Woodchuck Documentation

Release 0.2

Neal H. Walfield

November 27, 2011

CONTENTS

Contents:

CHAPTER

ONE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHAPTER

TWO

BACKGROUND

Mobile devices promise to keep users connected. Yet, limited energy, data-transfer allowances, and cellular coverage reveal this assurance to be more a hope than a guarantee. This situation can be improved by increasing battery capacity, providing more generous data-transfer allowances, and expanding cellular coverage. We propose an alternative: modifying software to more efficiently use the available resources. In particular, many applications exhibit flexibility in when they must transfer data. For example, podcast managers can prefetch podcasts and photo sharing services can delay uploads until good conditions arise. More generally, applications that operate on data streams often have significant flexibility in when they update the stream.

More efficiently managing the available energy, the user's data-transfer allowance and data availability can improve the user experience. Increasing battery life raises the user's confidence that a charge will last the whole day, even with intense use. Alternatively, a smaller battery can be used decreasing the device's monetary cost as well as its weight and size. Explicitly managing the data-transfer allowance enables users to choose less expensive data plans without fearing that the allowance will be exceeded, which may result in expensive overage fees and bill shock, a common occurrence in the US. Finally, accounting for availability by, e.g., prefetching data, hides spotty and weak network coverage and user-perceived latency is reduced.

We have encountered two main challenges to exploiting scheduling flexibility in data streams: predicting needed data and coordinating resource consumption. First, applications need to predict when and what to prefetch. Consider Alice, who listens to the latest episode of the hourly news on her 5 PM commute home. A simple policy prefetches episodes as they are published. As Alice only listens to the 4 PM or 5 PM episode, downloading episodes as they are published wastes energy and her data-transfer allowance. An alternative policy prefetches when power and WiFi are available, typically overnight. But, Alice wants the latest episode on her commute home, not the one from 6 AM. The scheduling algorithm needs to learn when and how Alice (the individual, not an aggregate model) uses data streams. The second challenge is coordinating the use of available resources. In particular, the data-transfer allowance and local storage must be partitioned between the applications and the user. This management should not interfere with the user by, e.g., exhausting the transfer allowance so that the user cannot surf the web, or causing an out-of-space error to occur when the user saves files. Further, the allocations should adapt to the user's changing preferences.

Research on scheduling transmissions on smart phones has focused on reducing energy consumption by predicting near-term conditions. Bartendr delays transmissions until the signal strength is likely strong; TailEndr groups transmissions to amortizes energy costs; BreadCrumbs, among others, predicts WiFi availability to reduce energy spent needlessly scanning. contextual deadline, as predicted from observed user behavior. We also consider the cellular data-transmission allowance, which is an increasingly common constraint. Further, because we enable aggressive prefetching, we consider how to manage storage. Given multimedia data access patterns in which subscribed to data is used at most once, common replacement techniques, such as LRU, perform poorly.

In short, Woodchuck enables better scheduling of background data-stream updates to save energy, to make better use of data-transfer allowances, to improve disconnected operation, and to hide data-access latencies, all of which advance our ultimate goal of improving the user experience. To use Woodchuck, applications provide simple descriptions of transmission tasks. Woodchuck uses these and *predictions* of when, where and how data will be used based on application-input and historical data as well as when streams will be updated to schedule the requests so as to minimize

battery use, to respect any data-transmission allowance, and to maximize the likelihood that data that the user accesses is available. We also consider how to *manage storage* for holding prefetched data.

CHAPTER

THREE

PROGRAMMING MODEL

Before detailing Woodchuck's API, we provide a brief introduction to Woodchuck's main concepts and some case studies of how we envision some applications could exploit Woodchuck.

Woodchuck's model is relatively simple: there are managers, streams and objects. A manager represents an application (e.g., a podcast client). It contains streams. A stream represents a data source (e.g., a podcast feed). It references objects. An object represents some chunk of data (e.g., a podcast). Typically, users explicitly subscribe to a stream. The stream are regularly updated to discover new objects, which may be downloaded when convenient.

Managing the various object types is straightforward. An application registers a manager by calling ManagerRegister. Given a manager, the application registers streams using manager.StreamRegister. Similarly, an application registers objects with a stream using stream.ObjectRegister.

Associated with each object (Manager, Stream or Object) is a UUID and a cookie. The UUID uniquely identifies the object. The cookie is a free-form string that is uninterpreted by Woodchuck. It can be used by an application to store a database key or URL. This appears to greatly simplify the changes to the application as it eliminates the need for the application to manage a map between Woodchuck's UUIDs and local stream and object identifiers.

Woodchuck makes an upcall, StreamUpdate and ObjectTransfer, to the application when the application should update a stream or transfer an object, respecitvely. After updating a stream, the application invokes stream.UpdateStatus and registers any newly discovered objects using stream.ObjectRegister. ObjectTransfer tells the application to transfer an object. After attempting the transfer, the application responds by calling object.TransferStatus.

When a user uses an object, an application can report this to Woodchuck using object.Used. The application can include a bitmask representing the portions of the object that were used. This assumes that there is some serial representation as is the case with videos and books.

When space becomes scarce, Woodchuck can delete files. When an application registers an object, it can include a deletion policy, which indicates whether the object is precious and may only be deleted by the user, whether Woodchuck may delete it without consulting the application, or whether to ask the application to delete the object. In the last case, Woodchuck uses the ObjectDeleteFiles upcall. The application responds using Object.FilesDeleted indicating either: the object has been deleted; the object should be preserved for at least X seconds longer; or, the object has been shrunk. Shrinking an object is useful for data like email where an email's bulky attachments can be purged while still retaining the body.

One thing that I have not yet considered is an interface to allow applications to implement custom deletion policies. Although an application can delete a file at any file and communicate this to Woodchuck using the object.FilesDeleted interface, there is currently no mechanism for an application to say: "Tell me when there is storage pressure and I'll find the best files to delete."

3.1 Case Studies

To evaluate the applicability of the model, I've been using a few case studies: podcasts, blogs, weather and package repository updates. (Email and calendaring are similar to podcasts. Social networking (facebook, twitter, flickr) appears to be hybrid of podcasts and blogs.)

3.1.1 Podcast Manager

A podcast manager fits the proposed model very well. The podcast application registers one stream for each podcast subscription. When it updates a stream, it registers each new podcast episode as a Woodchuck object. When a podcast is viewed or listened to, it is easy to determine which parts were used.

3.1.2 Blog Reader

A blog reader is similar to the podcast application: a subscription cleanly maps to Woodchuck's stream concept and articles to Woodchuck's object. Unlike the podcast application, new objects are typically transferred inline as part of a stream update. That is, a stream update consists not of an enumeration of new objects and references, but the objects' contents. When such an application updates a stream, it registers new objects as usual and also marks them as having been transferred.

It should be relatively easy for Woodchuck to detect that the objects were delivered inline: the transfer time is the same as the stream update time. Nevertheless, I've exposed a stream property named stream.ObjectsMostInline, which an application can set if it expects this behavior.

Determining use for the application is also relatively straightforward: when an article is viewed, it has been used. It is possible to infer partial use for longer articles where scrolling is required. If the blog reader displays blogs using a continuous reader (like Google Reader), then this won't work, but it is still possible for the application to infer use based on how fast the user scrolls.

3.1.3 Package Repository Updates

At first glance, managing a package repository looks like managing podcasts. Unlike the podcast manager, prefetching most applications is useless: few users install more than dozens of applications. The few packages it makes sense to prefetch are updates to installed packages. Woodchuck can't distinguish these on its own. It is possible to teach Woodchuck this by way of object's priority property (*org.woochuck.object.Priority*). The application manager would then set this to high (e.g., 10) for packages that are installed and low (e.g., 1) for packages that are not installed. Woodchuck learns to trust the application based on actual use.

An alternative, planned approach is to provide a mechanism that allows applications to implement their own scheduling strategy. This can be down by having Woodchuck make an upcall indicating that the application should fetch the X MBs of most useful data.

If it turns out there are too many packages, just register those for which prefetching makes sense. But, always report the number of actually transferred packages when calling *org.woochuck.stream.UpdateStatus*.

3.1.4 Weather

The weather application is quite different from the podcast and blog applications. Most people, I think, are interested in monitoring a few locations at most, e.g., Baltimore and San Jose. In this case, the stream is not a series of immutable objects, but a series of object updates for a single object.

The best approach is to represent weather updates as a stream. Updating the stream means getting the latest weather. But then, the stream appears to have no objects. How do we track use? What about publication time? One solution is that after each update, the application creates a new object and marks it as having been transferred. The application should not register missed updates. Mostly likely it doesn't even know how frequently the weather is updated. To indicate that a new update is available, create a new object. If the update is only available in the future, set the object's TriggerEarliest property appropriately (*org.woochuck.object.TriggerEarliest*).

CHAPTER

FOUR

DBUS INTERFACE

Woodchuck exposes its functionality via DBus. Applications, however, do not need to use this low-level interface. Instead, there is a C library that wraps Woodchuck's functionality and Python modules. **Application developers can ignore this section** and read just about the interface they are interested in and only refer to this chapter for additional details, as required.

4.1 org.woodchuck

class `org.`**`woodchuck`**

The top-level interface to Woodchuck.

By default, Woodchuck listens on the session bus. It registers the DBus service name *org.woodchuck* and uses the object */org/woodchuck*.

`ManagerRegister`(*Properties*, *OnlyIfCookieUnique*, *UUID*)

Register a new manager.

Also see the `org.woodchuck.manager.ManagerRegister()`.

Returns The UUID of a new manager object. Manipulate the manager object using `org.woodchuck.manager` interface and the object /org/woodchuck/manager/*UUID*.

Parameters

- **Properties a{sv}** (*in*) – Dictionary of initial values for the various properties. See the `org.woodchuck.manager` interface for the list of properties and their meanings.

  The following properties are required: *HumanReadableName*

  Note: The a{ss} type is also supported, but then only properties with a string type may be expressed. (This is a concession to dbus-send, as it does not support parameters with the variant type.)

- **OnlyIfCookieUnique b** (*in*) – Only succeed if the supplied cookie is unique among all top-level managers.

- **UUID s** (*out*) – The new manager's unique identifier (a 16-character alpha-numeric string).

`ListManagers`(*Recursive*, *Managers*)

Return a list of the known managers.

Parameters

- **Recursive b** (*in*) – Whether to list all descendents (true) or just top-level manager (false).

- **Managers a(ssss)** (*out*) – An array of <*UUID*, *Cookie*, *HumanReadableName*, *ParentUUID*>.

LookupManagerByCookie (*Cookie*, *Recursive*, *Managers*)
Return the managers whose *Cookie* property matches the specified cookie.

Parameters

- **Cookie s** (*in*) – The cookie to match.
- **Recursive b** (*in*) – If true, consider any manager. If false, only consider top-level managers.
- **Managers a(sss)** (*out*) – An array of <*UUID*, *HumanReadableName*, *ParentUUID*>.

