also

`decode-date`, page 144

`local-time-zone-offset`, page 150

`make date-class`, page 151

encode-day/time-duration *Function*

Summary	Creates a day / time duration from a set of integer values.	
Signature	**`encode-day/time-duration`** ***days hours minutes seconds microseconds*** **`=>`** ***duration***	
Arguments	*days*	An instance of **`<integer>`**.
	hours	An instance of **`<integer>`**.
	minutes	An instance of **`<integer>`**.
	seconds	An instance of **`<integer>`**.
	microseconds	An instance of **`<integer>`**.
Values	*duration*	An instance of **`<day/time-duration>`**.
Description	Creates an instance of **`<day/time-duration>`**.	
See also	**`decode-duration`**, page 146 **`encode-year/month-duration`**, page 149	

encode-year/month-duration *Function*

Summary	Creates a year / month duration from a set of integer values.	
Signature	**`encode-year/month-duration`** ***years months*** **`=>`** ***duration***	
Arguments	*years*	An instance of **`<integer>`**.
	months	An instance of **`<integer>`**.
Values	*duration*	An instance of **`<year/month-duration>`**.
Description	Creates an instance of **`<year/month-duration>`**.	
See also	**`decode-duration`**, page 146	

encode-day/time-duration, page 149

local-daylight-savings-time? *Function*

Summary: Checks whether the local machine is using Daylight Savings Time.

Signature: **`local-daylight-savings-time? () =>`** ***dst?***

Arguments: None.

Values: *dst?* An instance of **`<boolean>`**.

Description: Returns **`#t`** if the local machine is using Daylight Savings Time, and **`#f`** otherwise.

local-time-zone-name *Function*

Summary: Returns the time zone name in use by the local machine.

Signature: **`local-time-zone-name () =>`** ***time-zone-name***

Arguments: None.

Values: *time-zone-name* An instance of **`<string>`**.

Description: Returns the time zone name in use by the local machine, if available, or a string of the form **`+/-HHMM`** if the time zone name is unknown.

local-time-zone-offset *Function*

Summary: Returns the offset of the time-zone from Greenwich Mean Time, expressed as a number of minutes.

Signature **`local-time-zone-offset () =>`** ***time-zone-offset***

Arguments None.

Values *time-zone-offset* An instance of **`<integer>`**.

Description Returns the offset of the time-zone from Greenwich Mean Time, expressed as a number of minutes. A positive number represents an offset ahead of GMT, and a negative number represents an offset behind GMT. The return value is an instance of **`<integer>`** (for example, -300 represents the offset for EST, which is 5 hours behind GMT). The return value incorporates daylight savings time when necessary.

make *date-class* *G.f. method*

Summary Creates an instance of the **`<date>`** class.

Signature **`make`** ***date-class*** **`#key`** ***iso8601-string year month day hours minutes seconds microseconds time-zone-offset***
`=>` ***date-instance***

Arguments

date-class	The class **`<date>`**.
iso8601-string	An instance of **`false-or(<string>)`**. Default value: **`#f`**.
year	An instance of **`limited(<integer>, min: 1800, max: 2199)`**.
month	An instance of **`limited(<integer>, min: 1, max: 12)`**.
day	An instance of **`limited(<integer>, min: 1, max: 31)`**.
hours	An instance of **`limited(<integer>, min: 0, max: 23)`**. Default value: 0.

minutes An instance of **`limited(<integer>, min: 0, max: 59)`**. Default value: 0.

seconds An instance of **`limited(<integer>, min: 0, max: 59)`**. Default value: 0.

microseconds An instance of **`limited(<integer>, min: 0, max: 999999)`**. Default value: 0.

time-zone-offset An instance of **`<integer>`**. Default value: 0.

Values *date-instance* An instance of **`<date>`**.

Description Creates an instance of **`<date>`**.

The make method on **`<date>`** takes the same keywords as the **`<date>`** class.

Note: The iso8601-string keyword accepts a richer subset of the ISO 8601 specification than is produced by the **`as-iso8601-string`** function.

Example

```
make (<date>, iso8601-string: "19970717T1148-0400")
```

See also **`<date>`**, page 134

`encode-date`, page 148

<year/month-duration> *Sealed class*

Summary The class of objects representing durations with a coarse resolution.

Superclasses **`<duration>`**

Init-keywords *year* An instance of **`<integer>`**.

month An instance of **`<integer>`**.

Description

The class of objects representing durations in units of calendar years and months. It is a subclass of **`<duration>`**.

Use this class to represent a number of calendar years and months. If you need to represent durations in units of days or fractions thereof (to microsecond resolution), use **`<day/time-duration>`** instead.

Operations

`< + - decode-duration encode-year/month-duration`

See also

`<day/time-duration>`, page 144

`<duration>`, page 147

8

The File-System Module

8.1 Introduction

The File-System module is part of the System library and provides a generic interface to the file system of the local machine. Remotely mounted file systems are also accessible using this module.

This chapter describes the functions and types that the File-System module contains.

8.2 Types specific to file system operations

The File-System module contains a number of types specifically designed for use by interfaces in the module.

Firstly, the type **`<file-type>`** represents the types of entity that may be present on a given file system. Three entities are allowed: a file, a directory, and a link to a file or directory located elsewhere in the file system. Together, these represent any entity that can be placed on a file system mounted on the local machine. The **`<file-type>`** type is defined as

```
one-of(#"file", #"directory", #"link")
```

The type **`<pathname>`** represents the set of classes that may be used to represent pathnames that indicate entities on the file system. It is defined as a type alias of **`<string>`**.

Lastly, the type **`<copy/rename-disposition>`** represents the possible values of the **`if-exists:`** keyword to the functions **`rename-file`** and **`copy-file`** described in Section 8.3. It is defined as

```
one-of (#"signal", #"replace")
```

If the value of the keyword for either function is **`#"signal"`** (the default for both functions), then you are prompted before an existing file is overwritten as a result of a copy or rename operation. If **`#"replace"`**, then an existing file is overwritten without prompting.

8.3 Manipulating files

The File-System module contains a number of interfaces that let you perform standard file management operations on files already resident on the filesystem. You can rename, copy, or delete any file, and you can set any available properties for the file.

delete-file *Function*

`delete-file` ***file*** **`=> ()`**

Use this function to delete a specified file. If *file* is actually a link, then the link is deleted, rather than the file it points to.

rename-file *Function*

copy-file *Function*

`rename-file` ***old-file new-file*** **`#key`** ***if-exists*** **`=> ()`**

`copy-file` ***old-file new-file*** **`#key`** ***if-exists*** **`=> ()`**

Renames or copies *old-file* to *new-file*.

For both functions, if *new-file* already exists, then the behavior depends on the value of *if-exists*, which is an instance of **`<copy/rename-disposition>`**. The default behavior is to prompt you before overwriting a file.

file-property-setter *Sealed generic function*

`file-property-setter` ***new-value file key*** **`=>`** ***new-value***

Sets a property of *file* to *new-value*. The property that is set is specified by the value of *key*, which must be one of the following:

`#"author" #"size" #"creation-date" #"access-date" #"modification-date" #"write-date" #"readable?" #"writeable?" #"executable?"`

The type of *new-value* (and hence the type of the return value of **`file-property-setter`**), is determined by the value of *key*. For example, if *key* is **`#"readable?"`**, then *new-value* should be an instance of **`<boolean>`**. For full details, see **`file-property-setter`**, page 168.

