Gwydion Streams for Dylan

Designed by the Gwydion Project

Version 3.1 : 17 Nov 94

This document describes the `Streams` library designed by the Gwydion Project at Carnegie Mellon University. The primary goals of this stream interface are efficiency, simplicity, and the ability to extend the stream protocol to more complex stream objects and more complex data objects. The stream design focuses on byte-oriented, buffered input and output. Implementations and users have access to the internal buffers of streams for efficiency reasons. Implementations and users can extend the stream protocol to a new stream subclass just by extending the interface to the stream 's buffer.

The `Streams` library contains none of the following kinds of functionality:

- complex stream subclasses implementing streams such as concatenated, broadcast, echo, two-way conglomerate, and so on.
- formatted output, such as Common Lisp's **format** function or C's **printf**.
- formatted input, such as C's **scanf**.
- so called **read**able printing, such as Common Lisp's **print** function.
- parsing, such as Common Lisp's **read** function.
- output and input of arbitrary binary representations, such as dumping integers and floats in raw, implementation-dependent form.
- streams that do not support the stream protocol entirely; for example, a record-oriented stream may not be able to support the **read-byte** function.

Most of these items fit naturally on top of the `Streams` library, but regardless of that, separate libraries should provide the kinds of functionality listed above.

1. Overview

This section introduces some underlying concepts of the `Streams` library and provides a sketch of the required stream classes and functions. The `Streams` library exports at least one module, the `Streams` module, and it is this module that exports all of the definitions described in this document.

1.1. Classes

The following classes are required (indentation shows subclass relationship):

```
<stream> [Abstract]
   <random-access-stream> [Abstract]
      <string-input-stream> [Abstract]
         <byte-string-input-stream> [Concrete]
      <string-output-stream> [Abstract]
         <byte-string-output-stream> [Concrete]
   <file-stream> [Abstract Instantiable]
```

Implementations are required to provide a concrete class that is a subclass of **<random-access-stream>** and **<file-stream>**, and when users call the **make** function on **<file-stream>**, they get an instance o f this implementation-dependent concrete class.

1.2. Functions

This subsection sketches the functions exported from the `Streams` module. This is not a complete list of applicable functions; for example, **make** is not listed for any class, and sequence operations are not listed for the **<buffer>** class.

The following functions comprise the *Basic I/O Protocol* (described in Section *The* <*stream*> *Protocol*):

read-byte
peek-byte
read-line
input-available?
flush-input
force-output
synchronize-output

The following functions comprise the *Data Extension Protocol* (described in Section *The* <*stream*> *Protocol*):

read-as
read-into!
write
write-line

The following functions comprise the *Random Access Protocol* (described in Section *The* <*random-access-stream*> *Protocol*):

stream-position
stream-position-setter
adjust-stream-position
stream-size

The following function comprises the **<string-output-stream>** protocol (described in Section *The* <*string-output-stream*> *Protocol*):

string-output-stream-string

The following functions comprise the *Buffer Access Protocol* (described in Section *The* <*stream*> *Protocol*):

get-input-buffer
release-input-buffer
fill-input-buffer
input-available-at-source?
get-output-buffer
release-output-buffer
empty-output-buffer
force-secondary-buffers
synchronize

The following functions comprise the *Stream Extension Protocol* (described in Section *The* <*stream*> *Protocol*):

close
stream-extension-get-input-buffer
stream-extension-release-input-buffer
stream-extension-fill-input-buffer
stream-extension-input-available-at-source?
stream-extension-get-output-buffer
stream-extension-release-output-buffer
stream-extension-empty-output-buffer
stream-extension-force-secondary-buffers
stream-extension-synchronize

The following functions comprise the **<buffer>** protocol (described in Section *The* <*buffer*> *Protocol*):

buffer-subsequence
copy-from-buffer!
copy-into-buffer!

The following functions comprise the *Locking Protocol* (described in Section *The Locking Protocol*):

lock-stream
unlock-stream
stream-locked?

1.3. Buffers

The `Streams` module provides the framework for defining various high-level streams, such as Common Lisp has. As stated previously, one of the primary goals of this stream interface is efficiency. The design of Common Lisp streams is inherently inefficient. For example, in Common Lisp if you have a general output routine for writing multiple items (or multiple components of an object) to a stream, you pay a big penalty in having to perform generic function invocation in the middle of your loop (or over a sequence of stream operations). This is because you cannot know until run time what sort of stream you have.

This stream interface solves this problem by providing a clean protocol for directly manipulating the internal buffers of streams. Since all streams have internal buffers that are instances of the same class, users can write efficient, stream-independent output routines by operating on the streams' buffers. Buffer methods can be fully determined at compile time. A Buffer is an instance of the **<buffer>** class, which is a sealed subclass of **<vector>**. Buffers have element type **<byte>** (see Subsection *Miscellaneous Definitions* for type definitions).

When users manipulate buffers directly, they are entirely responsible for maintaining the buffer's state. Once a user gets a stream's buffer, the stream object has no means to track activity in the buffer. To use a stream's buffer, users must explicitly get the buffer. At this time, the stream indicates where the pending input or output is in the buffer. If the stream is an input stream, users can consume as much of the pending input as necessary, filling the buffer with input from the stream's source any number of times, including none. If the stream is an output stream, users can store as much pending output in the buffer as necessary, forcing out the output any number of times, including none. When users are done with a stream's buffer, they must explicitly release the buffer and indicate to the stream the buffer's current state.

Stream implementations are required to be able to determine when an application is explicitly using a stream's buffer. Between the time when an application explicitly gets a stream's buffer and then later releases the buffer, the application is said to *hold* the buffer. Many routines in the stream interface are defined to only work correctly when the application holds the stream's buffer. These routines are defined, with respect to guaranteeing correct behavior and blocking behavior, to allow implementations freedom to provide safe and good development environment support and to provide good support for stream implementations in multi-threaded execution environments. See Subsections *Stream Extension Protocol* and *Buffer Access Protocol*.

When applications do not hold a stream's buffer, they cannot make any assumptions about the maintenance of the buffer. In particular, it is an error to save references to buffers, perform operations on buffers when not holding them (such as **element-setter**), and so on.