TransferDesirability (*RequestType*, *Versions*, *Desirability*, *Version*)
Evaluate the desirability of executing a transfer right now.

Parameters

- **RequestType u** (*in*) – The type of request:
  - 1: User initiated
  - 2: Application initiated
- **Versions a(xttu)** (*in*) – Array of <*ExpectedSize*, *ExpectedTransferUp*, *ExpectedTransferDown*, *Utility*> tuples. See `org.woodchuck.object.Versions` for a description.
- **Desirability u** (*out*) – The desirability of executing the job now:
  - 0: Avoid if at all possible.
  - 5: Now is acceptable but waiting is better.
  - 9: Now is ideal.
- **Version u** (*out*) – The version to transfer as an index into the passed Versions array. -1 means do not download anything.

4.2 org.woodchuck.manager

class `org.woodchuck.`**manager**
Object: /org/woodchuck/manager/*ManagerUUID*

Unregister (*OnlyIfNoDescendents*)
Unregister this manager and any descendent objects. This does not remove any files; only the metadata stored on the Woodchuck server is deleted.

Parameters **OnlyIfNoDescendents b** (*in*) – If true, fail if this manager has any descendents.

ManagerRegister (*Properties*, *OnlyIfCookieUnique*, *UUID*)
Register a new manager, which is subordinate to this one.

This enables the creation of a manager hierarchy, which is useful for separating a program's components. For instance, a web browser might have a page cache and a set of files that should be downloaded later. Each should be registered as a child manager to the top-level web browser manager.

Parameters

- **Properties a{sv}** (*in*) – Dictionary of initial values for the various properties.

  The following properties are required: *HumanReadableName*.

Note: The a{ss} type is also supported, but then only properties with a string type may be expressed. (This is a concession to dbus-send, as it does not support parameters with the variant type.)

- **OnlyIfCookieUnique b** (*in*) – Only succeed if the supplied cookie is unique among all sibling managers.
- **UUID s** (*out*) – The new manager's unique identifier (a 16-character alpha-numeric string).

ListManagers (*Recursive*, *Managers*)
Return a list of child managers.

Parameters

- **Recursive b** (*in*) – Whether to list all descendents (true) or just immediate children (false).
- **Managers a(ssss)** (*out*) – An array of <*UUID*, *Cookie*, *HumanReadableName*, *ParentUUID*>.

LookupManagerByCookie (*Cookie*, *Recursive*, *Managers*)
Return the managers whose *Cookie* property matches the specified cookie.

Parameters

- **Cookie s** (*in*) – The cookie to match.
- **Recursive b** (*in*) – If true, consider any descendent manager. If false, only consider immediate children.
- **Managers a(sss)** (*out*) – An array of <*UUID*, *HumanReadableName*, *ParentUUID*>.

StreamRegister (*Properties*, *OnlyIfCookieUnique*, *UUID*)
Register a new stream.

Parameters

- **Properties a{sv}** (*in*) – Dictionary of initial values for the various properties. See the `org.woodchuck.stream` interface for the list of properties and their meanings.

  The following properties are required: *HumanReadableName*

  Note: The a{ss} type is also supported, but then only properties with a string type may be expressed. (This is a concession to dbus-send, as it does not support parameters with the variant type.)
- **OnlyIfCookieUnique b** (*in*) – Only succeed if the supplied cookie is unique among all streams belonging to this manager.
- **UUID s** (*out*) – The new stream's unique identifier.

ListStreams (*Streams*)
Return a list of streams.

Parameters **Streams a(sss)** (*out*) – An array of <*UUID*, *Cookie*, *HumanReadableName*>.

LookupStreamByCookie (*Cookie*, *Streams*)
Return a list of streams with the specified cookie.

Parameters

- **Cookie s** (*in*) – The cookie to match.
- **Streams a(ss)** (*out*) – An array of <*UUID*, *HumanReadableName*>.

FeedbackSubscribe (*DescendentsToo*, *Handle*)
Indicate that the calling process would like to receive upcalls pertaining to this manager and (optionally) any of its descendents.

Feedback is sent until `FeedbackUnsubscribe()` is called.

Parameters

- **DescendentsToo b** (*in*) – If true, also make upcalls for any descendents.
- **Handle s** (*out*) – An opaque handle, that must be passed to `FeedbackUnSubscribe()`.

FeedbackUnsubscribe (*Handle*)
Request that Woodchuck cancel the indicated subscription.

Parameters **Handle s** (*in*) – The handle returned by `FeedbackSubscribe()`.

FeedbackAck (*ObjectUUID*, *ObjectInstance*)
Ack the feedback with the provided UUID.

Parameters

- **ObjectUUID s** (*in*) –
- **ObjectInstance u** (*in*) –

ParentUUID
This manager's parent manager.

HumanReadableName
A human readable name for the manager. When displaying a manager's human readable name, the human readable name of each of its ancestors as well as its own will be concatenated together. Thus, if the manager's parent is called "Firefox" and it has a child web cache, the human readable name of the child should be "Web Cache," not "Firefox Web Cache." The latter would result in "Firefox Firefox Web Cache" being displayed to the user.

Cookie
A free-form string uninterpreted by the server and passed to any manager upcalls.

By convention, this is set to the application's DBus name thereby allowing all application's to easily lookup the UUID of their manager and avoiding any namespace collisions.

DBusServiceName
The DBus service name of the service to start when there is work to do, e.g., streams to update or objects to transfer. See `org.woodchuck.upcall`.

DBusObject
The DBus object to send upcalls to. This defaults to '/org/woodchuck'.

Priority
The priority, relative to other managers with the same parent manager.

Enabled
Whether the manager is enabled. If a manager is disabled, any streams or objects managed by it (or any descendents) will be updated or transferred, respectively.

RegistrationTime
The time at which the object was registered.

4.3 org.woodchuck.stream

class `org.woodchuck.stream`
Object: /org/woodchuck/stream/*StreamUUID*.

`Unregister` (*OnlyIfEmpty*)
Unregister this stream and any descendent objects. This does not remove any files, only metadata stored on the Woodchuck server is deleted.

Parameters **OnlyIfEmpty b** (*in*) – If true, fail if this stream has any registered objects.

`ObjectRegister` (*Properties*, *OnlyIfCookieUnique*, *UUID*)
Register a new object.

Parameters

- **Properties a{sv}** (*in*) – Dictionary of initial values for the various properties. See the `org.woodchuck.object` interface for the list of properties and their meanings.

  No properties are required.

  Note: The a{ss} type is also supported, but then only properties with a string type may be expressed. (This is a concession to dbus-send, as it does not support parameters with the variant type.)
- **OnlyIfCookieUnique b** (*in*) – Only succeed if the supplied cookie is unique among all objects in this stream.
- **UUID s** (*out*) – The new object's unique identifier.

`ListObjects` (*Objects*)
Return a list of objects in this stream.

Parameters **Objects a(sss)** (*out*) – An array of <*UUID*, *Cookie*, *HumanReadableName*, *ParentUUID*>.

`LookupObjectByCookie` (*Cookie*, *Objects*)
Return the objects whose *Cookie* property matches the specified cookie.

Parameters

- **Cookie s** (*in*) – The cookie to match.
- **Objects a(ss)** (*out*) – An array of <*UUID*, *HumanReadableName*>.

`UpdateStatus` (*Status*, *Indicator*, *TransferredUp*, *TransferredDown*, *TransferTime*, *TransferDuration*, *NewObjects*, *UpdatedObjects*, *ObjectsInline*)
Indicate that a stream has been updated.

This is typically called in reaction to a `org.woodchuck.upcall.StreamUpdate()` upcall, but should whenever a stream is updated.

Parameters

- **Status u** (*in*) – 0: Success.

  Transient errors (will try again later):

  – 0x100: Other.

  – 0x101: Unable to contact server.

  – 0x102: Transfer incomplete.

  Hard errors (give up trying to update this stream):

  - 0x200: Other.
  - 0x201: File gone.
- **Indicator u** (*in*) – The type of indicator displayed to the user, if any. A bitmask of:
  - 0x1: Audio sound
  - 0x2: Application visual notification
  - 0x4: Desktop visual notification, small, e.g., blinking icon
  - 0x8: Desktop visual notification, large, e.g., system tray notification
  - 0x10: External visible notification, e.g., an LED
  - 0x20: Vibrate
  - 0x40: Object-specific notification
  - 0x80: Stream-wide notification, i.e., an aggregate notification for all updates in the stream.
  - 0x100: Manager-wide notification, i.e., an aggregate notification for all updates in the manager.
  - 0x80000000: It is unknown if an indicator was shown.

  0 means that no notification was shown.
- **TransferredUp t** (*in*) – The approximate number of bytes uploaded. If unknown, pass -1.
- **TransferredDown t** (*in*) – The approximate number of bytes downloaded. If unknown, pass -1.
- **TransferTime t** (*in*) – The time at which the update was started (in seconds since the epoch). Pass 0 if unknown.
- **TransferDuration u** (*in*) – The time, in seconds, it took to perform the transfer. Pass 0 if unknown.
- **NewObjects u** (*in*) – The number of new objects discovered. If not known, pass -1.
- **UpdatedObjects u** (*in*) – The objects discovered to have changes. If not known, pass -1.
- **ObjectsInline u** (*in*) – The number of inline updates. If not known, pass -1.

`ParentUUID`
The manager this streams belongs to.

`HumanReadableName`
A human readable name for the stream. When displaying a stream's human readable name, it will always be displayed with the human readable name of the manager.

`Cookie`
A free-form string uninterpreted by the server and passed to any stream upcalls.

The application can set this to a database key or URL to avoid having to manage a mapping between Woodchuck UUIDs and local identifiers.

`Priority`
The priority, relative to other streams managed by the same manager.

`Freshness`
How often the stream should be updated, in seconds.

A value of UINT32_MAX is interpretted as meaning that the stream is never updated, in which case, there is no need to check for stream updates.

ObjectsMostlyInline
Whether objects are predominantly inline (i.e., delivered with stream updates) or not. Default: False.

Consider an RSS feed for a blog: this often includes the article text. This is unlike a Podcast feed, which often just includes links to the objects' contents.

RegistrationTime
The time at which the stream was registered.

LastUpdateTime
The time at which the stream was last successfully updated.

LastUpdateAttemptTime
The time at which the last update attempt occured .

LastUpdateAttemptStatus
The status code of the last update attempt .

4.4 org.woodchuck.object

class `org.woodchuck.object`
Object: /org/woodchuck/object/*ObjectUUID*

Unregister()
Unregister this object. This does not remove any files, only metadata stored on the Woodchuck server is deleted.

Transfer(*RequestType*)
This object is needed, e.g., the user just select an email to read.

This method is only useful for object's that make use of Woodchuck's simple transferer. See `org.woodchuck.object.Versions` for more information.

Parameters **RequestType u** (*in*) – The type of request.