8.4 Manipulating directories

The File-System module contains a number of interfaces that let you create and delete directories. These can be used in conjunction with the file manipulation operations described in Section 8.3 on page 156 to perform file management tasks at any position in the file system.

create-directory *Function*

`create-directory` ***parent name*** **`=>`** ***directory***

Creates a new directory *name* in the directory *parent*. The full pathname of the directory is returned, and you can use this return value in conjunction with **`concatenate`** to create pathnames of any entities that you create in the new directory.

delete-directory *Function*

`delete-directory` ***directory*** **`=> ()`**

Deletes a directory specified by *directory*. Whether or not the directory needs to be empty before it can be deleted is determined by the platform on which you are running.

ensure-directories-exist *Function*

`ensure-directories-exist` ***file*** **`=>`** ***created?***

Use this function when you want to guarantee that a particular directory structure has been created on disk. It ensures that the individual directories that constitute the pathname specified by *file* exist, and creates any that do not. If **`ensure-directories-exist`** actually creates any directories, then **`#t`** is returned.

do-directory *Function*

`do-directory` ***function directory*** **`=> ()`**

Performs *function* once for every item in the specified *directory*. The *function* must have the following signature:

function directory name type **`=> ()`**

where *directory* is the name of the directory specified to **`do-directory`**, *name* is an instance of **`<byte-string>`**, and *type* is an instance of **`<file-type>`**.

Within *function*, you can concatenate the values of *directory* and *name* to generate a **`<pathname>`** suitable for use by the other functions in the File-system module.

working-directory-setter *Function*

`working-directory-setter` ***directory*** **`=>`** ***directory***

Sets the working directory for the current process.

8.5 Finding out file system information

A number of functions return environment information regarding the directory structure of the file system. Each function takes no arguments, and returns a pathname or list of pathnames. The return values can be use in conjunction with other functions to perform file-based operations relative to the directories involved.

home-directory *Function*

`home-directory () =>` ***home-directory***

Returns the **`<pathname>`** of the current value of the home directory. You can use the return value of **`home-directory`** in conjunction with **`concatenate`** to specify the pathname of any entity in the home directory.

root-directories *Function*

`root-directories () =>` ***roots***

Returns a sequence containing the pathnames of the root directories of all the file systems connected to the local machine.

temp-directory *Function*

`temp-directory () =>` ***temp-directory***

Returns the **`<pathname>`** of the temporary directory in use on the local machine. If no temporary directory is defined, this function returns false. You can use the return value of **`temp-directory`** in conjunction with **`concatenate`** to specify pathnames of entities in the temporary directory.

working-directory *Function*

`working-directory () =>` ***working-directory***

Returns the **`<pathname>`** of the current working directory in the current process on the local machine. You can use the return value of **`working-`**

directory in conjunction with **concatenate** to specify pathnames of entities in the working directory.

8.6 Finding out file information

Several interfaces in the File-System module allow you to interrogate files for information. You can find out whether a file exists, what its name is, or which directory it resides in, and you can find the current properties of the file.

file-exists? *Function*

file-exists? ***file*** **=>** ***exists?***

Returns true if *file* exists, false otherwise. If *file* is actually a link, then **file-exists** checks the target of the link, and returns true if the target exists.

file-properties *Function*

file-properties ***file*** **=>** ***properties***

Returns all the properties of the specified file. The properties are returned as a concrete subclass of **<explicit-key-collection>**.

file-property *Sealed generic function*

file-property ***file key*** **=>** ***property***

Returns a particular property of the specified file. The property returned is dependent on the value of *key*, and as such, may be of a number of types. For more information about the possible values of *key*, see **file-property-setter**, page 168.

file-type *Function*

file-type ***file*** **=>** ***file-type***

Returns the file type of the entity specified by *file*, as an instance of **<file-type>**. A given entity can either be a file, a directory, or a link to a file or directory.

8.7 The FILE-SYSTEM module

This section contains a reference entry for each item included in the File-System module.

copy-file *Function*

Summary
: Creates a copy of a file.

Signature
: **`copy-file`** ***old-file new-file*** **`#key`** ***if-exists*** **`=> ()`**

Arguments

old-file	An instance of **`<pathname>`**.
new-file	An instance of **`<pathname>`**.
if-exists	An instance of **`<copy/rename-disposition>`**. Default value: **`#"signal"`**.

Values
: None

Description
: Copies *old-file* to *new-file*. If *new-file* already exists, the action of this function is controlled by the value of *if-exists*. The default is to prompt you before overwriting an existing file.

See also
: **`<copy/rename-disposition>`**, page 161

`rename-file`, page 172

<copy/rename-disposition> *Type*

Summary
: The type that represents possible actions when overwriting existing files.

Equivalent
: **`one-of(#"signal", #"replace")`**

Supertypes
: None.

Init-keywords None.

Description This type represents the acceptable values for the **`if-exists:`** argument to the **`copy-file`** and **`rename-file`** functions. Only two values are acceptable:

- If **`#"signal"`** is used, then you are warned before a file is overwritten during a copy or move operation.
- If **`#"replace"`** is used, then you are not warned before a file is overwritten during a copy or move operation.

Operations **`copy-file`** **`rename-file`**

See also **`copy-file`**, page 161

`rename-file`, page 172

create-directory *Function*

Summary Creates a new directory in the specified parent directory.

Signature **`create-directory`** ***parent name*** **`=>`** ***directory***

Arguments *parent* An instance of **`<pathname>`**.

name An instance of **`<string>`**.

Values *directory* An instance of **`<pathname>`**.

Description Creates *directory* in the specified *parent* directory. The return value of this function can be used with **`concatenate`** to create pathnames of entities in the new directory.

See also **`delete-directory`**, page 163

delete-directory *Function*

Summary Deletes the specified directory.

Signature **`delete-directory`** ***directory*** **`=> ()`**

Arguments *directory* An instance of **`<pathname>`**.

Values None.

Description Deletes the specified directory. Whether or not the directory must be empty before it can be deleted is platform dependent.

See also **`create-directory`**, page 162

`delete-file`, page 163

delete-file *Function*

Summary Deletes the specified file system entity.

Signature **`delete-file`** ***file*** **`=> ()`**

Arguments *file* An instance of **`<pathname>`**.

Values None.

Description Deletes the file system entity specified by *file*. If *file* refers to a link, the link is removed, but the actual file that the link points to is not removed.

do-directory *Function*

Summary

Executes the supplied function once for each entity in the specified directory.

Signature

do-directory ***function directory*** **=> ()**

Arguments

function	An instance of **<function>**.
directory	An instance of **<pathname>**.

Values

None.

Description

Executes *function* once for each entity in *directory.*

The signature of *function* is

function directory name type **=> ()**

where *directory* is an instance of **<pathname>**, *name* is an instance of **<byte-string>**, and *type* is an instance of **<file-type>**.

Within *function*, the values of *directory* and *name* can be concatenated to generate a **<pathname>** suitable for use by the other functions in the module.

The following calls are equivalent

```
do-directory(my-function, "C:\USERS\JOHN\FOO.TEXT")
do-directory(my-function, "C:\USERS\JOHN\")
```

as they both operate on the contents of **C:\USERS\JOHN**. The call

```
do-directory(my-function, "C:\USERS\JOHN")
```

is not equivalent as it will operate on the contents of **C:\USERS**.

ensure-directories-exist *Function*

Summary

Ensures that all the directories in the pathname leading to a file exist, creating any that do not, as needed.

Signature

`ensure-directories-exist` ***file*** **`=>`** ***created?***

Arguments

file	An instance of **`<pathname>`**.

Values

created?	An instance of **`<boolean>`**.

Description

Ensures that all the directories in the pathname leading to a file exist, creating any that do not, as needed. The return value indicates whether or not any directory was created.

The following calls are equivalent

```
ensure-directories-exist("C:\USERS\JOHN\FOO.TEXT")
ensure-directories-exist("C:\USERS\JOHN\")
```

as they will both create the directories **`USERS`** and **`JOHN`** if needed. The call

```
ensure-directories-exist("C:\USERS\JOHN")
```

is not equivalent as it will only create **`USERS`** if needed.