1.4. Usage Models

The `Streams` library supports three styles of usage. The first, and probably most common, is a simple style that uses high-level input and output functions, such as **read-byte**, **read-as**, **write**, and so on. The simple style will be the least efficient, but it provides convenience and is generally useful. In the second style, applications use the internal buffers of streams to grab multiple bytes of input at once or to provide multiple bytes of output at once. Using the buffers is relatively efficient, less convenient for the programmer, but generally just as useful as the simple style. In the third style, applications directly allocate buffers and use them for reading and writing very large chunks of data, probably whole files at once. Directly using buffers separately from the internal buffers of streams provides the greatest efficiency and is slightly less complicated than the second style of usage; however, the third style has very specific uses and does not mix well with the first style.

The second style of usage is quite flexible. Users can mix the first and second styles of usage. Though users must explicitly get and release buffers to use buffers directly, when the buffers are not held, simple style input and output integrates smoothly with input read from and output placed in buffers directly. The second style of usage also provides more precise control over blocking behavior for some operations that are logically equivalent in the two styles.

Implementations should support the third style of usage without any unnecessary inefficiencies, such as double buffering.

The Streams library provides functions such as **read-into!** and **write** that should have method that allow users to very efficiently fill a large buffer from a stream's source or deliver a large buffer to a stream's destination.

1.5. Characters

Because international character standards and file formats are still converging on a clear winner, this stream interface specifies very little about characters. Implementors are required to provide a subclass of **<character>** called **<byte-character>** and methods specialized on this class for the **read-as** and **write** generic functions. The **<byte-character>** class represents the ASCII character set (or extensions to ASCII). The Streams library requires support for ASCII characters because they capture a large portion of character I/O, and their use should be standard. Implementations are free to support other characters, and their support can naturally fit into the Streams library.

1.6. Locking

The Streams library provides two mutual exclusion mechanisms, the *Buffer Access Protocol* and the *Locking Protocol*. The *Locking Protocol* isolates access to a stream so that only one thread may use the stream at one time, and in a singlethreaded Dylan implementation, these functions do nothing. The *Buffer Access Protocol* isolates access to a buffer within a single thread, and applications use these functions regardless of whether the Dylan implementation is multithreaded. The *Buffer Access Protocol* functions that get a buffer first lock the stream, and those functions that release the buffer unlock the stream. Thus, getting a buffer both isolates access to the stream for a single thread and ensures that the single thread does not try to get the stream's buffer multiple times while already holding the buffer.

Stream locks have multilocking semantics. A single thread may repeatedly lock a particular stream, but that thread must unlock the stream once for each time it locked the stream. This allows a high-level printing routine to lock a stream across several calls to output functions, ensuring all the output is contiguous at the stream's destination. For example, the **write-line** function locks its stream argument and then calls the **write** function twice. The **write** function locks its stream argument by calling **get-output-buffer**, but because of the multilocking semantics, the call to **write** within **write-line** does not block waiting for a lock. Before returning, **write-line** unlocks the stream so that other routines may call output functions on the stream or get the stream's buffer for direct manipulation.

The *Buffer Access Protocol* isolates access to a buffer within a single thread to prevent reentrancy problems and programming mistakes. Essentially, the lightweight buffer locking ensures that applications do not call output functions that directly manipulate a stream's buffer from within routines that are already directly manipulating the stream's buffer. This situation must be forbidden because the inner call to get the buffer cannot reliably return the state of the stream's buffer while the application already holds the buffer.

1.7. Miscellaneous Features

The Streams library provides streams that support input operations, streams that support output operations, and streams that support both. The kind of operations supported by a particular stream is determined by keywords supplied when making the stream (such as with **<file-stream>** instances) or by the class of the stream (such as with **<string-input-stream>**). Implementations of output streams should arrange to have the Dylan run-time system force output when an application exits, but portable programs do not rely on this feature.

When *opening* is applicable, users open a stream by calling **make** on a stream class. The initialization protocols for the different subclasses of **<stream>** take keyword arguments that are appropriate to each subclass. The **make** function returns an open stream object. Unless otherwise specified, it is an error to use stream operations on closed streams.

When describing the stream protocol relating to buffers, this document uses the term *end* to discuss the end of valid data in buffers. The use of *end* is always an exclusive end of the data, that is, the buffer's element indicated by an end value is never part of the valid data. An end value may not be a valid buffer index; because Dylan sequences have zero-based indexes, an end value may be equal to the size of the buffer.

This document declares some arguments and return values to have the type **<boolean>**. This is not part of Dylan, and the Streams library does not export it. As used in this document, it has the following definition:

```
union(singleton(#t), singleton(#f))
```

This document declares some arguments and return values to have the type **<fixed-integer>** to designate an implementation-dependent, finite integer type. The intent is that these integers are known to be small, lightweight integers (and will never be infinite precision integers).

2. Classes and Other Types

The `Streams` module exports the following classes:

<stream> [Abstract Class]

This class is a subclass of **<object>**. All streams inherit from this class. **Make** of **<stream>** subclasses accepts a size: keyword argument that indicates the user's choice for a buffer size. Users must check the size of the stream's buffer because implementations may ignore the size: argument.

<random-access-stream> [Abstract Class]

This class is a subclass of **<stream>**. All required streams in the `Streams` library inherit from this class, which means they support the Random Access Protocol. Some implementations may provide streams that do not inherit from this class (for example, a Unix™ socket stream).

<file-stream> [Abstract Instantiable Class]

This class is a subclass of **<stream>**. When users instantiate this class, they get an implementation-dependent indirect instance that is both a **<file-stream>** and a **<random-access-stream>**. See Section *The <file-stream> Protocol* for the details of making these streams.

<string-input-stream> [Abstract Class]

This class is a subclass of **<random-access-stream>**. **Make** of **<string-input-stream>** subclasses requires a string: keyword argument that is an instance of <string>, and input operations on these streams read from the supplied string.

<bye-string-input-stream> [Concrete Class]

This class is a subclass of **<string-input-stream>**. The string: argument to **make** must be a **<byte-string>**.

<string-output-stream> [Abstract Class]

This class is a subclass of **<random-access-stream>**. These streams collect their output, and when requested, they return the output as a **<byte-string>**.

<byte-string-output-stream> [Concrete Class]

This class is a subclass of **<string-output-stream>**.