- 1 - User initiated
- 2 - Application initiated

TransferStatus(*Status*, *Indicator*, *TransferredUp*, *TransferredDown*, *TransferTime*, *TransferDuration*, *ObjectSize*, *Files*)
Indicate that an object has been transferred.

This is typically called in reaction to a `org.woodchuck.upcall.ObjectTransfer()` upcall, but should whenever an object is transferred.

The value of the object's `Instance` property will be incremented by 1.

Parameters

- **Status u** (*in*) – 0: Success.

  Transient errors (will try again later):

  – 0x100: Other.

  – 0x101: Unable to contact server.

  – 0x102: Transfer incomplete.

Hard errors (give up trying to transfer this object):

- 0x200: Other.
- 0x201: File gone.

- **Indicator u** (*in*) – The type of indicator displayed to the user, if any. A bitmask of:
  - 0x1: Audio sound
  - 0x2: Application visual notification
  - 0x4: Desktop visual notification, small, e.g., blinking icon
  - 0x8: Desktop visual notification, large, e.g., system tray notification
  - 0x10: External visible notification, e.g., an LED
  - 0x20: Vibrate
  - 0x40: Object-specific notification
  - 0x80: Stream-wide notification, i.e., an aggregate notification for all updates in the stream.
  - 0x100: Manager-wide notification, i.e., an aggregate notification for all updates in the manager.
  - 0x80000000: It is unknown if an indicator was shown.

  0 means that no notification was shown.
- **TransferredUp t** (*in*) – The approximate number of bytes uploaded. (Pass -1 if unknown.)
- **TransferredDown t** (*in*) – The approximate number of bytes downloaded. (Pass -1 if unknown.)
- **TransferTime t** (*in*) – The time at which the transfer was started (in seconds since the epoch). Pass 0 if unknown.
- **TransferDuration u** (*in*) – The time, in seconds, it took to perform the transfer. Pass 0 if unknown.
- **ObjectSize t** (*in*) – The size of the object on disk (in bytes). Pass -1 if unknown.
- **Files a(sbu)** (*in*) – An array of <*Filename*, *Dedicated*, *DeletionPolicy*> tuples.

  Filename is the absolute filename of a file that contains data from this object.

  Dedicated indicates whether *Filename* is dedicated to that object (true) or whether it includes other state (false).

  DeletionPolicy indicates if the file is precious and may only be deleted by the user (0), if the file may be deleted by woodchuck without consulting the application (1), or if the application is willing to delete the file (via `org.woodchuck.upcall.ObjectDeleteFiles()`) (2).

`Used`(*Start*, *Duration*, *UseMask*)

Parameters

- **Start t** (*in*) – When the user started using the object.
- **Duration t** (*in*) – How long the user used the object. -1 means unknown. 0 means instantaneous.
- **UseMask t** (*in*) – Bit mask indicating which portions of the object were used. Bit 0 corresponds to the first 1/64 of the object, bit 1 to the second 1/64 of the object, etc.

FilesDeleted(*Update*, *Arg*)
Call when an objects files have been removed or in response to org.woodchuck.upcall.ObjectDelete.

Parameters

- **Update u** (*in*) – Taken from enum woodchuck_delete_response (see <woodchuck/woodchuck.h>):
  - 0: Files deleted. ARG is ignored.
  - 1: Deletion refused. Preserve for at least ARG seconds before asking again.
  - 2: Files compressed. ARG is the new size in bytes. (-1 = unknown.)
- **Arg t** (*in*) –

ParentUUID
The stream this object belongs to.

Instance
The number of times this object has been transferred.

HumanReadableName
A human readable name.

Cookie
Uninterpretted by Woodchuck. This is passed in any object upcalls.

The application can set this to a database key or URL to avoid having to manage a mapping between Woodchuck UUIDs and local identifiers.

Versions
An array of <*URL*, *ExpectedSize*, *ExpectedTransferUp*, *ExpectedTransferDown*, *Utility*, *UseSimpleTransferer*> tuples. Each tuple designates the same object, but with a different quality.

URL is optional. Its value is only interpretted by Woodchuck if *UseSimpleTransferer* is also true.

ExpectedSize is the expected amount of disk space required when this transfer completes. If this is negative, this indicates that transferring this objects frees space.

ExpectedTransferUp is the expected upload size, in bytes.

ExpectedTransferDown is the expected download size, in bytes.

Utility is the utility of this version of the object relative to other versions of this object. Woodchuck interprets the value linearly: a version with twice the utility is consider to offer twice the quality. If bandwidth is scarce but the object is considered to have a high utility, a lower quality version may be transferred. If a version has no utility, then it shouldn't be listed here.

UseSimpleTransferer specifies whether to use Woodchuck's built in simple transferer for transferring this object. When Woodchuck has transferred an object, it will invoke the `org.woodchuck.upcall.ObjectTransferred()` upcall.

If *UseSimpleTransferer* is false, Woodchuck will make the `org.woodchuck.upcall.ObjectTransfer()` upcall to the application when the application should transfer the object. Woodchuck also specified which version of the object to transfer.

Filename
Where to save the file(s). If FILENAME ends in a /, interpreted as a directory and the file is named after the URL.

Wakeup
Whether to wake the application when this job completes (i.e., by sending a dbus message) or to wait until

a process subscribes to feedback (see `org.woodchuck.manager.FeedbackSubscribe()`). This is only meaningful if the Woodchuck server transfers the file (i.e., *UseSimpleTransferer* is true).

TriggerTarget
Approximately when the transfer should be performed, in seconds since the epoch. (If the property Period is not zero, automatically updated after each transfer.)

The special value 0 means at the next available opportunity.

TriggerEarliest
The earliest time the transfer may occur. Seconds prior to TriggerTarget.

TriggerLatest
The latest time the transfer may occur. After this time, the transfer will be reported as having failed.

Seconds after TriggerTarget.

TransferFrequency
The period (in seconds) with which to repeat this transfer. Set to 0 to indicate that this is a one-shot transfer. This is useful for an object which is updated periodically, e.g., the weather report. You should not use this for a self-contained stream such as a blog. Instead, on transferring the feed, register each contained story as an individual object and mark it as transferred immediately. Default: 0.

DontTransfer
Set to true if this object should not be transferred, e.g., because the application knows the user has no interest in it.

NeedUpdate
Set to true if an update for this object is available. This is automatically cleared by TransferStatus.

Priority
The priority, relative to other objects in the stream.

DiscoveryTime
The time at which the object was discovered (in seconds since the epoch). This is normally the time at which the stream was updated.

PublicationTime
The time at which the object was published (in seconds since the epoch).

RegistrationTime
The time at which the object was registered.

LastTransferTime
The time at which the object was last successfully transferred.

LastTransferAttemptTime
The time at which the last transfer attempt occured .

LastTransferAttemptStatus
The status code of the last transfer attempt .

4.5 org.woodchuck.upcall

class `org.woodchuck.upcall`

ObjectTransferred(*ManagerUUID*, *ManagerCookie*, *StreamUUID*, *StreamCookie*, *ObjectUUID*, *ObjectCookie*, *Status*, *Instance*, *Version*, *Filename*, *Size*, *TriggerTarget*, *TriggerFired*)

Upcall from Woodchuck indicating that a transfer has completed. After processing, the application should acknowledge the feedback using FeedbackACK, otherwise, it will be resent.

•org.freedesktop.DBus.Method.NoReply: true

Parameters

- **ManagerUUID s** (*in*) – The manager's UUID.
- **ManagerCookie s** (*in*) – The manager's cookie.
- **StreamUUID s** (*in*) – The stream's UUID.
- **StreamCookie s** (*in*) – The stream's cookie.
- **ObjectUUID s** (*in*) – The object's UUID.
- **ObjectCookie s** (*in*) – The object's cookie.
- **Status u** (*in*) – Whether the transfer was successful or not. See the status argument of `org.woodchuck.object.TransferStatus()` for the possible values.
- **Instance u** (*in*) – The number of transfer attempts (not including this one).

  This is the instance number of the feedback.
- **Version usxttub** (*in*) – Index and value of the version transfered from the versions array (at the time of transfer). See `org.woodchuck.object.Versions`.
- **Filename s** (*in*) – The location of the data.
- **Size t** (*in*) – The size (in bytes).
- **TriggerTarget t** (*in*) – The target time.
- **TriggerFired t** (*in*) – The time at which the transfer was attempted.

`StreamUpdate` (*ManagerUUID*, *ManagerCookie*, *StreamUUID*, *StreamCookie*)

Update the specified stream.

Respond by calling `org.woodchuck.stream.UpdateStatus()`.

•org.freedesktop.DBus.Method.NoReply: true

Parameters

- **ManagerUUID s** (*in*) – The manager's UUID.
- **ManagerCookie s** (*in*) – The manager's cookie.
- **StreamUUID s** (*in*) – The stream's UUID.
- **StreamCookie s** (*in*) – The stream's cookie.

`ObjectTransfer` (*ManagerUUID*, *ManagerCookie*, *StreamUUID*, *StreamCookie*, *ObjectUUID*, *ObjectCookie*, *Version*, *Filename*, *Quality*)

Transfer the specified object.

Respond by calling `org.woodchuck.object.TransferStatus()`.

•org.freedesktop.DBus.Method.NoReply: true

Parameters

- **ManagerUUID s** (*in*) – The manager's UUID.
- **ManagerCookie s** (*in*) – The manager's cookie.

- **StreamUUID s** (*in*) – The stream's UUID.
- **StreamCookie s** (*in*) – The stream's cookie.
- **ObjectUUID s** (*in*) – The object's UUID.
- **ObjectCookie s** (*in*) – The object's cookie.
- **Version (usxttub)** (*in*) – Index and value of the version to transfer from the versions array (at the time of the upcall). See `org.woodchuck.object.Versions`.
- **Filename s** (*in*) – The value of `org.woodchuck.object.Filename`.
- **Quality u** (*in*) – Target quality from 1 (most compressed) to 5 (highest available fidelity). This is useful if all possible versions cannot be or are not easily expressed by the Version parameter.

`ObjectDeleteFiles` (*ManagerUUID*, *ManagerCookie*, *StreamUUID*, *StreamCookie*, *ObjectUUID*, *ObjectCookie*, *Files*)

Delete the files associated with the specified object. Respond by calling `org.woodchuck.object.FilesDeleted()`.

•org.freedesktop.DBus.Method.NoReply: true

Parameters

- **ManagerUUID s** (*in*) – The manager's UUID.
- **ManagerCookie s** (*in*) – The manager's cookie.
- **StreamUUID s** (*in*) – The stream's UUID.
- **StreamCookie s** (*in*) – The stream's cookie.
- **ObjectUUID s** (*in*) – The object's UUID.
- **ObjectCookie s** (*in*) – The object's cookie.
- **Files a(sbu)** (*in*) – The list of files associated with this object, as provided the call to `org.woodchuck.object.TransferStatus()`.