Example

```
ensure-directories-exist("C:\USERS\JOHN\FOO.TEXT")
```

See also

`create-directory`, page 162

file-exists? *Function*

Summary

Returns **`#t`** if the specified file exists.

Signature

`file-exists?` ***file*** **`=>`** ***exists?***

Arguments

file	An instance of **`<pathname>`**.

Values *exists?* An instance of **`<boolean>`**.

Description Returns **`#t`** if *file* exists. If it refers to a link, the target of the link is checked.

file-properties *Function*

Summary Returns all the properties of a file system entity.

Signature **`file-properties`** ***file*** **`=>`** ***properties***

Arguments *file* An instance of **`<pathname>`**.

Values *properties* An instance of a concrete subclass of **`<explicit-key-collection>`**.

Description Returns all the properties of *file*. The keys to the properties collection are the same as those use by **`file-property`**, above.

Example

```
file-properties() [#"size"]
```

See also **`file-property`**, page 166

`file-property-setter`, page 168

file-property *Sealed generic function*

Summary Returns the specified property of a file system entity.

Signature **`file-property`** ***file*** **`#key`** ***key*** **`=>`** ***property***

Arguments *file* An instance of **`<pathname>`**.

key One of **`#"author"`**, **`#"size"`**, **`#"creation-date"`**, **`#"access-date"`**, **`#"modification-date"`**, **`#"write-date"`**, **`#"readable?"`**, **`#"writeable?"`**, **`#"executable?"`**.

Values *property* The value of the property specified by *key*. The type of the value returned depends on the value of *key*: see the description for details.

Description Returns the property of *file* specified by *key*. The value returned depends on the value of *key*, as shown in Table 8.1.

Table 8.1 Return value types of **`file-property`**

Value of *key*	Type of return value
`#"author"`	**`false-or(<string>)`**
`#"size"`	**`<integer>`**
`#"creation-date"`	**`<date>`**
`#"access-date"`	**`<date>`**
`#"modification-date"`	**`<date>`**
`#"write-date"`	**`<date>`**
`#"readable?"`	**`<boolean>`**
`#"writeable?"`	**`<boolean>`**
`#"executable?"`	**`<boolean>`**

Not all platforms implement all of the above keys. Some platforms may support additional keys. The **`#"author"`** key is supported on all platforms but may return **`#f`** if it is not meaningful on a given platform. The **`#"modification-date"`** and **#"write-date"** keys are identical. Use of an unsupported key signals an error.

All keys listed above are implemented by Win32, though note that **#"author"** always returns **#f**.

See also

file-property-setter, page 168

file-properties, page 166

file-property-setter *Sealed generic function*

Summary

Sets the specified property of a file system entity to a given value.

Signature

file-property-setter ***new-value file key*** **=>** ***new-value***

Arguments

new-value	The type of this depends on the value of *key*. See the description for details.
file	An instance of **<pathname>**.
key	One of **#"author"**, **#"size"**, **#"creation-date"**, **#"access-date"**, **#"modification-date"**, **#"write-date"**, **#"readable?"**, **#"writeable?"**, **#"executable?"**.

Values

new-value	The type of this depends on the value of *key*. See the description for details.

Description

Sets the property of *file* specified by *key* to *new-value*. The type of *new-value* depends on the property specified by key, as shown in Table 8.2 below.

Table 8.2 Return value types of **file-property-setter**

Value of *key*	Type of *new-value*
#"author"	**false-or(<string>)**
#"size"	**<integer>**
#"creation-date"	**<date>**
#"access-date"	**<date>**
#"modification-date"	**<date>**
#"write-date"	**<date>**
#"readable?"	**<boolean>**
#"writeable?"	**<boolean>**
#"executable?"	**<boolean>**

Note that **file-property-setter** returns the value that was set, and so return values have the same types as specified values, depending on the value of *key*.

Not all platforms implement all of the above keys. Some platforms may support additional keys. The **#"modification-date"** and **#"write-date"** keys are identical. Use of an unsupported key signals an error.

The only property that can be set on Win32 is **#"writeable?"**.

See also

file-property, page 166

file-properties, page 166

<file-system-error> *Error*

Summary

Error type signaled when any other functions in the File-System module signal an error.

Superclasses

`<error>` and **`<simple-condition>`**

Init-keywords

Description

Signalled when one of the file system functions triggers an error, such as a permissions error when trying to delete or rename a file.

Operations

None.

file-type *Function*

Summary

Returns the type of the specified file system entity.

Signature

`file-type` ***file*** **`=>`** ***file-type***

Arguments	*file*	An instance of **`<pathname>`**.
Values	*file-type*	An instance of **`<file-type>`**.

Description

Returns the type of *file*, the specified file system entity. A file system entity can either be a file, a directory, or a link to another file or directory.

<file-type> *Type*

Summary

The type representing all possible types of a file system entity.

Equivalent

`one-of(#"file", #"directory", #"link")`

Supertypes None.

Init-keywords None.

Description The type representing all possible types of a file system entity. An entity on the file system can either be a file, a directory or folder, or a link to another file or directory. The precise terminology used to refer to these different types of entity depends on the operating system you are working in.

Operations `do-directory`

home-directory *Function*

Summary Returns the current value of the home directory.

Signature `home-directory () =>` ***home-directory***

Arguments None.

Values *home-directory* An instance of `<pathname>`.

Description Returns the current value of the home directory. The return value of this function can be used with concatenate to create pathnames of entities in the home directory.

`<pathname>` *Type*

Summary The type representing a file system entity.

Equivalent `<string>`

Supertypes None.

Init-keywords None.

Description A type that identifies a file system entity.

Operations **`copy-file`** **`create-directory`** **`delete-directory`** **`delete-file`** **`do-directory`** **`ensure-directories-exist`** **`file-exists?`** **`file-properties`** **`file-property`** **`file-property-setter`** **`file-type`** **`home-directory`** **`rename-file`**

rename-file *Function*

Summary Renames a specified file.

Signature **`rename-file`** ***old-file new-file*** **`#key`** ***if-exists*** **`=> ()`**

Arguments

old-file	An instance of **`<pathname>`**.
new-file	An instance of **`<pathname>`**.
if-exists	An instance of **`<copy/rename-disposition>`**. Default value: **`#"signal"`**.

Values None

Description Renames *old-file* to *new-file*. If *new-file* already exists, the action of this function is controlled by the value of *if-exists*. The default is to prompt you before overwriting an existing file.

This operation may fail if the source and destination are not on the same file system.

See also **`copy-file`**, page 161

`<copy/rename-disposition>`, page 161

root-directories *Function*

Summary Returns a sequence containing the pathnames of the root directories of the file systems on the local machine.

Signature **`root-directories () =>`** ***roots***

Arguments None.

Values *roots* An instances of **`<sequence>`**.

Description Returns a sequence containing the pathnames of the root directories of the file systems on the local machine.

temp-directory *Function*

Summary Returns the pathname of the temporary directory in use.

Signature **`temp-directory () =>`** ***temp-directory***

Arguments None.

Values *temp-directory* An instance of **`<pathname>`**, or false.

Description Returns the pathname of the temporary directory in use. The return value of this function can be used with **`concatenate`** to create pathnames of entities in the temporary directory. If no temporary directory is defined, **`temp-directory`** returns **`#f`**. On Windows the temporary directory is specified by the **`TMP`** environment variable.

working-directory *Function*

Summary Returns the working directory for the current process.

Signature **working-directory () =>** ***working-directory***

Arguments None.

Values *working-directory*

An instance of **<pathname>**.