<byte-character> [Type]

This type is a subtype of **<character>**. Characters of this type represents the ASCII character set (or extensions to ASCII).

<buffer> [Concrete Class]

This class is a sealed subclass of **<vector>**. These are the buffers used by every general instance of **<stream>**. The element type of buffers is **<byte>**.

The **<buffer>** class appears to the user to be semantically the same as the **<byte-vector>** class, but implementors of the `Streams` library may make use of internal systems storage or other internal features. For example, an implementation might make buffers more suitable for directly passing them to system calls or for maintaining interior pointers into buffers, but users will be unaware o f any such implementation tactics.

<buffer-index> [Type]
This is the type of values used to index buffers.

<byte> [Type]
This type represents limited integers: `limited(<integer>, min: 0, max: 255)`.

<byte-vector> [Concrete Class]
This class is a sealed subclass of **<vector>**. The element type of byte-vectors is **<byte>**.

3. Constants

The `Streams` module exports the following constants:

$maximum-buffer-size [Constant]
This constant holds the maximum *size:* argument that users can supply when making buffers and streams.

4. The <stream> Protocol

The **<stream>** protocol categorizes operations into five groups. This section discusses four of those groups:

Basic I/O Protocol
Functions implemented on top of the Stream Extension Protocol. Users should not need to add methods to the Basic I/O Protocol functions when they define new stream subclasses.

Data Extension Protocol
Generic functions that are specialized to return or accept various classes of objects when reading or writing. Implementations are required to provide sealed methods for a few classes of data objects. Implementations are free to forgo methods for many classes of objects, and there may not even be a default method for **<object>**. The Data Extension Protocol typically specializes methods based on the classes of data objects returned by reading or accepted for writing. Sometimes these functions need to specialize on the stream as well as the data; for example, to most efficiently deliver a buffer to an output stream's destination, a method may need to be specific to a stream.

Buffer Access Protocol
Functions implemented on top of the Stream Extension Protocol. These provide the means for users to get and release streams' buffers. Users should never define new methods for these functions.

Stream Extension Protocol
Generic functions that anyone can use to extend the stream protocol to new subclasses of streams. Implementations are required to have sealed methods for this group's functions on the standard stream subclasses.

The fifth group is the Locking Protocol (see Section *The Locking Protocol*).

4.1. Basic I/O Protocol

This subsection of the **<stream>** protocol describes the Basic I/O Protocol. Users should not need to add methods to the Basic I/O Protocol functions when they define new stream subclasses. The reading and writing functions primarily only operate on bytes. This subsection describes the following functions:

read-byte
peek-byte
read-line
input-available?
flush-input
force-output
synchronize-output

read-byte [Function]

Arguments

stream :: <stream>

#key signal-eof? :: <boolean> = #t

Values

byte :: union(<byte>, singleton(#f))

Description

Returns one byte from *stream*. This function blocks until input is available. If reading from stream encounters the end of the stream, then the *signal-eof?* argument determines the behavior of this function. If *signal-eof?* is `#t` (the default), then this function signals an **<end-of-file>** condition (see Section *Conditions*); otherwise, this function returns `#f`.

peek-byte [Function]

Arguments

stream :: <stream>

values

byte :: union(<byte>, singleton(#f))

Description

Returns the next byte in the input without advancing the *stream*'s position. This function blocks until input is available. If reading from *stream* encounters the end of the stream, then this function returns `#f`.

read-line [Function]

Arguments

stream :: <stream>

#key signal-eof? :: <boolean> = #t

Values

line :: union(<string>, singleton(#f))

eof? :: <boolean>

Description

Returns as a **<byte-string>** all the input to the next newline character.[1] The resulting string excludes the newline character. This routine blocks until it encounters a newline or the end of the stream's source. As a second value, this function returns a boolean to indicate whether the line terminated with the end of the stream's source (`#t`) or a newline (`#f`).

Whenever a call to **read-line** encounters the end of *stream*'s source immediately (that is, there is no input to read), then the *signal-eof?* argument determines the behavior of this function. In this situation, if *signal-eof?* is `#t` (the default), then this function signals an **<end-of-file>** error; otherwise, it returns `#f` and `#t` as multiple values.

input-available? [Function]

Arguments

stream :: <stream>

Values

input-available? :: <boolean>

Description

Returns `#t` when *stream* has available input or when the stream is at the end of its source. If this function returns `#t`, then the next call to **read-byte** will not block. Note, though the next call to **read-byte** will not block, **read-byte** will signal an **<end-of-file>** condition (or return `#f`) if the stream is at the end of its source.

flush-input [Function]

Arguments
stream :: <stream>
Values
none
Description
Flushes all pending input from *stream*, both buffered input and, if possible, any that is available at *stream*'s source. This function returns no values.

force-output [Function]

Arguments
stream :: <stream>
Values
none
Description
Forces any pending output from *stream*'s buffer to *stream*'s destination. This function corresponds to **force-output-buffer** but provides a higher-level interface for the more common situation where the user does not hold the stream's output buffer. This function returns no values.

synchronize-output [Function]

Arguments
stream :: <stream>
Values
none
Description
Forces any pending output from *stream*'s buffer to *stream*'s destination. This function also does whatever it can to ensure the output reaches the stream's destination before returning, thereby synchronizing the output destination with the application. This function essentially calls the functions **get-output-buffer**, **empty-output-buffer**, **force-secondary-buffers**, and **synchronize**. See the definition of **synchronize** for more information. This function returns no values.

4.2. Data Extension Protocol

This subsection of the **<stream>** protocol describes the Data Extension Protocol. These functions provide higher-level reading and writing operations, and they allow users to extend reading and writing to new classes of data objects. The Data Extension Protocol typically specializes methods based on the classes of data objects returned by reading or accepted for writing. Sometimes these functions need to specialize on the stream as well as the data; for example, to most efficiently deliver a buffer to an output stream's destination, a method may need to be specific to a stream. This subsection describes the following functions:

read-as
read-into!
write
write-line

read-as [Generic Function]

Arguments
result-class :: <class>
stream :: <stream>
#key signal-eof? :: <boolean> = #t
Values
object :: union(<object>, singleton(#f))
eof? :: <object>
Description
Reads and returns an instance of *result-class* from *stream*. The second return value indicates whether reading

from *stream* encountered the end of the stream's source. Methods of **read-as** may take appropriate keywords to specify the constraints of the read. This function blocks until it can complete the read specified.