CHAPTER

FIVE

C LIBRARY

The C library provides a more convenient interface to access Woodchuck's functionality than the low-level DBus interface. To do so, it makes a few assumption about how the streams and objects are managed. In particular, it assumes that a single application uses the specified manager and that it does so in a particular way. First, it assumes that the application only uses a top-level manager; hierarchical managers are not supported. It also assumes that streams and objects are uniquely identified by their respective cookies (thereby allowing the use of `org.woodchuck.LookupManagerByCookie()`). For most applications, these limitations should not present a burden.

The C library currently only works with programs using the glib mainloop and the gobject object system.

The C library is currently only documented in the header files <woodchuck/woodchuck.h> and <woodchuck/gwoodchuck.h>. Please refer to it for reference. Note, however, that the interface is very similar to the `PyWoodchuck` interface.

CHAPTER

SIX

PYTHON MODULES

There are two python modules for interacting with a Woodchuck server: *pywoodchuck* and *woodchuck*. *pywoodchuck* is a high-level module, which provides a Pythonic interface. It hides a fair amount of complexity while sacrificing only a small amount of functionality. It is recommended for most applications. The *woodchuck* module is a thin wrapper on top of the DBus interface.

6.1 pywoodchuck

The `pywoodchuck` module provides a high-level Pythonic interface to Python.

6.1.1 PyWoodchuck

class `pywoodchuck.`**`PyWoodchuck`**(*human_readable_name*, *dbus_service_name*, *request_feedback=True*)

A high-level, pythonic interface to Woodchuck.

This module assumes that a single application uses the specified manager and that it does so in a particular way. First, it assumes that the application only uses a top-level manager; hierarchical managers are not supported. It also assumes that streams and objects are uniquely identified by their respective cookies (thereby allowing the use of `org.woodchuck.LookupManagerByCookie()`). For most applications, these limitations should not present a burden.

If applications violate these assumptions, i.e., by manipulating the manager in an incompatible way using a low-level interface, PyWoodchuck may refuse to work with the manager.

Note: In order to process upcalls, **your application must use a main loop**. Moreover, DBus must know about the main loop. If you are using glib, **before accessing the session bus**, run:

```
from dbus.mainloop.glib import DBusGMainLoop
DBusGMainLoop(set_as_default=True)
```

or, if you are using Qt, run:

```
from dbus.mainloop.qt import DBusQtMainLoop
DBusQtMainLoop(set_as_default=True)
```

A `PyWoodchuck` instance behaves like a dictionary: iterating over it yields the streams contained therein; streams can be indexed by their stream identifier; and, stream can also be removed (`_Stream.unregister()`) using del. Note: you *cannot* register a stream by assigning a value to a key.

Registers the application with Woodchuck, if not already registered.

Parameters

- **human_readable_name** – A string that can be shown to the user that identifies the application.
- **dbus_service_name** – The application's DBus service name, e.g., org.application. This must be unique among all top-level Woodchuck managers. (This is also used as the underlying manager's cookie.) This is used by Woodchuck to start the application if it is not running by way of `org.freedesktop.DBus.StartServiceByName()`.
- **request_feedback** – Whether to request feedback, i.e., upcalls. If you say no here, you (currently) can't later enable them. If you enable upcalls, you must use a mainloop.

Example: if upcalls are not required:

```
import pywoodchuck
w = pywoodchuck.PyWoodchuck("RSS Reader", "org.rssreader")
```

Example: if you are interested in the `stream_update_cb()` and `object_transfer_cb()` upcalls:

```
import pywoodchuck

class mywoodchuck (pywoodchuck.PyWoodchuck):
    def stream_update_cb(self, stream):
        print "stream update called on %s" % (stream.identifier,)
    def object_transfer_cb(self, stream, object,
                           version, filename, quality):
        print "object transfer called on %s in stream %s" \
            % (object.identifier, stream.identifier);

w = mywoodchuck("RSS Reader", "org.rssreader")
```

The returned object behaves like a dict, which maps stream identifiers to `_Stream` objects.

available()

Returns Whether the Woodchuck daemon is available.

If the Woodchuck daemon is not available, all other methods will raise a `woodchuck.WoodchuckUnavailableError`.

Note:: Unlike nearly all other functions in pywoodchuck, this function is thread safe.

Example:

```
import pywoodchuck

w = pywoodchuck.PyWoodchuck("RSS Reader", "org.rssreader")
if not w.available ():
    print "Woodchuck functionality not available."
else:
    print "Woodchuck functionality available."
```

stream_register(*stream_identifier*, *human_readable_name*, *freshness=0*)

Register a new stream with Woodchuck.

Parameters

- **stream_identifier** – A free-form string, which is uninterpreted by the server and provided on upcalls (this is the stream's cookie). It must uniquely identify the stream within the application. It can be an application specific key, e.g., the URL of an RSS feed.

- **human_readable_name** – A string that can be shown to the user and which should unambiguously identify the stream *in the context of the application*. If the "Foo Email Client" manages a single inbox, setting human_readable_name to "Inbox" is sufficient for the user to identify the stream; "Foo Email Client: Inbox" is unnecessarily long as "Foo Email Client" is redundant.
- **freshness** – A hint to Woodchuck indicating approximately how often the stream should be updated, in seconds. (Practically, this means how often `PyWoodchuck.stream_update_cb()` will be called.) Woodchuck interprets 0 as meaning there are no freshness requirements and it is completely free to choose when to update the stream. A value of `woodchuck.never_updated` is interpretted as meaning that the stream is never updated and `PyWoodchuck.stream_update_cb()` will never be called.

Returns Returns a `_Stream` instance.

Example:

```
import pywoodchuck
import woodchuck

w = pywoodchuck.PyWoodchuck("RSS Reader", "org.rssreader")

w.stream_register("http://feeds.boingboing.net/boingboing/iBag",
                  "BoingBoing")
try:
    w.stream_register("http://feeds.boingboing.net/boingboing/iBag",
                      "BoingBoing")
except woodchuck.ObjectExistsError as exception:
    print "Stream already registered:", exception

del w["http://feeds.boingboing.net/boingboing/iBag"]
```

streams_list()

List all streams managed by this application.

Returns Returns a list of `_Stream` instances.

Example:

```
import pywoodchuck

w = pywoodchuck.PyWoodchuck("Application", "org.application")
w.stream_register("id:foo", "Foo")
w.stream_register("id:bar", "Bar")
for s in w.streams_list ():
    print "%s: %s" % (s.human_readable_name, s.identifier)
    del w[s.identifier]
```

Note: This is equivalent to iterating over the PyWoodchuck instance:

```
import pywoodchuck

w = pywoodchuck.PyWoodchuck("Application", "org.application")
w.stream_register("id:foo", "Foo")
w.stream_register("id:bar", "Bar")
for s in w.values ():
    print "%s: %s" % (s.human_readable_name, s.identifier)
    del w[s.identifier]
```

---

stream_unregister(*stream_identifier*)
Unregister the indicated stream and any objects in contains.

Note: This function is an alias for `_Stream.unregister()`:

```
pywoodchuck[stream_identifier].unregister()
```

It is also equivalent to using the del operator except instead of raising `woodchuck.NoSuchObject`, del raises `KeyError` if the object does not exist:

```
del pywoodchuck[stream_identifier].
```

---

stream_updated(*stream_identifier*, **args*, ***kwargs*)
Tell Woodchuck that a stream has been successfully updated.

Note: This function is an alias for `_Stream.updated()`:

```
pywoodchuck[stream_identifier].updated(...)
```

Parameters **stream_identifier** – The stream's identifier.

The remaining parameters are passed through to `_Stream.updated()`.

stream_update_failed(*stream_identifier*, **args*, ***kwargs*)
Tell Woodchuck that a stream update failed.

Parameters **stream_identifier** – The stream's identifier.

Note: This function is an alias for `_Stream.update_failed()`:

```
pywoodchuck[stream_identifier].update_failed(...)
```

The remaining parameters are passed through to `_Stream.update_failed()`.

object_register(*stream_identifier*, **args*, ***kwargs*)
Register an object.

Note: This function is an alias for `_Stream.object_register()`:

```
pywoodchuck[stream_identifier].object_register (...)
```

Parameters **stream_identifier** – The stream's identifier.

The remaining parameters are passed through to `_Stream.object_register()`.

objects_list(*stream_identifier*)
List the objects in a stream.

Note: This function is an alias for `_Stream.objects_list()`:

```
pywoodchuck[stream_identifier].objects_list (...)
```

And for iterating over a `_Stream` object:

for obj in pywoodchuck[stream_identifier].values (): pass

Parameters **stream_identifier** – The stream's identifier.

object_transferred(*stream_identifier*, *object_identifier*, **args*, ***kwargs*)
Tell Woodchuck that an object was successfully transferred.

Note: This function is an alias for `_Stream.object_transferred()`:

```
pywoodchuck[stream_identifier].object_transferred (...)
```

Parameters

- **stream_identifier** – The stream's identifier.
- **object_identifier** – The object's identifier.

The remaining parameters are passed through to `_Stream.object_transferred()`.

object_transfer_failed(*stream_identifier*, *object_identifier*, **args*, ***kwargs*)
Indicate that the program failed to transfer the object.

Note: This function is an alias for `_Stream.object_transfer_failed()`:

```
pywoodchuck[stream_identifier].object_transfer_failed (...)
```

Parameters

- **stream_identifier** – The stream's identifier.
- **object_identifier** – The object's identifier.

The remaining parameters are passed through to `_Stream.object_transfer_failed()`.

object_used(*stream_identifier*, *object_identifier*, **args*, ***kwargs*)
Indicate that the object has been used.

Note: This function is an alias for `_Object.used()`:

```
pywoodchuck[stream_identifier][object_identifier].used (...)
```

Parameters

- **stream_identifier** – The stream's identifier.
- **object_identifier** – The object's identifier.

The remaining parameters are passed through to `_Object.used()`.

object_files_deleted(*stream_identifier*, *object_identifier*, **args*, ***kwargs*)
Indicate that the files associated with the object have been deleted, compressed (e.g., an email attachment, but not the body, was deleted) or that a deletion request has been vetoed, because, e.g., the application thinks the user still needs the data.

Note: This function is an alias for `_Object.files_deleted()`:

```
pywoodchuck[stream_identifier][object_identifier].files_deleted (...)
```

Parameters

- **stream_identifier** – The stream's identifier.
- **object_identifier** – The object's identifier.

The remaining parameters are passed through to `_Object.files_deleted()`.

object_unregister (*stream_identifier*, *object_identifier*)
Unregister an object.

Note: This function is an alias for `_Object.unregister()`:

```
pywoodchuck[stream_identifier][object_identifier].unregister ()
```

Parameters

- **stream_identifier** – The stream's identifier.
- **object_identifier** – The object's identifier.

manager_property_get (*property*)
Get a property associated with the manager (i.e., the application).

