NAME

libarchive — functions for reading and writing streaming archives

LIBRARY

Streaming Archive Library (libarchive, −larchive)

OVERVIEW

The **libarchive** library provides a flexible interface for reading and writing streaming archive files such as tar and cpio. The library is inherently stream-oriented; readers serially iterate through the archive, writers serially add things to the archive. In particular, note that there is no built-in support for random access nor for in-place modification.

When reading an archive, the library automatically detects the format and the compression. The library currently has read support for:

- old-style tar archives,
- most variants of the POSIX "ustar" format,
- the POSIX "pax interchange" format,
- GNU-format tar archives,
- most common cpio archive formats,
- ISO9660 CD images (with or without RockRidge extensions),
- Zip archives.

The library automatically detects archives compressed with gzip(1), bzip2(1), or compress(1) and decompresses them transparently.

When writing an archive, you can specify the compression to be used and the format to use. The library can write

- POSIX-standard "ustar" archives,
- POSIX "pax interchange format" archives,
- POSIX octet-oriented cpio archives,
- two different variants of shar archives.

Pax interchange format is an extension of the tar archive format that eliminates essentially all of the limitations of historic tar formats in a standard fashion that is supported by POSIX-compliant pax(1) implementations on many systems as well as several newer implementations of tar(1). Note that the default write format will suppress the pax extended attributes for most entries; explicitly requesting pax format will enable those attributes for all entries.

The read and write APIs are accessed through the **archive_read_XXX**() functions and the **archive_write_XXX**() functions, respectively, and either can be used independently of the other.

The rest of this manual page provides an overview of the library operation. More detailed information can be found in the individual manual pages for each API or utility function.

READING AN ARCHIVE

To read an archive, you must first obtain an initialized struct archive object from **archive_read_new**(). You can then modify this object for the desired operations with the various **archive_read_set_XXX**() and **archive_read_support_XXX**() functions. In particular, you will need to invoke appropriate **archive_read_support_XXX**() functions to enable the corresponding compression and format support. Note that these latter functions perform two distinct operations: they cause the corresponding support code to be linked into your program, and they enable the corresponding auto-detect code. Unless you have specific constraints, you will generally want to invoke **archive_read_support_compression_all**() and **archive_read_support_format_all**() to enable auto-detect for all formats and compression types currently supported by the library.

Once you have prepared the struct archive object, you call **archive_read_open**() to actually open the archive and prepare it for reading. There are several variants of this function; the most basic expects you to provide pointers to several functions that can provide blocks of bytes from the archive. There are convenience forms that allow you to specify a filename, file descriptor, *FILE* * object, or a block of memory from which to read the archive data. Note that the core library makes no assumptions about the size of the blocks read; callback functions are free to read whatever block size is most appropriate for the medium.

Each archive entry consists of a header followed by a certain amount of data. You can obtain the next header with **archive_read_next_header**(), which returns a pointer to an struct archive_entry structure with information about the current archive element. If the entry is a regular file, then the header will be followed by the file data. You can use **archive_read_data**() (which works much like the read(2) system call) to read this data from the archive. You may prefer to use the higher-level **archive_read_data_skip**(), which reads and discards the data for this entry, **archive_read_data_to_buffer**(), which reads the data into an in-memory buffer, **archive_read_data_to_file**(), which copies the data to the provided file descriptor, or **archive_read_extract**(), which recreates the specified entry on disk and copies data from the archive. In particular, note that **archive_read_extract**() uses the struct archive_entry structure that you provide it, which may differ from the entry just read from the archive. In particular, many applications will want to override the pathname, file permissions, or ownership.

Once you have finished reading data from the archive, you should call **archive_read_close**() to close the archive, then call **archive_read_finish**() to release all resources, including all memory allocated by the library.

The archive_read(3) manual page provides more detailed calling information for this API.

WRITING AN ARCHIVE

You use a similar process to write an archive. The **archive_write_new**() function creates an archive object useful for writing, the various **archive_write_set_XXX**() functions are used to set parameters for writing the archive, and **archive_write_open**() completes the setup and opens the archive for writing.

Individual archive entries are written in a three-step process: You first initialize a struct archive_entry structure with information about the new entry. At a minimum, you should set the pathname of the entry and provide a *struct stat* with a valid *st_mode* field, which specifies the type of object and *st_size* field, which specifies the size of the data portion of the object. The **archive_write_header**() function actually writes the header data to the archive. You can then use **archive_write_data**() to write the actual data.

After all entries have been written, use the **archive_write_finish**() function to release all resources.

The archive_write(3) manual page provides more detailed calling information for this API.

DESCRIPTION

Detailed descriptions of each function are provided by the corresponding manual pages.

All of the functions utilize an opaque struct archive datatype that provides access to the archive contents.

The struct archive_entry structure contains a complete description of a single archive entry. It uses an opaque interface that is fully documented in archive_entry(3).

Users familiar with historic formats should be aware that the newer variants have eliminated most restrictions on the length of textual fields. Clients should not assume that filenames, link names, user names, or group names are limited in length. In particular, pax interchange format can easily accommodate pathnames in arbitrary character sets that exceed *PATH_MAX*.

RETURN VALUES

Most functions return zero on success, non-zero on error. The return value indicates the general severity of the error, ranging from **ARCHIVE_WARN**, which indicates a minor problem that should probably be reported to the user, to **ARCHIVE_FATAL**, which indicates a serious problem that will prevent any further operations on this archive. On error, the **archive_errno**() function can be used to retrieve a numeric error code (see errno(2)). The **archive_error_string**() returns a textual error message suitable for display.

archive_read_new() and **archive_write_new**() return pointers to an allocated and initialized struct archive object.

archive_read_data() and **archive_write_data**() return a count of the number of bytes actually read or written. A value of zero indicates the end of the data for this entry. A negative value indicates an error, in which case the **archive_errno**() and **archive_error_string**() functions can be used to obtain more information.

ENVIRONMENT

There are character set conversions within the archive_entry(3) functions that are impacted by the currently-selected locale.

SEE ALSO

tar(1), archive_entry(3), archive_read(3), archive_util(3), archive_write(3), tar(5)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

Some archive formats support information that is not supported by struct archive_entry. Such information cannot be fully archived or restored using this library. This includes, for example, comments, character sets, or the arbitrary key/value pairs that can appear in pax interchange format archives.

Conversely, of course, not all of the information that can be stored in an struct archive_entry is supported by all formats. For example, cpio formats do not support nanosecond timestamps; old tar formats do not support large device numbers.