Description Returns the **<pathname>** of the current working directory in the current process on the local machine. You can use the return value of **working-directory** in conjunction with **concatenate** to specify pathnames of entities in the working directory.

See also **working-directory-setter**, page 174

working-directory-setter *Function*

Summary Sets the working directory for the current process.

Signature **working-directory-setter** ***directory*** **=>** ***directory***

Arguments *directory* An instance of **<pathname>**.

Values *directory* An instance of **<pathname>**.

Description Sets the working directory for the current process.

Note that the following calls are equivalent

```
working-directory() := "C:\USERS\JOHN\FOO.TEXT";
working-directory() := "C:\USERS\JOHN\";
```

as they will both set the working directory to **C:\USERS\JOHN**. The call

```
working-directory() := "C:\USERS\JOHN";
```

is not equivalent as it sets the working directory to **C:\USERS**.

Example

```
working-directory() := "C:\USERS\JOHN\";
```

See also **`working-directory`**, page 173

9

The Operating-System Module

9.1 Introduction

The Operating-System module is part of the System library. It provides an interface to some features of the host machine's operating system.

This chapter describes the functions and constants that the Operating-System module contains.

9.2 Manipulating environment information

The Operating-System module contains a number of interfaces for examining and specifying information about the operating system environment of the host machine. As well as providing constants that you can use in your code, you can examine and set the value of any environment variable in the system.

The following constants contain machine-specific information.

$architecture-little-endian *Constant*

$machine-name *Constant*

$os-name *Constant*

$os-variant *Constant*

$os-version *Constant*

$platform-name *Constant*

These constants contain information about the hardware and software resident on the host machine. The constants let you programmatically check the current system conditions before executing a piece of code.

The constant **`$architecture-little-endian`** is a boolean value that is true if the processor architecture is little-endian and false if it is big-endian. (A processor is little-endian if the rightmost bit in a word is the least-significant bit.) For processors implementing the Intel x86 architecture this value is **`#t`**.

The constant **`$machine-name`** specifies the hardware installed in the machine. The constant **`$os-name`** specifies the operating system that is running.

The constant **`$os-variant`** is a symbol value distinguishing between variants of the operating system identified by **`$os-name`**, where relevant; otherwise it has the same value as **`$os-name`**. On Windows, the possible values are **`#"win3.1"`**, **`#"win95"`**, **`#"win98"`**, and **`#"winnt"`**.

The constant **`$os-version`** is a string value that identifies the version of the operating system. For Windows NT, a typical value would be **`"4.0.1381 Service Pack 3"`**. For Windows 95, a typical value would be **`"4.0.1212 B"`**.

The **$platform-name** constant is an amalgam of the information contained in **$machine-name** and **$os-name**, to enable you to conveniently conditionalize your code based on both values.

The following two functions let you manipulate the values of any environment variables in the system.

environment-variable *Function*

environment-variable-setter *Function*

environment-variable ***name*** **=>** ***value***

environment-variable-setter ***new-value name*** **=>** ***new-value***

The function **environment-variable** returns the current value of any environment variable. The function **environment-variable-setter** lets you specify the value of any environment variable. All arguments and returns values in these functions are instances of **<byte-string>**.If the environment variable passed to **environment-variable** does not exist, it creates it. For **environment-variable-setter**, if *new-value* is **#f**, then the environment variable specified is undefined, if undefining environment variables is supported.

The following functions access information about the user logged on to the current machine, where available.

login-name *Function*

login-name () => ***name-or-false***

Returns as an instance of **<string>** the name of the user logged on to the current machine, or **#f** if unavailable.

login-group *Function*

login-group () => ***group-or-false***

Returns as an instance of **<string>** the group (for example NT domain, or Windows Workgroup) of which the user logged on to the current machine is a member, or **#f** if the group unavailable.

owner-name *Function*

`owner-name () =>` ***name-or-false***

Returns as an instance of **`<string>`** the name of the user who owns the current machine (that is, the name entered when the machine was registered), or **`#f`** if the name unavailable.

owner-organization *Function*

`owner-organization () =>` ***organization-or-false***

Returns as an instance of **`<string>`** the organization to which the user who owns the current machine belongs, or **`#f`** if the name unavailable.

9.3 Manipulating application information

The Operating-System module contains a number of functions for manipulating information specific to a given application, rather than the environment as a whole. You can run or quit any application, and interrogate the running application for application-specific information.

run-application *Function*

`run-application` ***command*** **`#key`** ***under-shell? inherit-console?***
activate? minimize?
`=>` ***status***

Runs the application specified by *command*. Using this function is equivalent to typing *command* in an MS-DOS console window. The function returns the exit status of the application.

If *under-shell?* is **`#t`**, an MS-DOS shell is created to run the application; otherwise, the application is run directly. It is **`#f`** by default.

If *inherit-console?* is **`#t`**, the new application uses the same console window as the current application; otherwise, the new application is created with a separate console window. It is **`#t`** by default.

If the *activate?* keyword is **`#t`**, the shell window becomes the active window. It is **`#t`** by default.

If the *minimize?* keyword is **#t**, the command's shell will appear minimized. It is **#f** by default.

exit-application *Function*

exit-application ***status*** **=> ()**

Terminates the running application. Returns the value of *status*.

application-arguments *Function*

application-name *Function*

application-filename *Function*

application-arguments () => ***arguments***

application-name () => ***name***

application-filename () => ***false-or-filename***

These functions respectively return the arguments passed to the running application, the name of the running application, and the full filename (that is, the absolute pathname) of the running application.

These functions take no arguments. The function **application-arguments** returns an instance of **<simple-object-vector>**; **application-name** returns an instance of **<byte-string>**; and **application-filename** returns an instance of **false-or(<byte-string>)**.

tokenize-command-string *Function*

tokenize-command-string ***line*** **=>** ***command*** **#rest** ***arguments***

This argument passed to this function is an MS-DOS command that could be used to start an application from the MS-DOS command line. It returns the command itself, together with any command-line arguments. All arguments and return values are instances of **<byte-string>**. (In the case of the arguments returned, each individual argument is an instance of **`<byte-string>`**.) You can use this function to break up any MS-DOS command into its constituent parts.

9.4 The OPERATING-SYSTEM module

This section contains a reference entry for each item exported from the Operating-System module's Operating-System module.

application-arguments *Function*

Summary Returns the arguments passed to the running application.

Signature **`application-arguments =>`** ***arguments***

Arguments None.

Values *arguments* An instance of **`<simple-object-vector>`**.

Description Returns the arguments passed to the running application as a vector of instances of **`<byte-string>`**.

See also **`application-filename`**, page 181

`application-name`, page 183

`tokenize-command-string`, page 191

application-filename *Function*

Summary Returns the full filename of the running application.

Signature **`application-filename =>`** ***false-or-filename***

Arguments None.

Values *false-or-filename* An instance of **`false-or(<byte-string>)`**.

Description

Returns the full filename (that is, the absolute pathname) of the running application, or **`#f`** if the filename cannot be determined.

Example

The following is an example of an absolute pathname naming an application:

```
"C:\Program Files\foo\bar.exe"
```

See also

`application-arguments`, page 182

`application-name`, page 181

`tokenize-command-string`, page 191

application-name

Function

Summary

Returns the name of the running application.

Signature

`application-name =>` ***name***

Arguments

None.

Values

name	An instance of **`<byte-string>`**.

Description

Returns the name of the running application. This is normally the command name as typed on the command line and may be a non-absolute pathname.

Example

The following is an example of a non-absolute pathname used to refer to the application name.

```
"foo\bar.exe"
```

See also

`application-arguments`, page 182

`application-filename`, page 181

`tokenize-command-string`, page 191

environment-variable *Function*

Summary

Returns the value of a specified environment variable.