Parameters **property** – A property, e.g., enabled.

See `stream_property_set()` for an example use of a similar function.

manager_property_set (*property*, *value*)
Set a property associated with the manager (i.e., the application).

Parameters

- **property** – A property, e.g., enabled.
- **value** – The new value.

See `stream_property_set()` for an example use of a similar function.

stream_property_get (*stream_identifier*, *property*)
Get a stream's property.

Parameters

- **stream_identifier** – The stream's identifier.
- **property** – A property, e.g., freshness.

See `stream_property_set()` for an example use of this function.

stream_property_set (*stream_identifier*, *property*, *value*)
Set a stream's property.

Parameters

- **stream_identifier** – The stream's identifier.

- **property** – A property, e.g., freshness.
- **value** – The new value.

Example:

```
import pywoodchuck
import woodchuck

w = pywoodchuck.PyWoodchuck("HMail", "org.hmail")
w.stream_register("user@provider.com/INBOX", "Provider Inbox",
                  freshness=30*60)

print w.stream_property_get ("user@provider.com/INBOX", "freshness")
w.stream_property_set ("user@provider.com/INBOX",
                       "freshness", 15*60)
print w.stream_property_get ("user@provider.com/INBOX", "freshness")
```

Note: Properties can also be get and set by accessing the equivalently named attributes. Thus, the above code could be rewritten as follows:

```
import pywoodchuck
import woodchuck

w = pywoodchuck.PyWoodchuck("HMail", "org.hmail")
w.stream_register("user@provider.com/INBOX", "Provider Inbox",
                  freshness=30*60)

print["user@provider.com/INBOX].freshness
w["user@provider.com/INBOX"].freshness = 15*60
print w["user@provider.com/INBOX"].freshness
```

---

object_property_get(*stream_identifier*, *object_identifier*, *property*)
Get an object's property.

Parameters

- **stream_identifier** – The stream identifier.
- **object_identifier** – The object's identifier.
- **property** – A property, e.g., publication_time.

See `stream_property_set()` for an example use of a similar function.

object_property_set(*stream_identifier*, *object_identifier*, *property*, *value*)
Set an object's property.

Parameters

- **stream_identifier** – The stream identifier.
- **object_identifier** – The object's identifier.
- **property** – A property, e.g., publication_time.
- **value** – The new value.

See `stream_property_set()` for an example use of a similar function.

object_transferred_cb(*stream*, *object*, *status*, *instance*, *version*, *filename*, *size*, *trigger_target*, *trigger_fired*)

Virtual method that should be implemented by the child class if it is interested in receiving object transferred notifications (`org.woodchuck.upcall.ObjectTransferred()`).

This upcall is invoked when Woodchuck transfers an object on behalf of a manager. This is only done for objects using the simple transferer.

Parameters

- **stream** – The stream, an instance of `_Stream`.
- **object** – The object, an instance of `_Object`.
- **status** – Whether the transfer was successfully. The value is taken from `woodchuck.TransferStatus`.
- **instance** – The number of transfer attempts (not including this one).
- **version** – The version that was transferred. An array of: the index in the version array, the URL, the expected size, the expected bytes uploaded, expected bytes transferred, the utility and the value of use simple transferer.
- **filename** – The name of the file containing the data.
- **size** – The size of the file, in bytes.
- **trigger_target** – The time the application requested the object be transferred.
- **trigger_fired** – The time at which the file was actually transferred.

Example: for an example of how to implement an upcall, see the opening example to `PyWoodchuck`.

stream_update_cb(*stream*)

Virtual method that should be implemented by the child class if it is interested in receiving stream update notifications (`org.woodchuck.upcall.StreamUpdate()`).

This upcall is invoked when a stream should be updated. The application should update the stream and call `stream_updated()` or `stream_update_failed()`, as appropriate.

Parameters **stream** – The stream, an instance of `_Stream`.

Example: for an example of how to implement an upcall, see the opening example to `PyWoodchuck`.

object_transfer_cb(*stream*, *object*, *version*, *filename*, *quality*)

Virtual method that should be implemented by the child class if it is interested in receiving object transfer notifications (`org.woodchuck.upcall.ObjectTransfer()`).

This upcall is invoked when an object should be transferred. The application should transfer the object and call either `object_transferred()` or `object_transfer_failed()`, as appropriate.

Parameters

- **stream** – The stream, an instance of `_Stream`.
- **object** – The object, an instance of `_Object`.
- **version** – The version to transfer. An array of: the index in the version array, the URL, the expected size, the expected bytes uploaded, expected bytes transferred, the utility and the value of use simple transferer.
- **filename** – The name of the filename property.
- **quality** – The degree to which quality should be sacrified to reduce the number of bytes transferred. The target quality of the transfer. From 1 (most compressed) to 5 (highest available fidelity).

Example: for an example of how to implement an upcall, see the opening example to PyWoodchuck.

object_delete_files_cb(*stream*, *object*)
Virtual method that should be implemented by the child class if it is interested in receiving deletion requests (org.woodchuck.upcall.ObjectDeleteFiles()).

This upcall is invoked when an object's files should be transferred. The application should respond with object_files_deleted().

Parameters

- **stream** – The stream, an instance of _Stream.
- **object** – The object, an instance of _Object.

Example: for an example of how to implement an upcall, see the opening example to PyWoodchuck.

class pywoodchuck.**_Stream**(*pywoodchuck*, *llobject*)
Encapsulates a Woodchuck stream. This object should never be explicitly instantiated by user code. Instead, use PyWoodchuck[stream_identifier] to obtain a reference to an instance.

A _Stream instance behaves like a dictionary: iterating over it yields the objects contained therein; objects can be indexed by their object identifier; and, objects can also be removed (_Object.unregister()) using del. Note: you *cannot* register an object by assigning a value to a key.

Stream properties, such as freshness, can be get and set by assigning to the like named instance attributes, e.g.:

```
stream.freshness = 60 * 60
```

Parameters

- **pywoodchuck** – The pywoodchuck instance containing the stream (an instance of PyWoodchuck).
- **llobject** – The low-level object representing the stream (an instance of woodchuck._Stream).

unregister()
Unregister the stream and any objects it contains. This just causes Woodchuck to become unaware of the stream and delete any metadata about it; this does not actually remove any objects' files.

Note: This function is eqivalent to calling:

```
del pywoodchuck[stream_identifier]
```

Example: See PyWoodchuck.stream_register() for an example use of this function.

updated(*indicator=0*, *transferred_up=None*, *transferred_down=None*, *transfer_time=None*, *transfer_duration=None*, *new_objects=None*, *updated_objects=None*, *objects_inline=None*)
Tell Woodchuck that the stream has been successfully updated. Call this function whenever the stream is successfully updated, not only in response to a stream_update_cb() upcall. If a stream update fails, this should be reported using _Stream.update_failed().

Parameters

- **indicator** – What indicators, if any, were shown to the user indicating that the stream was updated. A bit-wise mask of woodchuck.Indicator. Default: None.
- **transferred_up** – The number of bytes uploaded. If not known, set to None. Default: None.

- **transferred_down** – The number of bytes transferred. If not known, set to None. Default: None.
- **transfer_time** – The time at which the update was started (in seconds since the epoch). If not known, set to None. Default: None.
- **transfer_duration** – The amount of time the update took, in seconds. If not known, set to None. Default: None.
- **new_objects** – The number of newly discovered objects. If not known, set to None. Default: None.
- **updated_objects** – The number of objects with updates. If not known, set to None. Default: None.
- **objects_inline** – The number of objects whose content was delivered inline, i.e., with the update. If not known, set to None. Default: None.

Example of reporting a stream update for which five new objects were discovered and all of which were delivered inline:

```
import pywoodchuck
import time

w = pywoodchuck.PyWoodchuck("Application", "org.application")

w.stream_register("stream identifier", "human_readable_name")

transfer_time = int (time.time ())

# Perform the transfer

transfer_duration = int (time.time ()) - transfer_time

w["stream identifier"].updated (
    transferred_up=2048, transferred_down=64000,
    transfer_time=transfer_time,
    transfer_duration=transfer_duration,
    new_objects=5, objects_inline=5)

del w["stream identifier"]
```

Note: The five new objects should immediately be registered using `object_register()` and marked as transferred using `_Object.transferred()`.

---

update_failed(*reason*, *transferred_up=None*, *transferred_down=None*)

Tell Woodchuck that a stream update failed. Call this function whenever a stream update is attempted, not only in response to a `stream_update_cb()` upcall.

Parameters

- **reason** – The reason the update failed. Taken from `woodchuck.TransferStatus`.
- **transferred_up** – The number of bytes uploaded. If not known, set to None. Default: None.
- **transferred_down** – The number of bytes transferred. If not known, set to None. Default: None.

Example of reporting a failed stream update:

```
import pywoodchuck
import woodchuck

w = pywoodchuck.PyWoodchuck("Application", "org.application")

w.stream_register("stream identifier", "human_readable_name")

# Try to transfer the data.

w["stream identifier"].update_failed (
    woodchuck.TransferStatus.TransientNetwork,
    transferred_up=1038, transferred_down=0)

del w["stream identifier"]
```

object_register(*object_identifier*, *human_readable_name*, *transfer_frequency=None*, *expected_size=None*, *versions=None*)

Register an object.

Parameters

- **object_identifier** – The object's identifier. This must be unique among all object's in the same stream.
- **human_readable_name** – A human readable name that can be shown to the user, which is unambiguous in the context of the stream.
- **transfer_frequency** – How often the object should be transferred. If 0 or None, this is a one-shot transfer. Default: None.

Expected_size The expected amount of disk space required after this transfer completes. If this object represents an upload and space will be freed after the transfer completes, this should be negative.

Versions An array of [*URL*, *expected_size*, *expected_transfer_up*, *expected_transfer_down*, *utility*, use_simple_transferer'] specifying alternate versions of the object. *expected_size* is the expected amount of disk space required when this transfer completes. *expected_transfer_up* is the expected upload size, in bytes. *expected_transfer_down* is the expected transfer size, in bytes. *utility* is the utility of this version relative to other versions. The utility is assumed to be a linear function, i.e.,a version with 10 has twice as much value as another version with 5. *use_simple_transferer* is a boolean indicating whether Woodchuck should use its simple transferer to fetch the object.

Returns Returns a `_Object` instance.

Note: The caller may provide either *expected_size* or *versions*, but not both.

---

objects_list()

List the objects in the stream.

Returns Returns a list of `_Object` instances.

