Chapter 1

Overview

The etar library provides support for archiving data using the pax/tar format.

1.1 Concepts

1.1.1 Archive

An archive is a series of blocks of size 512 bytes. An entry consists of exactly one header block, followed by zero or more payload blocks. At the end of the archive there are two consecutive blocks with all zero bytes.

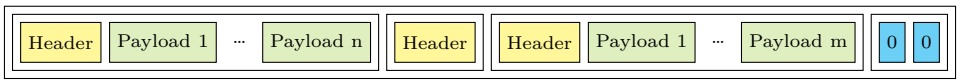

Figure 1.1: Structure of an archive

1.2 Error Handling

API still unstable

Chapter 2

ARCHIVE

The ARCHIVE class is the central piece of this library. It allows clients to manipulate archives.

2.1 Initialization

To create a new instance of the ARCHIVE class, the client has to use the make feature and provide a STORAGE_BACKEND which will then be used to do I/O. A client then can either open the archive for archiving or for unarchiving. It is not possible to archive and unarchive simultaneously.

2.2 Archiving

To use the archiving mode, one has to use the `open_archive` feature of ARCHIVE. Then, one can add entries using `add_entry`. Once all entries are written, the user has to call finalize, which will write the end-of-archive indicator and then close the archive.

2.3 Unarchiving

To use the unarchiving mode, one has to call `open_unarchive` after creation. Next, the client may register arbitrarily many UNARCHIVERs using `add_unarchiver`. Once all of them were added, the client may either use `unarchive` which will then unarchive all entries and close the archive or `unarchive_next_entry` until `unarchiving_finished` becomes True. In the latter case, the client has to call `close` himself.

Chapter 3

STORAGE_BACKEND

STORAGE_BACKEND provides a unified interface for different storage methods an archive could use. Currently the only implementation is FILE_STORAGE_BACKEND, providing support for archives that are stored in a file.

3.1 FILE_STORAGE_BACKEND

A FILE_STORAGE_BACKEND is either created from a file with `make_from_file` or from a filename with `make_from_filename`.

3.2 Implementing a Custom STORAGE_BACKEND

To implement a custom STORAGE_BACKEND, one has to implement the following features:

3.2.1 Creation Procedures

If `default_create` is redefined, `Precursor` must be called. Every other creation procedure should call `default_create`.

3.2.2 open_read

`open_read`
Open backend for read access. Reading should start from the beginning.

3.2.3 open_write

`open_write`
Open backend for write access. Writing should start from the beginning.

3.2.4 close

`close`
Close backend.

3.2.5 archive_finished

`archive_finished: BOOLEAN`
Indicate whether the next two blocks contain the end-of-archive indicator (only zero bytes). The next `read_block` calls should not skip these two blocks but read them again (not necessarily from the backend again, the implementation is free to chache these blocks). `archive_finished` should return `True` too, if an error occured (or occurs while checking for the end-of-archive indicator), does not have enough blocks available or if the backend is closed.

3.2.6 block_ready

`block_ready: BOOLEAN`
Indicate whether there is a block that can be read with `last_block`. `False` if an error occured.

3.2.7 is_readable

`is_readable: BOOLEAN`
Indicates whether this backend can be read from. If an error occured, this has to return `False`

3.2.8 is_writable

`is_writable: BOOLEAN`
Indicates whether this backend can be written to. If an error occured, this has to return `False`

3.2.9 is_closed

`is_closed: BOOLEAN`
Indicates whether this backend is closed.

3.2.10 read_block

`read_block`
Read next block from backend. If there are not enough bytes for a full block, an error should be reported.

3.2.11 last_block

`last_block: MANAGED_POINTER`
Last block that was read.

3.2.12 write_block

`write_block (block: MANAGED_POINTER)`
Write `block` to the backend (starting from the beginning).

3.2.13 finalize

`finalize`
Write the end-of-archive indicator and close backend.

Chapter 4

ARCHIVABLE

Everything that one wants to add to an archive has to inherit from ARCHIVABLE, which provides an interface that ARCHIVE uses to write it. The etar library provides two implementations.

4.1 FILE_ARCHIVABLE

FILE_ARCHIVABLE allows to archive plain files. A client has to provide a FILE for which the FILE_ARCHIVABLE will be created.

4.2 DIRECTORY_ARCHIVABLE

DIRECTORY_ARCHIVABLE allows to archive a directory (without its contents!). On creation the client has to provide a FILE (!) (for which is_directory holds).

4.3 Implementing a custom ARCHIVABLE

To implement a custom ARCHIVABLE, one has to implement the following features:

4.3.1 Creation Procedures

There is nothing to consider for creation procedures.

4.3.2 required_blocks

required_blocks: INTEGER
Has to return how many blocks are needed to archive the payload.

4.3.3 header

header: TAR_HEADER
Has to return a TAR_HEADER object suitable for the archivable type and the payload.

4.3.4 write_block_to_managed_pointer

`write_block_to_managed_pointer (p: MANAGED_POINTER; a_pos: INTEGER)`
Has to write the next block to `p` (writing should start at position `a_pos`). This feature has to increase `written_blocks` by one. In case the payload does not fill a whole block, it has to be padded to full block size (`{TAR_CONST}.tar_block_size`).

4.3.5 write_to_managed_pointer

`write_to_managed_pointer (p: MANAGED_POINTER; a_pos: INTEGER)`
Has to write the whole payload (padded to `{TAR_CONST}.tar_block_size`) bytes. Calling this feature must not change the state of blockwise writing.

4.3.6 Utility Features

To implement the features listed above, the following utility features are provided:

Padding

To pad a block to some size, one can use `pad (p: MANAGED_POINTER; a_pos, n: INTEGER)`
It pads a given block `p` with `n` zero-bytes, starting from position `a_pos`. If `n` is zero, nothing will be padded (but it's legal to call it with `n = 0`).

4.3.7 End of Payload

To determine whether the last payload block was written, one can compare `required_blocks` with `written_blocks`

4.3.8 Bytes to Blocks

The feature `needed_blocks (n: INTEGER): INTEGER)` can be used to determine how many blocks are required to store `n` bytes.

Chapter 5

UNARCHIVER