Signature

`environment-variable` ***name*** **`=>`** ***value***

Arguments

name	An instance of **`<byte-string>`**.

Values

value	An instance of **`<byte-string>`**, or **`#f`**.

Description

Returns the value of the environment variable specified by *name*, or **`#f`** if there is no such environment variable.

See also

`environment-variable-setter`, page 184

environment-variable-setter *Function*

Summary

Sets the value of an environment variable.

Signature

`environment-variable-setter` ***new-value name*** **`=>`** ***new-value***

Arguments

new-value	An instance of **`<byte-string>`**, or **`#f`**.
name	An instance of **`<byte-string>`**.

Values

new-value	An instance of **`<byte-string>`**, or **`#f`**.

Description

Changes the value of the environment variable specified by *name* to *new-value*. If *new-value* is **`#f`**, the environment variable is undefined. If the environment variable does not already exist, **`environment-variable-setter`** creates it.

Note: Windows 95 places restrictions on the number of environment variables allowed, based on the total length of the names and values of the existing environment variables. The function **`environment-variable-setter`** only creates a new environment variable if it is possible within these restrictions. See the relevant Windows 95 documentation for more details.

See also **`environment-variable`**, page 184

exit-application *Function*

Summary Terminates execution of the running application.

Signature **`exit-application`** ***status*** **`=> ()`**

Arguments *status* An instance of **`<integer>`**.

Values None.

Description Terminates execution of the running application, returning the value of *status* to whatever launched the application, for example an MS-DOS window or Windows 95/NT shell.

See also **`run-application`**, page 190

login-name *Function*

Summary Returns as an instance of **`<string>`** the name of the user logged on to the current machine, or **`#f`** if unavailable.

Signature **`login-name () =>`** ***name-or-false***

Arguments None.

Values *name-or-false* An instance of **`false-or(<string>)`**.

Description Returns as an instance of **`<string>`** the name of the user logged on to the current machine, or **`#f`** if unavailable.

See also **`login-group`**, page 186

login-group *Function*

Signature **`login-group () =>`** ***group-or-false***

Arguments None.

Values *group-or-false* An instance of **`false-or(<string>)`**.

Description Returns as an instance of **`<string>`** the group (for example NT domain, or Windows Workgroup) of which the user logged on to the current machine is a member, or **`#f`** if the group is unavailable.

See also **`login-name`**, page 185

$machine-name *Constant*

Summary Constant specifying the type of hardware installed in the host machine.

Type Symbol.

Initial value **`#"x86"`**

Description This constant is a symbol that represents the type of hardware installed in the host machine.

Example **`#"x86"`**, **`#"alpha"`**

See also **`$os-name`**, page 187

`$os-variant`, page 187

`$os-version`, page 188

`$platform-name`, page 189

$os-name *Constant*

Summary

Constant specifying the operating system running on the host machine.

Type

Symbol.

Initial value

`#"win32"`

Description

This constant is a symbol that represents the operating system running on the host machine.

Example

`#"win32"`, **`#"osf3"`**

See also

`$machine-name`, page 186

`$os-variant`, page 187

`$os-version`, page 188

`$platform-name`, page 189

$os-variant *Constant*

Summary

Constant specifying which variant of an operating system the current machine is running, where relevant.

Type

Symbol.

Initial value

See Description.

Description

This constant is a symbol value distinguishing between variants of the operating system identified by **`$os-name`**, where relevant; otherwise it has the same value as **`$os-name`**. On

Windows, the possible values are **#"win3.1"**, **#"win95"**, **#"win98"**, and **#"winnt"**.

See also

$machine-name, page 186

$os-name, page 187

$os-version, page 188

$platform-name, page 189

$os-version *Constant*

Summary

Constant specifying which version of an operating system the current machine is running.

Type

<string>

Initial value

See Description.

Description

The constant **$os-version** is a string value that identifies the version of the operating system. For Windows NT, a typical value would be **"4.0.1381 Service Pack 3"**. For Windows 95, a typical value would be **"4.0.1212 B"**.

See also

$machine-name, page 186

$os-name, page 187

$os-variant, page 187

$platform-name, page 189

owner-name *Function*

Summary

Returns the name of the user who owns the current machine, if available.

Signature **`owner-name () =>`** ***name-or-false***

Arguments None.

Values name-or-false An instance of **`false-or(<string>)`**.

Description Returns as an instance of **`<string>`** the name of the user who owns the current machine (that is, the name entered when the machine was registered), or **`#f`** if the name is unavailable.

owner-organization *Function*

Summary Returns the organization to which the user who owns the current machine belongs, if available.

Signature **`owner-organization () =>`** ***organization-or-false***

Arguments None.

Values *organization-or-false*
An instance of **`false-or(<string>)`**.

Description Returns as an instance of **`<string>`** the organization to which the user who owns the current machine belongs, or **`#f`** if the name is unavailable.

$platform-name *Constant*

Summary Constant specifying the operating system running on and the type of hardware installed in the host machine.

Type Symbol.

Initial value **`#"x86-win32"`**

Description

This constant is a symbol that represents the both the operating system running on, and the type of hardware installed in, the host machine. It is a combination of the **`$os-name`** and **`$machine-name`** constants.

Example

`#"x86-win32"`, **`#"alpha-osf3"`**

See also

`$machine-name`, page 186

`$os-name`, page 187

run-application *Function*

Summary

Launches an application using the specified name and arguments.

Signature

`run-application` ***command*** **`#key`** ***minimize? activate?***
under-shell? inherit-console?
`=>` ***status***

Arguments

command	An instance of **`<string>`**.
minimize?	An instance of **`<boolean>`**.
activate?	An instance of **`<boolean>`**.

Values

status	An instance of **`<integer>`**.

Description

Launches an application using the name and arguments specified in command. Using this function is equivalent to typing the command in a MS-DOS window. The return value is the exit status returned by the application.

If the *minimize?* keyword is **`#t`**, the command's shell will appear minimized. It is **`#f`** by default.

If the *activate?* keyword is **`#t`**, the shell window becomes the active window. It is **`#t`** by default.

If the *under-shell?* keyword is **`#t`**, an MS-DOS shell is created to run the application; otherwise, the application is run directly. It is **`#f`** by default.

If the *inherit-console?* keyword is **`#t`**, the new application uses the same console window as the current application; otherwise, the new application is created with a separate console window. It is **`#t`** by default.

See also **`exit-application`**, page 185

tokenize-command-string *Function*

Summary Parses a command line into a command name and arguments.

Signature **`tokenize-command-string`** ***line*** **`=>`** ***command*** **`#rest`** ***arguments***

Arguments

Arguments	*line*	An instance of **`<byte-string>`**.
Values	*command*	An instance of **`<byte-string>`**.
	arguments	Instances of **`<byte-string>`**.

Description Parses the command specified in *line* into a command name and arguments. The rules used to tokenize the string are given in Microsoft’s C / C++ reference in the section “Parsing C Command-Line Arguments”.

See also **`application-arguments`**, page 182

`application-name`, page 183

10

The Network Library

10.1 Overview

This chapter covers the Network library. The Network library provides Internet address protocols and TCP/IP server and client sockets. It exports a single module, called Sockets.

Note: The Network library is not available in the Personal Edition of Harlequin Dylan.

10.2 Utilities

This section covers the **`start-sockets`** function, which all libraries using the Network library must call before *any* other call to the Network library API. It also covers the **`with-socket-thread`** macro which registers the current thread as a thread that will call a socket function that blocks.

start-sockets *Function*

```
start-sockets () => ()
```

Applications must call this function before using *any* other function or variable from the Network library.