Note: This function is equivalent to iterating over the stream:

```
for obj in stream.values ():
    print obj.identifier, obj.human_readable_name
```

Example:

```
import pywoodchuck

w = pywoodchuck.PyWoodchuck("Application", "org.application")
w.stream_register("stream identifier", "human_readable_name")
w["stream identifier"].object_register(
    "object 1", "human_readable_name 1")
w["stream identifier"].object_register(
    "object 2", "human_readable_name 2")
w["stream identifier"].object_register(
    "object 3", "human_readable_name 3")

for obj in w["stream identifier"].objects_list ():
    print "%s: %s" % (obj.human_readable_name, obj.identifier)

del w["stream identifier"]["object 2"]

for obj in w["stream identifier"].objects_list ():
    print "%s: %s" % (obj.human_readable_name, obj.identifier)

del w["stream identifier"]
```

object_transferred(*object_identifier*, **args*, ***kwargs*)
Tell Woodchuck that an object was successfully transferred.

This function is a wrapper for `_Object.transferred()`. It takes one additional argument, the object's identifier. Like `_Object.transferred()`, this function marks the object as transferred. Unlike `_Object.transferred()`, if the object is not yet registered, this function first registers it setting *human_readable_name* set to *object_identifier*.

Example:

```
import pywoodchuck
import woodchuck

w = pywoodchuck.PyWoodchuck("Podcasts", "org.podcasts")
w.stream_register("http://podcast.site/podcasts/SomePodcast.rss",
                  "Some Podcast")
w["http://podcast.site/podcasts/SomePodcast.rss"].object_transferred(
    "http://podcast.site/podcasts/SomePodcast/Episode-15.ogg",
    indicator=(woodchuck.Indicator.ApplicationVisual
               |woodchuck.Indicator.DesktopSmallVisual
               |woodchuck.Indicator.ObjectSpecific),
    transferred_up=39308, transferred_down=991203,
    files=[ ["/home/user/Podcasts/SomePodcast/Episode-15.ogg",
             True,
             woodchuck.DeletionPolicy.DeleteWithoutConsultation], ])

del w["http://podcast.site/podcasts/SomePodcast.rss"]
```

object_transfer_failed(*object_identifier*, **args*, ***kwargs*)
Indicate that the program failed to transfer the object.

This function is a wrapper for `_Object.transfer_failed()`. It takes one additional argument, the object's identifier. Like `_Object.transfer_failed()`, this function marks the object as having failed to be transferred. Unlike `_Object.transfer_failed()`, if the object is not yet registered, this function first registers it setting *human_readable_name* set to *object_identifier*.

object_files_deleted(*object_identifier*, **args*, ***kwargs*)
Indicate that the files associated with the object have been deleted, compressed (e.g., an email attachment,

but not the body, was deleted) or that a deletion request has been vetoed, because, e.g., the application thinks the user still needs the data.

Note: This function is an alias for `_Object.files_deleted()`:

```
pywoodchuck[stream_identifier][object_identifier].files_deleted (...)
```

Parameters **object_identifier** – The object's identifier.

The remaining parameters are passed through to `_Object.files_deleted()`.

class pywoodchuck.**_Object**(*stream*, *llobject*)

Encapsulates a Woodchuck object. This object should never be explicitly instantiated by user code. Instead, use PyWoodchuck[stream_identifier][object_identifier] to obtain a reference to an instance.

Object properties, such as publication time, can be gotten and set by assigning to the like named instance attributes, e.g.:

```
object.publication_time = time.time ()
```

Parameters

- **stream** – The stream containing the object (an instance of `_Stream`).
- **llobject** – The low-level object representing the object (an instance of `woodchuck._Object`).

unregister()

Unregister the object. This just causes Woodchuck to become unaware of the object and delete any associated metadata; this does not actually remove any of the object's files.

Note: This function is an alias for:

```
del pywoodchuck[stream_identifier][object_identifier]
```

---

transferred(*indicator=None*, *transferred_up=None*, *transferred_down=None*, *transfer_time=None*, *transfer_duration=None*, *object_size=None*, *files=None*)

Tell Woodchuck that the object was successfully transferred.

Call this function whenever an object transfer is attempted, not only in response to a `object_transfer_cb()` upcall.

Parameters

- **indicator** – What indicators, if any, were shown to the user indicating that the stream was updated. A bit-wise mask of `woodchuck.Indicator`. Default: None.
- **transferred_up** – The number of bytes uploaded. If not known, set to None. Default: None.
- **transferred_down** – The number of bytes transferred. If not known, set to None. Default: None.
- **transfer_time** – The time at which the update was started (in seconds since the epoch). If not known, set to None. Default: None.
- **transfer_duration** – The amount of time the update took, in seconds. If not known, set to None. Default: None.

- **object_size** – The resulting on-disk size of the object, in bytes. Pass None if unknown. Default: None.
- **files** – An array of [*filename*, *dedicated*, *deletion_policy*] arrays. *filename* is the name of a file that contains data from this object; *dedicated* is a boolean indicating whether this file is dedicated to the object (True) or shared with other objects (False); *deletion_policy* is drawn from woodchuck.DeletionPolicy and indicates this file's deletion policy.

Example:

```
import pywoodchuck
import woodchuck

w = pywoodchuck.PyWoodchuck("Podcasts", "org.podcasts")
w.stream_register("http://podcast.site/SomePodcast.rss",
                  "Some Podcast")
w["http://podcast.site/SomePodcast.rss"].object_register(
    "http://podcast.site/SomePodcast/Episode-15.ogg",
    "Episode 15: Title")

# Transfer the file.

w["http://podcast.site/SomePodcast.rss"]\
    ["http://podcast.site/SomePodcast/Episode-15.ogg"].transferred(
    indicator=(woodchuck.Indicator.ApplicationVisual
               |woodchuck.Indicator.DesktopSmallVisual
               |woodchuck.Indicator.ObjectSpecific),
    transferred_up=39308, transferred_down=991203,
    files=[ ["/home/user/SomePodcast/Episode-15.ogg",
             True,
             woodchuck.DeletionPolicy.DeleteWithoutConsultation], ])

del w["http://podcast.site/SomePodcast.rss"]
```

transfer_failed(*reason*, *transferred_up=None*, *transferred_down=None*)

Indicate that the program failed to transfer the object.

Parameters

- **reason** – The reason the update failed. Taken from `woodchuck.TransferStatus`.
- **transferred_up** – The number of bytes uploaded. If not known, set to None. Default: None.
- **transferred_down** – The number of bytes transferred. If not known, set to None. Default: None.

Example: For an example of a similar function, see `_Stream.stream_update_failed()`.

used(*start=None*, *duration=None*, *use_mask=18446744073709551615L*)

Indicate that the object has been used.

Parameters

- **start** – The time that the use of the object started, in seconds since the epoch.
- **duration** – The amount of time that the object was used, in seconds.
- **use_mask** – A 64-bit bit-mask indicating which parts of the object was used. Setting the least significant bit means the first 1/64 of the object was used, the second-least significant bit that the second 1/64 of the object, etc.

Example: indicate that the user view the first 2 minutes of a 64 minute video Podcast:

```
import pywoodchuck
import time

w = pywoodchuck.PyWoodchuck("Podcasts", "org.podcasts")
w.stream_register("http://videocast.site/podcasts/Videocast.rss",
                  "Video Podcast")
w["http://videocast.site/podcasts/Videocast.rss"].object_register(
    "http://videocast.site/podcasts/Episode-15.ogv",
    "Episode 15: Title")

# User clicks play:
start = int (time.time ())
use_mask = 0
length = 64

# Periodically sample the stream's position and update use_mask.
for pos in (1, 2):
    use_mask |= 1 << int (64 * (pos / float (length)) - 1)

# User clicks stop after 2 minutes.  `use_mask` is now
# 0x3: the least two significant bits are set.
end = int (time.time ())

w["http://videocast.site/podcasts/Videocast.rss"]\
    ["http://videocast.site/podcasts/Episode-15.ogv"].used (
    start, end - start, use_mask)

del w["http://videocast.site/podcasts/Videocast.rss"]
```

files_deleted(*update=0*, *arg=None*)

Indicate that the files associated with the object have been deleted, compressed (e.g., an email attachment, but not the body, was deleted) or that a deletion request has been vetoed, because, e.g., the application thinks the user still needs the data.

Parameters

- **update** – The type of update. Taken from `woodchuck.DeletionResponse`.
- **arg** – If update is woodchuck.DeletionResponse.Deleted, the value is ignored.

  If update is woodchuck.DeletionResponse.Refused, the value is the minimum number of seconds the object should be preserved, i.e., the minimum amount of time before Woodchuck should call `Upcalls.object_delete_files_cb()` again.

  If update is woodchuck.DeletionResponse.Compressed, the value is the number of bytes of disk space the object now uses.

Example: Indicating that an email attachment has been deleted, but not the email's body:

```
import pywoodchuck
import woodchuck

w = pywoodchuck.PyWoodchuck("HMail", "org.hmail")
w.stream_register("user@provider.com/INBOX", "Provider Inbox")

w["user@provider.com/INBOX"].object_register(
    "2721812449",
    "Subject Line")
```

```
w["user@provider.com/INBOX"]["2721812449"].transferred (
    transferred_up=3308, transferred_down=991203,
    files=[ ["/home/user/Maildir/.inbox/cur/2721812449",
             True,
             woodchuck.DeletionPolicy.DeleteWithConsultation], ])

w["user@provider.com/INBOX"]["2721812449"].files_deleted (
    woodchuck.DeletionResponse.Compressed, 1877)

del w["user@provider.com/INBOX"]
```

6.2 woodchuck

The woodchuck module is a low-level wrapper of the DBus interface. Each of Woodchuck's object types is mirrored by a similarly named Python class.

The woodchuck module uses a factory for managing instantiations of the objects. In particular, the factory ensures that there is at most one Python object per Woodchuck object. That is, the same Python object is shared by all users of a given Woodchuck object.

6.2.1 Woodchuck

The Woodchuck object wraps the top-level Woodchuck interface.

woodchuck.**Woodchuck**()
Return a reference to the top-level Woodchuck singleton.

Note: There is at most a single _Woodchuck instance. In other words, the Python object is shared among all users.

class woodchuck.**_Woodchuck**(**args*, ***kwargs*)
The top-level Woodchuck class.

manager_register(**args*, ***kwargs*)
Register a new top-level manager.

Parameters

- **only_if_cookie_unique** – If True, only succeed if the specified cookie is unique among top-level managers.
- **human_readable_name** – A string that can be shown to the user that identifies the manager.
- **properties** – Other properties to set.

Returns A _Manager object.

Example:

```
import woodchuck

w = woodchuck.Woodchuck ()
manager = w.manager_register(
    only_if_cookie_unique=True,
    human_readable_name="RSS Reader",
    cookie="org.rssreader",
```

```
        dbus_service_name="org.rssreader")
    manager.unregister ()
```

list_managers(**args*, ***kwargs*)

List known managers.

Parameters **recursive** – If True, list all managers. Otherwise, only list top-level managers.

Returns An array of `_Manager`

Example:

```
import woodchuck
print "The top-level managers are:"
for m in woodchuck.Woodchuck().list_managers (False):
    print m.human_readable_name + ": " + m.cookie
```

lookup_manager_by_cookie(**args*, ***kwargs*)

Return the set of managers with the specified cookie.