If reading from *stream* encounters the end of the stream, then the *signal-eof?* argument determines the behavior of this function. If *signal-eof?* is `#t` (the default), then this function signals an **<end-of-file>** error (see Section *Conditions*); otherwise, it returns `#f` and `#t` as multiple values.

The **read-as** function differs from **read-into!** in that **read-as** makes the object returned.

read-as [Sealed Method]

Arguments
- result-class :: <byte>
- stream :: <stream>
- #key signal-eof? :: <boolean> = #t

Values
- byte :: union(<byte>, singleton(#f))
- eof? :: <boolean>

Description

Returns a byte from *stream* according to the description of the **read-as** generic function.

read-as [Sealed Method]

Arguments
- result-class :: <byte-character>
- stream :: <stream>
- #key signal-eof? :: <boolean> = #t

Values
- char :: union(<byte-character>, singleton(#f))
- eof? :: <boolean>

Description

Returns a byte-character from *stream* according to the description of the **read-as** generic function.

read-as [Method]

Arguments
- result-class :: one-of(<byte-string>, <byte-vector>, <buffer>)[2]
- stream :: <stream>
- #key signal-eof? :: <boolean> = #t
  - count :: <fixed-integer>
  - to-eof? :: <boolean> = #f

Values
- result :: type-or(<byte-string>, <byte-vector>, <buffer>, singleton(#f))[3]
- eof?-or-how-much :: union(<boolean>, <fixed-integer>)

Description

Reads and returns an instance of *result-class* from *stream* according to the description of the **read-as** generic function, with the noted exceptions below. Implementations are required to provide sealed methods for the *result-class* values.

Supplying a *count:* argument specifies a required read. The argument is the required size of the *result*. If this method cannot satisfy the read request, then it regards *signal-eof?* according to the description of the **read-as** generic function.

Supplying the *to-eof?* argument as `#t` indicates the user wants to read all the data available up to the end of the stream's source. Supplying the *to-eof?* argument as `#t` effectively overrides the *signal-eof?* argument. In this situation, this method always returns an object of the requested type, and the second return value is always the number of bytes read. The *to-eof?* behavior exists as a convenience to support a style of reading on streams that do not adhere to the Random Access Protocol; streams that do adhere to the Random Access Protocol allow users

to compute how much data remains to be read.

This function returns buffers to support users who need to perform very big reads as efficiently as possible. Implementations should support returning buffers as directly as possible, avoiding double buffering or other unnecessary inefficiencies. Of course, any pending input in the stream's buffer must be placed in the result buffer as part of completing the read specified.

read-into! [Generic Function]

Arguments

destination :: <object>
stream :: <stream>
#key signal-eof? :: <boolean> = #t

Values

destination :: union(<object>, singleton(#f))
eof? :: <object>

Description

Fills in *destination* with input from *stream*. The second return value indicates whether reading from stream encountered the end of the stream's source. Methods of **read-into!** may take appropriate keywords to specify the constraints of the read. This function blocks until it can complete the read specified.

If reading from *stream* encounters the end of the stream, then the *signal-eof?* argument determines the behavior of this function. If *signal-eof?* is `#t` (the default), then this function signals an **<end-of-file>** error (see Section *Conditions*); otherwise, it returns `#f` and `#t` as multiple values.

read-into! [Method]

Arguments

destination :: type-or(<byte-string>, <byte-vector>, <buffer>)[3]
stream :: <stream>
#key signal-eof? :: <boolean> = #t
start :: <fixed-integer> = 0
end :: <fixed-integer> = destination.size
to-eof? :: <boolean> = #f

Values

destination :: type-or(<byte-string>, <byte-vector>, <buffer>, singleton(#f))
eof?-or-end :: union(<boolean>, <fixed-integer>)

Description

Fills in *destination* with input from *stream* according to the description of the **read-into!** generic function, with the noted exceptions below. Implementations are required to provide sealed methods for the *destination* classes.

When *to-eof?* is `#f`, the invocation specifies a required read. If this method cannot satisfy the read request designated by *start* and *end*,, then it regards *signal-eof?* according to the description of the **read-into!** generic function.

Supplying the *to-eof?* argument as `#t` indicates the user wants to read all the data available up to the end of the stream's source. Supplying the *to-eof?* argument as `#t` effectively overrides the *signal-eof?* argument and the *end* argument. In this situation, this method always returns *destination*, and the second return value is always the end of the data read into *destination*. If the stream has more than `(destination.size - start)` number of bytes available, then this method signals an error. The *to-eof?* behavior exists as a convenience to support a style of reading on streams that do not adhere to the Random Access Protocol; streams that do adhere to the Random Access Protocol allow users to compute how much data remains to be read.

This function takes a buffer to support users who need to perform very big reads as efficiently as possible. Implementations should support filling the argument buffer as directly as possible, avoiding double buffering or other unnecessary inefficiencies. Of course, any pending input in the stream's buffer must be placed in the argument buffer as part of completing the read specified.

write [Generic Function]

Arguments
object :: <object>
stream :: <stream>
#key
Values
stream :: <stream>
Description
Writes *object* to *stream*. Methods of **write** may take appropriate keywords to specify the constraints of the write.

Note, because a sealed method for **<byte>** is required, users should not extend this protocol to **<integer>**. Another library will provide more general and higher-level output functionality, such as printing **<integer>** instances in a human readable format or printing strings in a Dylan parse-able format.

write [Sealed Method]

Arguments
byte :: <byte>
stream :: <stream>
Values
stream :: <stream>
Description
Writes *byte* to *stream*.

write [Sealed Method]

Arguments
char :: <byte-character>
stream :: <stream>
Values
stream :: <stream>
Description
Writes *char* to *stream*.

write [Method]

Arguments
object :: type-or(<byte-vector>, <byte-string>, <buffer>)[3]
stream :: <stream>
#key start :: <fixed-integer> = 0
end :: <fixed-integer> = object.size
Values
stream :: <stream>
Description
Writes *object* to *stream*. Implementations are required to provide sealed methods for the specified classes of *object***:**.