This function is necessary because the Win32 API library Winsock2, which the Network library calls to get native socket functionality, requires applications to call an initialization function before calling any Winsock2 API functions. The call to **start-sockets** calls this underlying Winsock2 function.

Note that you must not call **start-sockets** from a top-level form in any DLL project. The combination of this, and the restriction that you must call **start-sockets** before calling anything else, implies that no Network library function or variable can be called (directly or indirectly) from a top-level form in any DLL project. Instead, the DLL project should define a start function that calls **start-sockets** (directly or indirectly) or re-export **start-sockets** so that their clients can arrange to have it called from a top-level form in an appropriate EXE project.

Applications using the Network library must arrange for **start-sockets** to be called (directly or indirectly) before *any* other sockets API functions. A good place to do this is at the beginning of your start function (usually the **main** method). For example:

```
define method main () => ();
  start-sockets();
  let the-server = make(<TCP-server-socket>, port: 7);
  ...
end;

begin
  main();
end;
```

New start functions that call **start-sockets** and that are defined for DLL projects that use the Network library will inherit all of the restrictions described above for **start-sockets**.

Calling a Network library function before calling **start-sockets** results in a **<sockets-not-initialized>** error. Calling **start-sockets** from a top-level form in a DLL project will result in unpredictable failures—probably access violations during initialization.

with-socket-thread *Statement macro*

```
with-socket-thread (#key server?)
  body
end;
```

Registers the current thread as a blocking socket thread, that is, a thread which will call a socket function that blocks, such as **read-element** or **accept**.

The reason for the registration is that Network library shutdown can then synchronize with these threads. The early part of the shutdown sequence should cause the threads to unblock with an **<end-of-stream-error>** so that they can do whatever application cleanup is necessary. Once these threads have exited, the rest of the shutdown sequence can be executed.

A server socket thread (blocking on **accept** rather than **read-element**) notices that the shutdown sequence is underway slightly later, with a **<blocking-call-interrupted>** condition.

10.3 Internet addresses

This section covers Internet address protocols.

10.3.1 Basic Internet address protocol

This section covers the class **<internet-address>** and related generic functions and constants.

<internet-address> *Open abstract primary instantiable class*

Superclasses: **<object>**

The class of objects representing Internet addresses used as endpoints for peer-to-peer socket connections.

To construct an **<internet-address>** object you must supply either the **name:** or **address:** keyword. For example:

```
  make (<internet-address>, name: "www.whatever.com")
```

or

```
  make (<internet-address>,  address: "9.74.122.0")
```

`make` on **`<internet-address>`** returns an instance of **`<ipv4-address>`**.

Keywords:

`name:` An instance of **`<string>`** representing a symbolic internet address.

`address:` An instance of **`<string>`** representing a presentation (dotted) form Internet address or an instance of **`<numeric-address>`** (see below).

host-name *Open generic function*

`host-name` ***internet-address*** **`=>`** ***name***

Returns an instance of **`<string>`** containing a symbolic host name. The *internet-address* argument must be an instance of **`<internet-address>`**.

Usually the name returned is the canonical host name. Note, however, that the implementation is conservative about making DNS calls. Suppose that the **`<internet-address>`** instance was created with the **`name:`** keyword and no other information. If the application has not made any other requests that would require a DNS call, such as to **`host-address`** or **`aliases`** (see below), the name that this function returns will be the one specified with the **`name:`** keyword, regardless of whether that is the canonical name or not.

host-address *Open generic function*

`host-address` ***internet-address*** **`=>`** ***address***

Returns an instance of **`<string>`** containing the presentation form of the host address. In the case of multi-homed hosts this will usually be the same as:

`multi-homed-internet-address.all-addresses.first.host-address`

In the case of an Internet address created using the **`address:`** keyword it will be either the keyword value or **`all-addresses.first.host-address`**.

numeric-host-address *Open generic function*

`numeric-host-address` ***internet-address*** **`=>`** ***numeric-address***

Returns the host address as a **`<numeric-address>`**.

all-addresses *Open generic function*

`all-addresses` ***internet-address*** **`=>`** ***sequence***

Returns an instance of **`<sequence>`** whose elements are **`<internet-address>`** objects containing all known addresses for the host.

aliases *Open generic function*

`aliases` ***internet-address*** **`=>`** ***sequence***

Returns an instance of **`<sequence>`** whose elements are instances of **`<string>`** representing alternative names for the host.

$loopback-address *Constant*

An instance of **`<internet-address>`** representing the loopback address: **`"127.0.0.1"`**.

$local-host *Constant*

An instance of **`<internet-address>`** representing the host on which the application using sockets is correctly running.

Note that this value is not necessarily the same as would be created by the expression

```
make (<internet-address>, name: "localhost")
```

The address assigned to the symbolic name **`localhost`** is dependent on the configuration of DNS. In some cases this may be configured to be the loopback address rather than a real address for the local host.

10.3.2 The <IPV6-ADDRESS> class

This name is reserved for future development.

10.3.3 The <NUMERIC-ADDRESS> class

This section describes numeric Internet representation and associated protocols.

<numeric-address> *Sealed abstract primary class*

Superclasses: **`<object>`**

The class of objects representing the numeric form of an Internet addresses.

Currently only ipv4 (32-bit) addresses are supported. Ipv6 addresses will be added when they are supported by Winsock2. In general **`<numeric-address>`** objects are accessed using the functions **`host-order`** or **`network-order`**, depending on the context in which they are employed.

network-order *Sealed generic function*

`network-order` ***address*** **`=>`** ***network-order-address***

Returns the value of the numeric address in network order. The argument is a general instance of **`<numeric-address>`**. The class of the object returned depends upon the particular subclass of the argument; the **`network-order`** method for **`<ipv4-numeric-address>`** returns an instance of **`<machine-word>`**.

Network order is big-endian byte order.

host-order *Sealed generic function*

`host-order` ***address*** **`=>`** ***host-order-address***

Like **`network-order`** but returns the value in host order.

Host order is either big-endian byte order on a big-endian host machine and little-endian on a little-endian host machine.

10.3.3.1 IPV4 addresses

<ipv4-numeric-address> *Open abstract primary instantiable class*

Superclasses: **`<numeric-address>`**

The single slot of this class contains a 32-bit value representing a ipv4 address. This slot is accessed by the generic functions **`network-order`** and **`host-order`** described above. **`<ipv4-numeric-address>`** has two concrete subclasses **`<ipv4-network-order-address>`** and **`<ipv4-host-order-address>`**. Make **`<ipv4-numeric-address>`** returns one or the other of these depending upon the value of the **`order:`** keyword.

Keywords:

`value:`	An instance of **`<machine-word>`**. Required.
`order:`	One of **`#"network-order"`** or **`#"host-order"`**. Required.

host-order *G.f. method*

`host-order` ***ip4-numeric-address*** **`=>`** ***machine-word***

Returns the numeric address in host order as an instance of **`<machine-word>`**. The argument is an instance of **`<ip4-numeric-address>`**.

network-order *G.f. method*

`network-order` ***ipv4-numeric-address*** **`=>`** ***machine-word***

Returns the numeric address in network order as an instance of **`<machine-word>`**. The argument is an instance of **`<ip4-numeric-address>`**.

as *G.f. method*

`as` ***string*** ***ipv4-numeric-address*** **`=>`** ***string***

Returns the presentation (dotted string) form of an instance of **`<ip4-numeric-address>`**.

<ipv4-network-order-address> *Sealed concrete class*

Superclasses: **`<ipv4-numeric-address>`**

Concrete subclass for network-order numeric addresses.

```
make(<ipv4-network-order-address>)
```

is equivalent to

```
make(<ipv4-numeric-address>, order: network-order)
```

<ipv4-host-order-address> *Sealed concrete class*

Superclasses: **`<ipv4-numeric-address>`**

Concrete subclass for host order numeric addresses.