Parameters

- **cookie** – The cookie to lookup.
- **recursive** – If False, only consider top-level managers, otherwise, consider any manager.

Returns An array of `_Manager`

Example:

```
import woodchuck
import random

w = woodchuck.Woodchuck()

cookie=str (random.random())
m = w.manager_register(True, cookie=cookie,
    human_readable_name="Test")

managers = w.lookup_manager_by_cookie(cookie, False)
assert len (managers) == 1
assert managers[0].UUID == m.UUID
assert managers[0].cookie == cookie
m.unregister (True)
```

6.2.2 Manager

The *_Manager* class wraps a Woodchuck manager.

woodchuck.**Manager**(***properties*)

Return a reference to a `_Manager` object. This function does not actually register a manager; a manager is assumed to already exist. This function should not normally be called from user code. Instead, call `_Woodchuck.manager_register()` or `_Woodchuck.lookup_manager_by_cookie()` to get a `_Manager` object.

Parameters

- **UUID** – The manager's UUID, required.
- **properties** – Other properties, e.g., human_readable_name. Assumed to correspond to the manager's actual values.

Returns A `_Manager` object with the specified properties.

Note: There is at most a single `_Manager` instance per Woodchuck manager object. In other words, the Python object is shared among all users.

class `woodchuck.`**`_Manager`**`(`*args*, ***kwargs*`)`

Instantiate a `Woodchuck._Manager`. Instantiating this object does not actually register a manager; the manager is assumed to already exist. A `Woodchuck._Manager` object should should not normally be directly instantiated from user code. Instead, use a method that returns an `_Manager`, such as `_Woodchuck.manager_register()` or `_Woodchuck.lookup_manager_by_cookie()` to get a `_Manager` object.

Parameters

- **UUID** – The UUID of the Manager.
- **properties** – Other properties, e.g., human_readable_name. Assumed to correspond to the manager's actual values.

`unregister``(`*args*, ***kwargs*`)`

Unregister the manager object thereby causing Woodchuck to permanently forget about the manager and any streams and objects it contained.

Parameters **only_if_empty** – If True, this method invocation only suceeds if the manager has no children, i.e., no descendent managers and no streams.

Example:

```
try:
    manager.unregister (True)
except woodchuck.NoSuchObject as exception:
    print "Can't remove stream %s: Does not exist: %s" \
        % (str (manager), exception)
except woodchuck.ObjectExistsError as exception:
    print "Can't remove manager %s: Not empty: %s" \
        % (str (manager), exception)
```

`manager_register``(`*args*, ***kwargs*`)`

Register a child manager.

Parameters

- **only_if_cookie_unique** – If True, only succeed if the specified cookie is unique among sibling managers.
- **human_readable_name** – A string that can be shown to the user that identifies the manager.
- **properties** – Other properties to set.

Returns A `_Manager` object.

Example:

```
import woodchuck

w = woodchuck.Woodchuck ()
manager = w.manager_register(
    only_if_cookie_unique=True,
    human_readable_name="Web Browser",
    cookie="org.webbrowser",
    dbus_service_name="org.webbrowser")
```

```
web_cache = manager.manager_register(
    only_if_cookie_unique=False,
    human_readable_name="Web Cache")

download_later = manager.manager_register(
    only_if_cookie_unique=False,
    human_readable_name="Downloads for Later")

manager.unregister (only_if_empty=False)
```

list_managers(**args*, ***kwargs*)

List managers that are a descendent of this one.

Parameters **recursive** – If True, list all descendent managers. Otherwise, only list managers that are an immediate descendent.

Returns An array of _Manager

See _Woodchuck.list_managers() for an example using a similar function.

lookup_manager_by_cookie(**args*, ***kwargs*)

Return the set of managers with the specified cookie that are a descendent of this one.

Parameters

- **cookie** – The cookie to lookup.
- **recursive** – If False, only consider immediate children, otherwise, consider any descendent.

Returns An array of _Manager

See _Woodchuck.lookup_manager_by_cookie() for an example.

stream_register(**args*, ***kwargs*)

Register a new stream.

Parameters

- **only_if_cookie_unique** – If True, only succeed if the specified cookie is unique.
- **human_readable_name** – A string that can be shown to the user that identifies the stream.
- **properties** – Other properties to set.

Returns A _Stream object.

Example:

```
import woodchuck
import random

w = woodchuck.Woodchuck()

cookie=str (random.random())
m = w.manager_register(True, cookie=cookie,
    human_readable_name="Test Manager")

s = m.stream_register(True, cookie=cookie,
    human_readable_name="Test Stream")

print m.list_streams ()

m.unregister (only_if_empty=False)
```

list_streams(**args*, ***kwargs*)
List this manager's streams.

Returns An array of _Stream

See _Woodchuck.list_managers() for an example using a similar function.

lookup_stream_by_cookie(**args*, ***kwargs*)
Return the set of streams with the specified cookie.

Parameters **cookie** – The cookie to match.

Returns An array of _Stream

See _Woodchuck.lookup_manager_by_cookie() for an example using a similar function.

feedback_subscribe(**args*, ***kwargs*)
Request that Woodchuck begin making upcalls for this manager.

Parameters **descendents_too** – If True, also makes upcalls for any descendent managers.

Returns An opaque handle, which must be passed to _Manager.feedback_unsubscribe().

At most, a single subscription is obtained per Manager. Thus, multiple subscriptions share the same handle. To stop receiving feedback, _Manager.feedback_unsubscribe() must be called the same number of times.

Example:

```
subscription = manager.feedback_subscribe (True)
...
manager.feedback_unsubscribe(subscription)
```

To actually receive upcalls refer to woodchuck.Upcalls.

feedback_unsubscribe(**args*, ***kwargs*)
Cancel an upcall subscription.

Parameters **handle** – The value returned by a previous call to _Manager.feedback_subscribe().

feedback_ack(**args*, ***kwargs*)
Invoke org.woodchuck.manager.FeedbackAck.

6.2.3 Stream

class woodchuck.**_Stream**(**args*, ***kwargs*)
Instantiate a Woodchuck._Stream. Instantiating this object does not actually register a stream; the stream is assumed to already exist. A Woodchuck._Stream object should should not normally be directly instantiated from user code. Instead, use a method that returns an _Stream, such as _Manager.stream_register() or _Manager.lookup_stream_by_cookie() to get a _Stream object.

Parameters

- **UUID** – The UUID of the stream.
- **properties** – Other properties, e.g., human_readable_name. Assumed to correspond to stream's actual values.

unregister(**args*, ***kwargs*)
Unregister the stream object thereby causing Woodchuck to permanently forget about the stream and any object it contained.

Parameters **only_if_empty** – If True, this method invocation only suceeds if the stream contains no objects.

Example:

```
try:
    stream.unregister (True)
except woodchuck.NoSuchObject as exception:
    print "Can't remove stream %s: Does not exist: %s"
        % (str (stream), exception)
except woodchuck.ObjectExistsError as exception:
    print "Can't remove stream %s: Not empty: %s"
        % (str (stream), exception)
```

object_register(**args*, ***kwargs*)
Register a new object.

Parameters

- **only_if_cookie_unique** – If True, only succeed if the specified cookie is unique.
- **human_readable_name** – A string that can be shown to the user that identifies the object.
- **properties** – Other properties to set.

Returns A `_Object` object.

See `_Manager.stream_register()` for an example using a similar function.

list_objects(**args*, ***kwargs*)
List this stream's objects.

Returns An array of `_Object`

See `_Woodchuck.list_managers()` for an example using a similar function.

lookup_object_by_cookie(**args*, ***kwargs*)
Return the set of objects with the specified cookie.

Parameters **cookie** – The cookie to match.

Returns An array of `_Object`

See `_Woodchuck.lookup_manager_by_cookie()` for an example using a similar function.

update_status(**args*, ***kwargs*)
Tell Woodchuck that the stream has been updated. Call this function whenever a stream is updated, not only in response to a `_Upcalls.stream_update_cb()` upcall.

Parameters

- **status** – On success, 0. Otherwise, the error code. See `TransferStatus` for possible values.
- **indicator** – What indicators, if any, were shown to the user indicating that the stream was updated. A bit-wise mask of `Indicator`. Default: None.
- **transferred_up** – The number of bytes uploaded. If not known, set to None. Default: None.
- **transferred_down** – The number of bytes transferred. If not known, set to None. Default: None.
- **transfer_time** – The time at which the update was started. If not known, set to None. Default: None.

- **transfer_duration** – The amount of time the update took, in seconds. If not known, set to None. Default: None.
- **new_objects** – The number of newly discovered objects. If not known, set to None. Default: None.
- **updated_objects** – The number of objects with updates. If not known, set to None. Default: None.
- **objects_inline** – The number of objects whose content was delivered inline, i.e., with the update. If not known, set to None. Default: None.

Example of reporting a stream update for which five new objects were discovered and all of which were delivered inline:

```
import woodchuck
import time

...

transfer_time = int (time.time ())
...
# Perform the transfer
...
transfer_duration = int (time.time ()) - transfer_time
stream.update_status (status=0,
                      transferred_up=2048,
                      transferred_down=64000,
                      transfer_time=transfer_time,
                      transfer_duration=transfer_duration,
                      new_objects=5,
                      objects_inline=5)
```

Note: The five new objects should immediately be registered using `_Stream.object_register()` and marked as transferred using `_Object.transfer_status()`.

Example of a failed update due to a network problem, e.g., the host is unreachable:

```
stream.update_status (woodchuck.TransientNetwork,
                      transferred_up=100)
```

6.2.4 Object

class `woodchuck._Object`(**args*, ***kwargs*)

The local representation for a Woodchuck object.

Instantiate a `Woodchuck._Object`. Instantiating this object does not actually register an object; the object is assumed to already exist. A `Woodchuck._Object` object should should not normally be directly instantiated from user code. Instead, use a method that returns an `_Object`, such as `_Stream.object_register()` or `_Stream.lookup_object_by_cookie()` to get a `_Object` object.

Parameters

- **UUID** – The UUID of the object.
- **properties** – Other properties, e.g., human_readable_name. Assumed to correspond to stream's actual values.

unregister(**args*, ***kwargs*)

Unregister the object object thereby causing Woodchuck to permanently forget about the object.

See `_Stream.unregister()` for an example using a similar function.

transfer(**args*, ***kwargs*)

Request that Woodchuck transfer the object. This only makes sense for object's that use Woodchuck's simple transferer.

Parameters **request_type** – Whether the request is user initiated or application initiated. See `TransferStatus` for possible values.

transfer_status(**args*, ***kwargs*)

Tell Woodchuck that the object has been transferred. Call this function whenever an object is transferred (or uploaded), not only in response to a `_Upcalls.object_transfer_cb()` upcall.