This function takes a buffer to support users who need to perform very big writes as efficiently as possible. Implementations should deliver the contents of the argument buffer as directly as possible to the stream's destination, avoiding double buffering or other unnecessary inefficiencies. Of course, any pending output in the stream's buffer must be delivered to the stream's destination ahead of the argument buffer.

write-line [Generic Function]

Arguments
object :: <object>
stream :: <stream>
#key
Values
stream :: <stream>
Description
Writes *object* to stream with the **write** function, and then writes a newline character. All keyword arguments are passed to the **write** function. Multi-threaded implementations should arrange for the output of the **write** call on *object* and the newline to be contiguous.

4.3. Buffer Access Protocol

This subsection of the **<stream>** protocol describes the Buffer Access Protocol. These functions provide users the means to get and release input and output buffers, fill input buffers, force output buffers, and so on. Users should never add methods to these functions. This subsection describes the following functions:

get-input-buffer
release-input-buffer
fill-input-buffer
input-available-at-source?
get-output-buffer
release-output-buffer
empty-output-buffer
force-secondary-buffers
synchronize

These functions call their corresponding functions from the Stream Extension Protocol. For example, **get-input-buffer** calls **stream-extension-get-input-buffer**. Users always call the Buffer Access Protocol functions to operate on buffers directly, never the **stream-extension-** functions. Threaded Dylan implementations should place system-dependent mutual exclusion calls in the Buffer Access Protocol functions. The existence of these functions allows users to more portably extend the stream protocol to new streams. Users avoid the following design and maintenance hassles:

- Deciding whether to write for a threaded or a non-threaded Dylan implementation.
- Rewriting mutual exclusion code for each Dylan implementation.
- Having to duplicate mutual exclusion code wherever the application uses stream buffers directly.

Note, when Dylan has a macro facility, some of the Buffer Access Protocol will change. Instead of having separate functions to get and release buffers, there will be two macros, **with-input-buffer** and **with-output-buffer**.

get-input-buffer [Function]

Arguments
stream :: <stream>
values
buffer :: <buffer>
next :: <buffer-index>
end :: <buffer-index>
Description
Returns the input buffer for *stream*. See the definition of **stream-extension-get-input-buffer** for details on the return values and behavior of this function.

If an application calls this function, and the application already holds the input or output buffer for stream, then this function might block. Multi-threaded implementations should eventually return. All implementations are free to provide some form of recovery for environmental reasons; for example, if the environment is single-threaded, and users can cause the main stream of I/O to block due to re-entrancy or whatever, then

implementations are free to detect this and take action to keep the environment accessible to the users.

release-input-buffer [Function]

Arguments
stream :: <stream>
next :: <buffer-index>
end :: <buffer-index>

Values
none

Description
Announces that the user is done with *stream*'s buffer and updates *stream*'s state relative to the buffer. See the definition of **stream-extension-get-input-buffer** for details on the arguments. If the application does not hold the buffer, this function signals an error. This function returns no values.

fill-input-buffer [Generic Function]

Arguments
stream :: <stream>
start :: <buffer-index>

Values
end :: <buffer-index>

Description
Gets as much input as is available and that will fit in *stream*'s buffer from the *start* location to the buffer size. Because *stream* has no way to keep track of its buffer's state while users manipulate the buffer directly, users must indicate where pending input, if any, is in the buffer. The *start* argument serves this purpose and provides users the flexibility to get more input while leaving some pending input in the buffer.

This function returns the end of the newly available input. If no input is available, this function blocks until some input is available; however, when this function detects the end of the stream's source, it returns zero instead of blocking.

If the application does not hold *stream*'s buffer, this function signals an error.

input-available-at-source? [Generic Function]

Arguments
stream :: <stream>

Values
input-available? :: <boolean>

Description
Returns `#t` when *stream*'s source has any available input or when the stream is at the end of its source. If this function returns `#t`, then the next call to **fill-input-buffer** will not block. If the application does not hold *stream*'s buffer, this function signals an error.

get-output-buffer [Function]

Arguments
stream :: <stream>

Values
buffer :: <buffer>
next :: <buffer-index>
size :: <buffer-index>

Description
Returns the output buffer for *stream.* See the definition of **stream-extension-get-output-buffer** for details on the return values and behavior of this function.

If an application calls this function, and the application already holds the input or output buffer for stream, then this function might block. Multi-threaded implementations should eventually return. All implementations are

free to provide some form of recovery for environmental reasons; for example, if the environment is single-threaded, and users can cause the main stream of I/O to block due to re-entrancy or whatever, then implementations are free to detect this and take action to keep the environment accessible to the users.

release-output-buffer [Function]

Arguments

stream :: <stream>
next :: <buffer-index>

Values

none

Description

Announces that the user is done with *stream*'s buffer and updates *stream*'s state relative to the buffer. See the definition of **stream-extension-get-input-buffer** for details on the arguments. If the application does not hold the buffer, this function signals an error. This function returns no values.

empty-output-buffer [Generic Function]

Arguments

stream :: <stream>
end :: <buffer-index>

Values

none

Description

Forces out the contents of *stream*'s buffer from location zero to *end*. Because *stream* has no way to keep track of its buffer's state while users manipulate the buffer directly, users must indicate where the pending output is in the buffer. When this function returns, the user may begin placing more output in the buffer.

If the application does not hold the buffer, this function signals an error.

This function returns no values.

force-secondary-buffers [Generic Function]

Arguments

stream :: <stream>

Values

none

Description

Forces any secondary buffering that a stream may have. It is rare that a stream would require secondary buffering, but a function that directly manipulates buffers and needs to force out all output must call both **empty-output-buffer** and **force-secondary-buffers**. Most streams applications will not call this function because most streams applications can call **force-output**.

This function returns no values.

synchronize [Generic Function]

Arguments

stream :: <stream>

Values

none

Description

Forces out the contents of *stream*'s buffer from location zero to *end*. This returns only when the buffer may be used further by the application. This function also does whatever it can to ensure the output reaches the stream's destination before returning, thereby synchronizing the output destination with the application. For example, if *stream* delivered its output to an editor, calling **empty-output-buffer** and **force-secondary-buffers** would only require the stream to inject the output into the editor's text representation, but calling **synchronize** might cause the stream to invoke the editor's redisplay function.