10.4 Sockets

This section describes socket classes and protocols.

10.4.1 The <ABSTRACT-SOCKET> class

<abstract-socket> *Open abstract uninstantiable free class*

Superclasses: **`<object>`**

The common superclass of all socket objects including **`<socket>`** (IP client socket), **`<server-socket>`** and **`<socket-accessor>`**.

Keywords:

`socket-descriptor:`

A Windows handle or UNIX fd (file descriptor) for the socket. In general users of the sockets API should not need to use this keyword. Only implementors of new socket classes should be interested.

Each subclass of **`<abstract-socket>`** must provide methods for **`close`** and for the following generic functions:

local-port *Open generic function*

`local-port` ***socket*** **`=>`** ***port-number***

Returns the local port number for an instance of **`<socket>`**, **`<datagram-socket>`** or **`<server-socket>`**. The return value is an instance of **`<integer>`**.

socket-descriptor *Open generic function*

socket-descriptor ***socket*** **=> descriptor**

Returns the descriptor (handle or fd) for the socket. The argument is an instance of **<abstract-socket>** and the return value an instance of **<accessor-socket-descriptor>**.

local-host *Open generic function*

local-host ***socket*** **=>** ***host-address***

Returns the address of the local host. The argument is an instance of **<abstract-socket>** and the return value an instance of **<internet-address>**.

10.4.2 The <SERVER-SOCKET> class

<server-socket> *Open abstract primary instantiable class*

Superclasses: **<abstract-socket>**

Server-sockets listen on a specified port for connection requests which come in over the network. Either the **port:** or **service:** keyword must be supplied.

Keywords:

service: An instance of **<string>** containing an abstract name for a service with a “well-known” port, such as **"ftp"** or **"daytime"**. Valid names depend on the configuration of the DNS. Required unless **port:** is supplied.

port: An instance of **<integer>** identifying the port on which the **<server-socket>** should listen for connection requests. Required unless **service:** is supplied.

protocol: An instance of **<string>** naming the protocol. Currently **"tcp"** is the only supported protocol. You can create instances of protocol-specific subclasses as an alternative to using the **protocol:** keyword. For example, **make(<server-socket>, protocol: "tcp", ...)** is equivalent to **make(<TCP-server-socket>, ...)**.

make on **(<server-socket>)** returns an instance of **<tcp-server-socket>** by default.

accept *Open generic function*

accept ***server-socket*** **#rest** ***args*** **#key =>** ***result***

Blocks until a connect request is received, then it returns a connected instance of **<socket>**. The particular subclass of **<socket>** returned depends on the actual class of the argument, which must be a general instance of **<server-socket>**. Calling accept on **<tcp-server-socket>** returns a connected **<tcp-socket>**. The keyword arguments are passed to the creation of the **<socket>** instance. For UDP sockets **accept** returns immediately with an instance of **<udp-socket>**. No blocking happens for UDP sockets because they are connectionless. After reading from a UDP socket returned from **accept** the socket can be interrogated for the location of the sender using **remote-host** and **remote-port**.

with-server-socket *Macro*

```
with-server-socket (server-var [:: server-class], keywords)
  body
end;
```

Creates an instance of **<server-socket>**, using the (optional) *server-class* argument and keyword arguments to make the **<server-socket>**, and binds it to the local variable named by *server-var*. The *body* is evaluated in the context of the binding and the **<server-socket>** is closed after the body is executed.

start-server *Macro*

```
start-server ([server-var = ]socket-server-instance,
                socket-var [, keywords])
  body
end;
```

Enters an infinite **while(#t) accept** loop on the server socket. Each time accept succeeds the **<socket>** returned from accept is bound to *socket-var* and the *body* is evaluated in the context of the binding. When *body* exits, **accept** is called again producing a new binding for *socket-var*. The optional keywords are passed to the call to **accept**.

10.4.3 The <TCP-SERVER-SOCKET> class

<tcp-server-socket> *Class*

Superclass: **<server-socket>**

The class of TCP server sockets. A server socket is an object which listens for requests for connections from the network. When accept is called on the server socket and a request for connection is detected, accept returns a connected **<socket>**.

Keywords:

element-type: Establishes a new default for the **element-type** of **<TCP-socket>** instances returned by calling **accept** with this server socket as the argument to **accept**. This default **element-type** may be overridden for any particular call to **accept** by using the **element-type:** keyword to **accept**. If no **element-type:** is specified when the server socket is created, **<byte-character>** is used as the default **element-type**.

accept *G.f. method*

accept ***server-socket*** **#rest** ***args*** **#key** ***element-type*** **=>** ***connected-socket***

This method on **accept** takes an instance of type **<tcp-server-socket>** and returns a connected instance of **<tcp-socket>**. The **element-type:** keyword controls the element type of the **<tcp-socket>** (stream) returned from **accept**. If the keyword is not supplied, the default value used is **#f**. The other keyword arguments are passed directly to the creation of the **<tcp-socket>** instance.

10.4.4 The <SOCKET> class

<socket> *Open abstract free instantiable class*

Superclasses: **<abstract-socket>**, **<external-stream>**

The class of general client sockets. All client sockets are streams.

Keywords:

direction: Specifies the direction of the stream. It must be one of **#"input"**, **#"output"**, and **"#input-output"**. This keyword is an inherited streams class keyword. See the Streams library documentation in the *System and I/O* library reference for a full description.

element-type: An instance of **<class>**. Useful values are **<byte-character>** and **<byte>**. This keyword is an inherited streams class keyword. See the Streams library documentation in the *System and I/O* library reference for a full description.

10.4.5 The <BUFFERED-SOCKET> class

<buffered-socket> *Class*

Superclasses: **<socket>**, **<double-buffered-stream>**

Socket streams whose elements are bytes or characters. These inherit buffering protocols and the implications of **read**, **write**, **read-element**, **write-element**, **force-output** and suchlike methods from **<double-buffered-stream>**.

Keywords:

force-output-before-read?:

An instance of **<boolean>**. Defaults value: **#t**. The methods which implement the stream reading protocols (**read**, **read-line**, **read-element** and so on) for instances of **<socket>** call **force-output** by default before blocking. This is to ensure that any pending output has been sent to the peer before the socket blocks waiting to read data sent by the peer. This corresponds to the expected, usual behavior of single-threaded client sockets and avoids deadlock in usual cases. Multi-threaded applications, particularly applications where one thread is reading and another thread is writing to the same socket, may wish to inhibit the default **force-output**. If the socket is created with **force-output-before-read?:** as **#f**, **force-output** will not be called before the read functions block.

10.4.6 The <TCP-SOCKET> class

The class of TCP client sockets.

<tcp-socket> *Class*

Superclasses: **<buffered-socket>**

The class of TCP client sockets.

Keywords:

Of the keywords below, **host:** and one of either **service:** or **port:** are required.

`host:` An instance of `<internet-address>` or `<string>`. The remote host to connect to. The `<string>` may be either a host name or a presentation-form Internet address. Required.

`service:` An instance of `<string>`. A `<string>` containing an abstract name for a service with a “well-known“ port, such as `"ftp"` or `"daytime"`. Valid names depend on the configuration of the DNS. Required unless `port:` is supplied.

`protocol:` An instance of `<string>` naming the protocol. Currently `#"tcp"` and `#"udp"` are the only supported protocols. You can create instances of protocol-specific subclasses as an alternative to using the `protocol:` keyword. For example `make(<socket>, protocol: #"tcp", …)` is equivalent to `make(<TCP-socket>, …)`. `make` on `<socket>` returns an instance of `<tcp-socket>` by default.