Parameters

- **status** – On success, 0. Otherwise, the error code. See `TransferStatus` for possible values.
- **indicator** – What indicators, if any, were shown to the user indicating that the stream was updated. A bit-wise mask of `Indicator`. Default: None.
- **transferred_up** – The number of bytes uploaded. If not known, set to None. Default: None.
- **transferred_down** – The number of bytes transferred. If not known, set to None. Default: None.
- **transfer_time** – The time at which the update was started. If not known, set to None. Default: None.
- **transfer_duration** – The amount of time the update took, in seconds. If not known, set to None. Default: None.
- **object_size** – The amount of disk space used by the object, in bytes. If not known, set to None. Default: None.
- **files** – The files belong to the object. An array of arrays consisting of a filename (a string), a boolean indicating whether the file is exclusive to the object, and the file's deletion policy (see `woodchuck.DeletionPolicy` for possible values).

Example of reporting an object transfer for an object that Woodchuck can deleted without consulting the user:

```
transfer_time = int (time.time ())
...
# Perform the transfer
...
transfer_duration = int (time.time ()) - transfer_time
stream.update_status(
    status=0,
    transferred_up=4096,
    transferred_down=1024000,
    transfer_time=transfer_time,
    transfer_duration=transfer_duration,
    files=( ("/home/user/Podcasts/Foo/Episode1.ogg", True,
             woodchuck.DeletionPolicy.DeleteWithoutConsultation),))
```

used(**args*, ***kwargs*)

Mark the object as having been used.

Parameters

- **start** – The time at which the user started using the object. If unknown, pass None. Default: None.
- **duration** – The amount of time the user used the object, in seconds. If unknown, pass None. Default: None.
- **use_mask** – A 64-bit mask indicating the parts of the object that were used. Setting the least-significant bit means that the first 1/64 of the object was used, the second bit means that the second 1/64 of the object was used, etc. If unknown, pass None. Default: None.

Example: Indicate that that the first two minutes of an hour-long video were viewed:

```
object.used(start_time, 120, 0x3)
```

files_deleted(**args*, ***kwargs*)

Indicate that some or all of the object's files have been deleted. This should be called whenever an object's files are deleted, not only in response to `Upcalls.object_delete_files_cb()`.

Parameters

- **update** – Taken from `woodchuck.DeletionResponse`.
- **arg** – If update is DeletionResponse.Deleted, the value is ignored.

  If update is DeletionResponse.Refused, the value is the minimum number of seconds the object should be preserved, i.e., the minimum amount of time before Woodchuck should call `Upcalls.object_delete_files_cb()` again.

  If update is DeletionResponse.Compressed, the value is the number of bytes of disk space the object now uses.

Example: An email's attachments are purged, but the body is preserved:

```
object.files_deleted (woodchuck.DeletionResponse.Compressed,
                      2338)
```

6.2.5 Upcall

class woodchuck.**Upcalls**(**args*, ***kwargs*)

A thin wrapper around org.woodchuck.upcalls.

To use this class, implement your own class, which inherits from this one and overrides the virtual methods of the upcalls that you are interested in (`Upcalls.object_transferred_cb()`, `Upcalls.stream_update_cb()`, `Upcalls.object_transfer_cb()` and `object_delete_files_cb()`). Instantiate the class and then call `woodchuck.feedback_subscribe()` to begin receiving feedback.

Example:

```
class Upcalls(woodchuck.Upcalls):
    def object_transferred_cb (self, **kwargs):
        # Transfer the kwargs[object_UUID] object.
        ...
upcalls = Upcalls ()
subscription = Manager.feedback_subscribe (False)
```

Note: In order to process upcalls, **your application must use a main loop**. Moreover, DBus must know about the main loop. If you are using glib, **before accessing the session bus**, run:

from dbus.mainloop.glib import DBusGMainLoop DBusGMainLoop(set_as_default=True)

or, if you are using Qt, run:

from dbus.mainloop.qt import DBusQtMainLoop DBusQtMainLoop(set_as_default=True)

Parameters **path** – The object that will receive the upcalls from woodchuck.

object_transferred_cb(*manager_UUID*, *manager_cookie*, *stream_UUID*, *stream_cookie*, *object_UUID*, *object_cookie*, *status*, *instance*, *version*, *filename*, *size*, *trigger_target*, *trigger_fired*)

Virtual method that should be implemented by the child class if it is interested in org.woodchuck.upcall.ObjectTransferred upcalls.

This upcall is invoked when Woodchuck transfers an object on behalf of a manager. This is only done for objects using the simple transferer.

Parameters

- **manager_UUID** – The manager's UUID.
- **manager_cookie** – The manager's cookie.
- **stream_UUID** – The stream's UUID.
- **stream_cookie** – The stream's cookie.
- **object_UUID** – The object's UUID.
- **object_cookie** – The object's cookie.
- **status** – Whether the transfer was successfully. The value is taken from `woodchuck.TransferStatus`.
- **instance** – The number of transfer attempts (not including this one).
- **version** – The version that was transferred. An array of: the index in the version array, the URL, the expected object size on disk (negative if transferring the object will free space), the expected upload size, the expected transfer size, the utility and the value of use simple transferer.
- **filename** – The name of the file containing the data.
- **size** – The size of the file, in bytes.
- **trigger_target** – The time the application requested the object be transferred.
- **trigger_fired** – The time at which the file was actually transferred.

stream_update_cb(*manager_UUID*, *manager_cookie*, *stream_UUID*, *stream_cookie*)

Virtual method that should be implemented by the child class if it is interested in org.woodchuck.upcall.StreamUpdate upcalls.

This upcall is invoked when a stream should be updated. The application should update the stream and call `_Stream.update_status()`.

Parameters

- **manager_UUID** – The manager's UUID.
- **manager_cookie** – The manager's cookie.
- **stream_UUID** – The stream's UUID.
- **stream_cookie** – The stream's cookie.

object_transfer_cb(*manager_UUID*, *manager_cookie*, *stream_UUID*, *stream_cookie*, *object_UUID*, *object_cookie*, *version*, *filename*, *quality*)

Virtual method that should be implemented by the child class if it is interested in org.woodchuck.upcall.ObjectTransfer upcalls.

This upcall is invoked when Woodchuck transfers an object on behalf of a manager. This is only done for objects using the simple transferer.

Parameters

- **manager_UUID** – The manager's UUID.
- **manager_cookie** – The manager's cookie.
- **stream_UUID** – The stream's UUID.
- **stream_cookie** – The stream's cookie.
- **object_UUID** – The object's UUID.
- **object_cookie** – The object's cookie.
- **version** – The version to transfer. the index in the version array, the URL, the expected object size on disk (negative if transferring the object will free space), the expected upload size, the expected transfer size, the utility and the value of use simple transferer.
- **filename** – The name of the filename property.
- **quality** – The degree to which quality should be sacrified to reduce the number of bytes transferred. The target quality of the transfer. From 1 (most compressed) to 5 (highest available fidelity).

object_delete_files_cb(*manager_UUID*, *manager_cookie*, *stream_UUID*, *stream_cookie*, *object_UUID*, *object_cookie*)

Virtual method that should be implemented by the child class if it is interested in org.woodchuck.upcall.ObjectDeleteFiles upcalls.

This upcall is invoked when Woodchuck wants a manager to free disk space.

Parameters

- **manager_UUID** – The manager's UUID.
- **manager_cookie** – The manager's cookie.
- **stream_UUID** – The stream's UUID.
- **stream_cookie** – The stream's cookie.
- **object_UUID** – The object's UUID.
- **object_cookie** – The object's cookie.

6.2.6 Constants

woodchuck.**never_updated**

A low-level wrapper of the org.woodchuck DBus interfaces.

class woodchuck.**RequestType**

Values for the request_type argument of `_Object.transfer()`.

UserInitiated

The user initiated the transfer request.

ApplicationInitiated
The application initiated the transfer request.

class woodchuck.**TransferStatus**
Values for the Indicator argument of woodchuck._Object.transfer_status(), woodchuck._Stream.update_status() and woodchuck.Upcalls.object_transferred_cb().

Success
The transfer was successful.

TransientOther
An unspecified transient error occurred.

TransientNetwork
A transient network error occured, e.g., the host was unreachable.

TransientInterrupted
A transient error occured during the transfer.

FailureOther
An unspecified hard error occured. Don't try again.

FailureGone
A hard error, the object is gone, occured.

class woodchuck.**Indicator**
Values for the Indicator argument of woodchuck._Object.transfer_status(), woodchuck._Stream.update_status().

Audio
An audio sound was emitted.

ApplicationVisual
An visual notification was displayed in the application.

DesktopSmallVisual
A small visual notification was displayed on the desktop, e.g., in the system tray.

DesktopLargeVisual
A large visual notification was displayed on the desktop.

ExternalVisual
An external visual notification was displayed, e.g., an LED was blinked.

Vibrate
The device vibrated.

ObjectSpecific
The notification was object specific.

StreamWide
The notification was stream-wide, i.e., an aggregate notification for all updates in the stream.

ManagerWide
The notification was manager-wide, i.e., an aggregate : notification for multiple stream updates.

Unknown
It is unknown whether an indicator was shown.

class woodchuck.**DeletionPolicy**
Values for the *deletion_policy* argument of woodchuck._Object.transfer_status().

Precious
The file is precious and will only be deleted by the user.

DeleteWithoutConsultation
Woodchuck may delete the file without consulting the application.

DeleteWithConsultation
Woodchuck may ask the application to delete the file.

class woodchuck.**DeletionResponse**
Values for the Update arguments of woodchuck._Object.files_deleted()

Deleted
The files associated with the object were deleted.

Refused
The application refuses to delete the object.

Compressed
The application compressed the object, e.g., for an email, it

6.2.7 Exceptions

exception woodchuck.**Error**
Base class for exceptions in this model. args[0] contains a more detailed description of the error.

exception woodchuck.**GenericError**
While invoking a Woodchuck method, a DBus error org.woodchuck.GenericError occured.

exception woodchuck.**NoSuchObject**
While invoking a Woodchuck method, a DBus error org.freedesktop.DBus.Error.UnknownObject occured.

exception woodchuck.**ObjectExistsError**
While invoking a Woodchuck method, a DBus error org.woodchuck.ObjectExists occured.

exception woodchuck.**NotImplementedError**
While invoking a Woodchuck method, a DBus error org.woodchuck.MethodNotImplemented occured.

exception woodchuck.**InternalError**
While invoking a Woodchuck method, a DBus error org.woodchuck.InternalError occured.

exception woodchuck.**InvalidArgsError**
While invoking a Woodchuck method, a DBus error org.woodchuck.InvalidArgs occured.

exception woodchuck.**UnknownError**
While invoking a Woodchuck method, an unknown DBus error with prefix org.woodchuck occured.

exception woodchuck.**WoodchuckUnavailableError**
The woodchuck server is unavailable. For whatever reason, it couldn't be started. This is a Python specific exception.

PYTHON MODULE INDEX

INDEX

O

P

R

S

T

U

V

W