As another example, consider an **empty-output-buffer** implementation that copied the contents of a stream's buffer and queued the copy for output to the stream's destination, possibly because delivering that output is especially slow or may incur network overhead. The implementation of **force-secondary-buffers** should cause the queued output to appear at the stream's destination eventually, even if the application exits, but the implementation of **synchronize** should wait until the queue of output becomes empty and possibly even perform an extra handshake with the destination to ensure the output was received.

If the application does not hold the buffer, this function signals an error.

This function returns no values.

4.4. Stream Extension Protocol

This subsection of the **<stream>** protocol describes the Stream Extension Protocol. These are the functions that users extend when defining new stream subclasses:

close
stream-extension-get-input-buffer
stream-extension-release-input-buffer
stream-extension-fill-input-buffer
stream-extension-input-available-at-source?
stream-extension-get-output-buffer
stream-extension-release-output-buffer
stream-extension-empty-output-buffer
stream-extension-force-secondary-buffers
stream-extension-synchronize

The make method for all stream classes takes a *size:* keyword argument that suggests the buffer size that the user thinks will be best for the stream's use. Users must still inspect the size returned when fetching the buffer because they may not get the size requested.

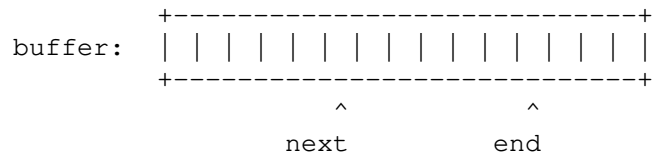
close [Generic Function]

Arguments
stream :: <stream>
Values
none
Description
Closes *stream* and potentially frees any resources backing it. If *stream* supports output, then this function forces any pending output. It is an error to call the **close** function on a stream while its buffer is held. This function returns no values.

stream-extension-get-input-buffer [Generic Function]

Arguments
stream :: <stream>
Values
buffer :: <buffer>
next :: <buffer-index>
end :: <buffer-index>
Description
Returns the input buffer for *stream*. Users of a stream object never call this function; it exists only for users to extend the stream protocol to new stream subclasses. See the definition of **get-input-buffer**.

This function also returns the stream's state relative to the buffer, which is the next available byte for input and the end of available bytes for input. The following diagram illustrates the additional return values:



Consider a buffer that has input in it from the stream's source. At any point the following is true:

- The bytes contained between locations zero, inclusively, and *next*, exclusively, have already been consumed by some user of the stream.
- The bytes contained between locations *next*, inclusively, and *end*, exclusively, have not been consumed by any user of the stream.
- The bytes contained between locations *end*, inclusively, and the size of the buffer, exclusively, are undefined.

This function may return a buffer with no available input; this is true when *next* == *end*.[4]

If *stream* is an output only stream, then this signals an error. If the application already holds the buffer, then this function signals an error.

stream-extension-release-input-buffer [Generic Function]

Arguments

stream :: <stream>
next :: <buffer-index>
end :: <buffer-index>

Values

none

Description

Announces that the user is done with *stream*'s buffer and updates *stream*'s state relative to the buffer. Users of a stream object never call this function; it exists only for users to extend the stream protocol to new stream subclasses. See the definition of **release-input-buffer**.

Because *stream* has no way to keep track of its buffer's state while users manipulate the buffer directly, users must pass information back to *stream* to indicate where pending input, if any, is in the buffer. The *next* argument indicates the location of the first byte, if any, remaining to be read. The *end* argument indicates the end of any pending input in the buffer. See **stream-extension-get-input-buffer** for details on these values.

This function returns no values.

stream-extension-fill-input-buffer [Generic Function]

Arguments

stream :: <stream>
start :: <buffer-index>

Values

end :: <buffer-index>

Description

Gets as much input as is available and that will fit in *stream*'s buffer from the *start* location to the buffer size.

This function returns the end of the newly available input. If no input is available, this function blocks until some input is available; however, when this function detects the end of the stream's source, it returns zero instead of blocking.

stream-extension-input-available-at-source? [Generic Function]

Arguments

stream :: <stream>

Values

input-available? :: <boolean>

Description

Returns `#t` when *stream*'s source has any available input or when the stream is at the end of its source. If this function returns `#t`, then the next call to **stream-extension-fill-input-buffer** will not block.

stream-extension-get-output-buffer [Generic Function]

Arguments

stream :: <stream>

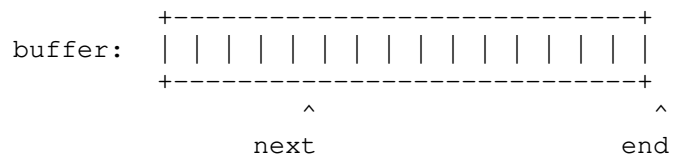
Values

buffer :: <buffer>
next :: <buffer-index>
end :: <buffer-index>

Description

Returns the output buffer for *stream*. Users of a stream object never call this function; it exists only for users to extend the stream protocol to new stream subclasses. See the definition of **get-output-buffer**.

This function also returns *stream*'s state relative to the buffer, which is the next location available to place output and the size of the buffer (the end of available locations for placing output). The following diagram illustrates the additional return values:



Consider a buffer that has pending output in it. At any point the following is true:

- The bytes contained between locations zero, inclusively, and *next*, exclusively, are pending output and need to be forced out to the stream's destination.
- The bytes contained between locations *next*, inclusively, and *end*, exclusively, are undefined.

This function never returns a full output buffer.

If *stream* is an input only stream, then this signals an error. If the application already holds the buffer, then this function signals an error.

stream-extension-release-output-buffer [Generic Function]

Arguments

stream :: <stream>
next :: <buffer-index>

Values

none

Description

Announces that the user is done with *stream*'s buffer and updates *stream*'s state relative to the buffer. Users of a stream object never call this function; it exists only for users to extend the stream protocol to new stream subclasses. See the definition of **release-output-buffer**.

Because *stream* has no way to keep track of its buffer's state while users manipulate the buffer directly, users must pass information back to *stream* to indicate where pending output, if any, is in the buffer. The *next* argument indicates the end of the pending output, if any. See **stream-extension-get-output-buffer** for details on these values.

This function returns no values.

stream-extension-empty-output-buffer [Generic Function]

Arguments
stream :: <stream>
end :: <buffer-index>

Values
none

Description
Forces out the contents of *stream*'s buffer from location zero to *end*. When this function returns, the user may begin placing more output in the buffer.