`port:` An instance of `<integer>` representing the remote port to connect to. Required unless `service:` is supplied.

`element-type:` An instance of `<class>`. Useful values for `<tcp-streams>` are `<byte-character>` and `<byte>`. This keyword is an inherited streams class keyword. See Chapter 5, “The Streams Module” for a full description.

remote-port *Open generic function*

`remote-port` ***socket*** `=>` ***port-number***

Returns the remote port number for a `<socket>`. The value returned is an instance of `<integer>`.

remote-host *Open generic function*

`remote-host` ***socket*** `=>` ***remote-host-address***

Returns the remote host for a **<socket>**. The value returned is an instance of **<internet-address>**.

10.4.7 The <UDP-SOCKET> class

The class of UDP client sockets.

<udp-socket> *Class*

Superclasses: **<buffered-socket>**

The class of UDP client sockets.

Keywords:

Of the keywords below, **host:** and one of either **service:** or **port:** are required.

host: An instance of **<internet-address>** or **<string>**. The remote host to connect to. The **<string>** may be either a host name or a **presentation-form** Internet address. Required.

service: An instance of **<string>**. A **<string>** containing an abstract name for a service with a "well-known port", such as **"ftp"** or **"daytime"**. Valid names depend on the configuration of the DNS. Required unless **port:** is supplied.

protocol: An instance of **<string>** naming the protocol. Currently **#"tcp"** and **#"udp"** are the only supported protocols. You can create instances of protocol-specific subclasses as an alternative to using the **protocol:** keyword. For example **make(<socket>, protocol: #"udp", …)** is equivalent to **make(<UDP-socket>, …)**. **make** on **<socket>** returns an instance of **<tcp-socket>** by default.

port: An instance of **<integer>** representing the remote port to connect to. Required unless **service:** is supplied.

element-type: An instance of **<class>**. Useful values for **<udp-socket>**s are **<byte-character>** and **<byte>**. This keyword is an inherited streams class keyword. See Chapter 5, "The Streams Module" for a full description.

10.4.8 The <UDP-SERVER-SOCKET> class

The class of UDP server sockets.

<udp-server-socket> *Class*

Superclass: **<server-socket>**

The class of UDP server sockets. A server socket is an object that listens for requests from the network. When **accept** is called on the UDP server socket, **accept** returns a **<udp-socket>**.

Keywords:

element-type: Establishes a new default for the element-type of **<UDP-socket>** instances returned by calling **accept** with this server socket as the argument to **accept**. This default element-type may be overridden for any particular call to **accept** by using the **element-type:** keyword to **accept**. If no **element-type:** is specified when the server socket is created, **<byte-character>** is used as the default element-type.

10.5 Socket conditions

This section lists the socket condition classes in the Network library.

10.5.1 <socket-condition>

All socket conditions are general instances of **<socket-condition>**. Some are recoverable and others are not.

<socket-condition> *Condition*

Superclasses: **`<simple-condition>`**

The class of socket conditions. It inherits the **`format-string:`** and **`format-arguments:`** keywords from **`<simple-condition>`**.

Slots:

`socket-condition-details`

Most socket conditions originate in error return codes from Harlequin Dylan's Winsock2 library, an FFI interface to the native socket library Winsock2.

The **`socket-condition-details`** slot provides information about the low-level failure which was the source for the condition. In most cases this slot will hold an instance of **`<socket-accessor-condition>`**, below.

When creating general instances of **`<socket-condition>`**, you can use the **`details:`** keyword to set the value for this slot.

10.5.2 <socket-error>

The class **`<socket-error>`** is the superclass of all unrecoverable socket conditions.

<socket-error> *Condition*

Superclasses: **`<socket-condition>`**

The class of socket conditions from which no recovery is possible.

10.5.2.1 <internal-socket-error>

The class **`<internal-socket-error>`** is the class of unexpected socket errors.

<internal-socket-error> *Condition*

Superclasses: **`<socket-error>`**

The class of unexpected errors from Harlequin Dylan's Winsock2 library, an FFI interface to the native socket library Winsock2.

Inspect the contents of the **`socket-condition-details`** slot for more information.

10.5.3 <recoverable-socket-condition>

The **`<recoverable-socket-condition>`** class is the general class of socket conditions for which an application may be able to take some remedial action.

<recoverable-socket-condition> *Condition*

Superclasses: **`<socket-condition>`**

The general class of socket conditions for which an application may be able to take some remedial action.

For instance, a web browser receiving such conditions as **`<connection-refused>`** or **`<host-not-found>`** (see below) would normally expect to report those conditions to the user and continue with some other connection request from the user, while a server receiving a **`<connection-closed>`** condition from a connected **`<socket>`** would probably close the **`<socket>`** and continue to handle other requests for connections.

10.5.3.1 <network-not-responding>

<network-not-responding> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

The network — probably a local network — is down. Try again later.

10.5.3.2 <invalid-address>

<invalid-address> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

A badly formed address string has been passed to a function trying to make an **`<internet-address>`**.

10.5.3.3 <host-not-found>

<host-not-found> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

The Domain Name Server (DNS) cannot resolve the named host or internet address. Try again with a different (correct) name or address.

10.5.3.4 <server-not-responding>

<server-not-responding> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

The Domain Name Server (DNS) did not respond or returned an ambiguous result. Try again.

10.5.3.5 <host-unreachable>

<host-unreachable> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

The remote host cannot be reached from this host at this time.

10.5.3.6 <socket-closed>

<socket-closed> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

The socket or server socket has been closed.

Most operations on closed instances of **`<TCP-socket>`** return instances of **`<stream-closed-error>`** (from the Streams library) rather than instances of **`<socket-closed>`**.

10.5.3.7 <connection-failed>

<connection-failed> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

The attempt to connect to the remote host was not successful. Connection failed for one of the following reasons: because the connect request timed out or because it was refused, or because the remote host could not be reached.

10.5.3.8 <connection-closed>

<connection-closed> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

The connection to the remote host has been broken. The socket should be closed. To try again, open a new socket.

10.5.3.9 <address-in-use>

<address-in-use> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

A process on the machine is already bound to the same fully qualified address. This condition probably occurred because you were trying to use a port with an active server already installed, or a process crashed without closing a socket.

10.5.3.10 <blocking-call-interrupted>

<blocking-call-interrupted> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

A blocking socket call, like **`read`**, **`write`** or **`accept`**, was interrupted.

10.5.3.11 <out-of-resources>

<out-of-resources> *Condition*

Superclasses: **`<recoverable-socket-condition>`**

The implementation-dependent limit on the number of open sockets has been reached. You must close some sockets before you can open any more. The limits for Windows NT (non-server machines) and Windows 95 are particularly small.

10.5.4 <socket-accessor-error>

<socket-accessor-error> *Condition*

Superclasses: **`<socket-error>`**

An implementation-specific error from the C-FFI interface to the native socket library. Usually instances of this class these appear in the **`socket-condition-details`** slot of another **`<socket-condition>`**.

10.5.4.1 <win32-socket-error>

<win32-socket-error> *Condition*

Superclasses: **`<socket-accessor-error>`**

A Win32-specific error from the Winsock2 library, a C-FFI interface to the native socket library Winsock2. A function in the FFI library has returned an error return code.

Slots:

`WSA-numeric-error-code`

Contains the numeric error code that was returned. An instance of **`<integer>`**.

`WSA-symbolic-error-code`

Contains an instance of **`<string>`** giving the symbolic (human-readable) form of the error code. For example, the string might be **`"wsanotsock"`**.

`explanation` An explanation if any of the error. An instance of **`<string>`**.

`calling-function`

The name of Winsock2 FFI interface function which returned the error code. An instance of **`<string>`**.

Index