This function returns no values.

stream-extension-force-secondary-buffers [Generic Function]

Arguments
stream :: <stream>

Values
none

Description
Forces any secondary buffering that *stream* may have. This function returns no values.

stream-extension-force-secondary-buffers [Method]

Arguments
stream :: <stream>

Values
none

Description
This is the default method for **stream-extension-force-secondary-buffers**. It simply returns `#f`. This method is provided because most streams will not use secondary buffers, so most users who define new streams can avoid providing methods for this function.

stream-extension-synchronize [Generic Function]

Arguments
stream :: <stream>
end :: <buffer-index>

Values
none

Description
Forces out the contents of *stream*'s buffer from location zero to *end*. This returns only when the buffer may be used further by the application. This function also does whatever it can to ensure the output reaches the stream's destination before returning, thereby synchronizing the output destination with the application. This function returns no values. See the description of **synchronize** for more information.

5. The <random-access-stream> Protocol

This section describes the following functions from the `Streams` module:

stream-position
stream-position-setter
adjust-stream-position
stream-size

Setting or adjusting an output stream's position to be at a location before the end of the stream does not truncate the stream. The stream must support overwriting of previous output.

stream-position [Generic Function]

Arguments
stream :: <random-access-stream>
Values
position :: <integer>
Description
Returns *stream*'s position for reading or writing as the offset from position zero. If the stream's buffer is held, then this function signals an error.

stream-position-setter [Generic Function]

Arguments
position :: <integer>
stream :: <random-access-stream>
Values
position :: <integer>
Description
Sets *stream*'s position for reading or writing to be *position*.. If *position* is less than zero or greater than `stream-size(stream)`, this function signals an error. If the stream's buffer is held, then this function signals an error.

adjust-stream-position [Generic Function]

Arguments
offset :: <integer>
stream :: <random-access-stream>
#key from :: one-of(#"start", #"current", #"end")^2 = #"start"
Values
position :: <integer>
Description
Sets *stream*'s position for reading or writing to be *offset* from the *from* argument. This function returns the new absolute position in the stream. If the new absolute position is less than zero, then this function signals an error. If the new absolute position is greater than `stream-size(stream)`, then this function extends the stream's size to equal the new position, and the bytes from the old size to the new size are filled with zeroes. If the stream's buffer is held, then this function signals an error.

stream-size [Generic Function]

Arguments
stream :: <random-access-stream>
Values
size :: <integer>
Description
Returns the number of bytes in *stream* . For input streams, this is the number of bytes that were available when the stream was made. For output streams, this is the number of bytes that would be present at the stream's destination if the application were to synchronize output and close the stream. If the stream's buffer is held, then this function signals an error.

6. The <file-stream> Protocol

The `Streams` library does not provide a general file system interface, offering such operations as probing files, deleting files, renaming files, querying file authors and write dates, and so on. These kinds of operations should be provided by a separate library.

When users instantiate the **<file-stream>** class, they get an implementation-dependent indirect instance that is both a **<file-stream>** and a **<random-access-stream>**. The **make** method for **<file-stream>** and its subclasses takes the following keywords:

name:
This parameter specifies the filename to open. It must be a **<string>**.

direction:
This parameter specifies whether the stream will support input, output, or both operations. It takes the following values:

`#"input"`
Results in an input stream. This is the default. If the file does not exist, then the make method signals a **<file-not-found>** error.

`#"output"`
Results in an output stream.

`#"input-output"`
Results in a stream that supports input an output operations.

if-exists:
This parameter specifies what action to take when *direction:* is `#"output"` or `#"input-output"`, and the indicated file already exists. It takes the following values:

`#"signal"`
Signals a **<file-exists>** error.

`#"replace"` (the default)
Replaces the existing file with new output.

`#"overwrite"`
Opens the file such that output operations destructively modify the exiting file. When specifying *direction:* as `#"input-output"`, it is important to distinguish whether the system should truncate the file size to zero. Specifying *if-exists:* as `#"overwrite"` preserves the existing contents of the file as opposed to `#"replace"`. The file position will be at the start of the file.

`#"append"`
Opens the file such that output operations destructively modify the exiting file. This is the same as `#"overwrite"`, except that the file position will be at the end of the file.

7. The <string-input-stream> Protocol

The make method for **<string-input-stream>** takes a required *string:* keyword argument. The make method also takes *start:* and *end:* keywords for the string. Reading starts at *start:*, and the stream signals **<end-of-file>** when the user attempts to read past the exclusive *end:*.

8. The <string-output-stream> Protocol

The **<string-output-stream>** class supports the following operation from the `Streams` module:

string-output-stream-string [Generic Function]
Arguments
stream :: <string-output-stream>
Values
output :: <string>
Description
Returns all the output accumulated in *stream* since the last call to **string-output-stream-string**. If this function was never called on *stream* since the stream's creation, then this function returns all the output accumulated since the stream's creation. Implementations are required to provide a sealed method for **<byte-string-output-stream>** that returns a **<byte-string>**. If the stream's buffer is held, then this function signals an error.

9. The <buffer> Protocol

The **<buffer>** class is a sealed subclass of **<vector>**. This section describes the following generic functions from the `Streams` module:[5]

buffer-subsequence
copy-from-buffer!
copy-into-buffer!

Implementations should provide very fast (machine-level) byte copying for the required methods for these functions.

buffer-subsequence [Generic Function]

Arguments
- buffer :: **<buffer>**
- result-class :: **<class>**
- start :: **<buffer-index>**
- end :: **<buffer-index>**

Values
- result :: **<sequence>**

Description

Returns an instance of *result-class* using the elements from *buffer* between *start,* inclusively, and *end,* exclusively. Implementations are required to provide sealed methods for the following *result-class* values:

<byte-string>
<byte-vector>
<buffer>

It is an error to call this function with *start* or *end* values outside of `range(from: 0, through: size(`*buffer*`))`.

copy-from-buffer! [Generic Function]

Arguments
- destination :: **<sequence>**
- buffer :: **<buffer>**
- buffer-start :: **<buffer-index>**
- #key start :: **<fixed-integer>** = 0
  - end :: **<fixed-integer>** = destination.size

Values
- *none*

Description

Fills *destination* from *start* (defaults to zero) to *end* (defaults to the size of destination) with data from *buffer* that is taken starting at *buffer-start*. Implementations are required to provide sealed methods for the following *destination* classes:

<byte-string>
<byte-vector>
<buffer>

It is an error to call this function such that any attempt to index *destination* or *buffer* is out of bounds. It is an error to call this function such that it attempts to index *buffer* elements in undefined ranges, as determined by the return values from **get-input-buffer**. It is an error for *destination* and *buffer* to be the same object. This function returns no values.

copy-into-buffer! [Generic Function]

Arguments

source :: <sequence>
buffer :: <buffer>
buffer-start :: <buffer-index>
#key start :: <fixed-integer> = 0
end :: <fixed-integer> = source.size

Values

none

Description

Fills *buffer* starting at *buffer-start* with elements taken from *source* from *start* (defaults to zero) to *end* (defaults to the size of *source*). Implementations are required to provide sealed methods for the following classes:

<byte-string>
<byte-vector>
<buffer>

It is an error to call this function such that any attempt to index *source* or *buffer* is out of bounds. It is an error for *destination* and *buffer* to be the same object. This function returns no values.

10. Locking Protocol

Stream locks have multilocking semantics. A single thread may repeatedly lock a particular stream, but that thread must unlock the stream once for each time it locked the stream. This allows a high-level printing routine to lock a stream across several calls to output functions, ensuring all the output is contiguous at the stream's destination. For example, the **write-line** function locks its stream argument and then calls the **write** function twice. The **write** function locks its stream argument by calling **get-output-buffer**, but because of the multilocking semantics, the call to **write** within **write-line** does not block waiting for a lock. Before returning, **write-line** unlocks the stream so that other routines may call output functions on the stream or get the stream's buffer for direct manipulation.

Note, when Dylan has a macro facility, the Locking Protocol will change. Instead of having separate functions to lock and unlock streams, there will be a macros, **with-locked-stream**.

The `Streams` module exports the following locking functions:

lock-stream [Function]

Arguments

stream :: <stream>

Values

none

Description

Returns after successfully locking the stream. A single thread may successfully call this function repeatedly, but the thread must call **unlock-stream** once for each call to **lock-stream**. If the thread calls **unlock-stream** fewer times than **lock-stream**, the stream remains locked, and any threads waiting to lock the stream will continue to wait. When a thread that does not hold the stream's lock calls **lock-stream**, **lock-stream** puts the calling thread to sleep until the lock is available.

In singlethreaded Dylan implementations, this function does nothing.

unlock-stream [Function]

Arguments

stream :: <stream>

Values

none

Description

Releases the stream's lock. If the stream is already unlocked, this function signals an error. Only the thread that has the stream locked may call this function, and if another thread tries to unlock the stream, this function signals

an error. When **unlock-stream** returns, the stream may still be locked. A thread that has repeatedly locked the stream, must call **unlock-stream** once for each call to **lock-stream**.

In singlethreaded Dylan implementations, this function does nothing.

stream-locked? [Function]

Arguments

stream :: <stream>

Values

locked? :: <boolean>

Description

Returns whether the stream is locked. In singlethreaded Dylan implementations, this function always returns `#f`.

11. Conditions

The `Streams` module exports the following condition names and accessors:

<end-of-file>

The **<end-of-file>** condition is a subclass of **<error>**. The recovery protocol is empty. The make method takes the stream: keyword argument and stores the value in the condition object. The **end-of-file-stream** function when called on an **<end-of-file>** instance returns the value passed as the stream: argument to make. The stream object can be useful when reporting the condition to a user or distinguishing which stream ended when reading from more than one stream at a time.

<file-not-found>

The **<file-not-found>** condition is a subclass of **<error>**. The recovery protocol is empty. The make method for **<file-not-found>** takes the filename: keyword argument and stores the value in the condition object. The **file-not-found-filename** function when called on a **<file-not-found>** instance returns the value passed as the filename: argument to make.

<file-exists>

The **<file-exists>** condition is a subclass of **<error>**. The recovery protocol is empty. The make method takes the filename: keyword argument and stores the value in the condition object. The **file-exists-filename** function when called on a **<file-exists>** instance returns the value passed as the filename: argument to make.

12. Footnotes

1. The definition of **read-line** is contingent on Dylan's resolution of handling newlines, and this function may have problems on some systems.

2. **One-of** returns a type expression that represents exactly the values passed in:

```
define constant one-of =
  method (value, #rest more-values) => type :: <type>;
    reduce(union, singleton(value), map(singleton, more-values));
  end method;
```

3. **Type-or** returns a type expression that represents the union of all types passed in:

```
define constant type-or =
  method (type :: <type>, #rest more-types) => type :: <type>;
    // Ensure more-types contains only types.
    do(rcurry(check-type, <type>), more-types);
    // Make a union of all types out of Dylan's binary union function.
    reduce(union, type, more-types);
  end method;
```

4. The **stream-extension-get-input-buffer** function cannot guarantee to return a buffer with available input. Implementations will define the **input-available?** function in terms of checking the stream's buffer, and upon finding no available input there, then calling **input-available-at-source?**. If **stream-extension-get-input-buffer** guaranteed returning a buffer with input available, then it might block getting that input. If **stream-extension-get-input-buffer** could block, then **input-available?** could not be defined in terms of the Stream Extension Protocol.

5. We need these functions so that users can have reasonable semantics for copying to and from buffers. There will likely be problems when implementations want to support **<unicode-string>** because copying unicode-string elements will take two buffer elements for each unicode-string element. Ignoring extended character classes, **copy-sequence** and **replace-sequence!** have problems. Using **copy-sequence** requires wrapping the **as** function around each call, and users have to hope their Dylan implementation does the right data flow analysis and source-level transforms to extract a sequence and create the right result type with exactly one allocation and one copying of the data. If **copy-sequence** took a result class, then users could assume this would happen in one action. Using **replace-sequence!** has weird growth semantics which is inconsistent with this proposal's refusal to allow users to arbitrarily grow and replace a stream's buffer; this could be confusing to some users. For example, users are surprisingly always baffled when they use Common Lisp's **delete** function, remove the first element of a list, and then cannot figure out why the result is not **eq** to the argument (even though the documentation always warned against this assumption).