NAME

archive_read_new, **archive_read_set_filter_options**, **archive_read_set_format_options**, **archive_read_set_options**, **archive_read_support_compression_all**, **archive_read_support_compression_bzip2**, **archive_read_support_compression_compress**, **archive_read_support_compression_gzip**, **archive_read_support_compression_lzma**, **archive_read_support_compression_none**, **archive_read_support_compression_xz**, **archive_read_support_compression_program**, **archive_read_support_compression_program_signature**, **archive_read_support_format_all**, **archive_read_support_format_ar**, **archive_read_support_format_cpio**, **archive_read_support_format_empty**, **archive_read_support_format_iso9660**, **archive_read_support_format_mtree**, **archive_read_support_format_raw**, **archive_read_support_format_tar**, **archive_read_support_format_zip**, **archive_read_open**, **archive_read_open2**, **archive_read_open_fd**, **archive_read_open_FILE**, **archive_read_open_filename**, **archive_read_open_memory**, **archive_read_next_header**, **archive_read_next_header2**, **archive_read_data**, **archive_read_data_block**, **archive_read_data_skip**, **archive_read_data_into_buffer**, **archive_read_data_into_fd**, **archive_read_extract**, **archive_read_extract2**, **archive_read_extract_set_progress_callback**, **archive_read_close**, **archive_read_finish** — functions for reading streaming archives

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_read_new(void);

int
archive_read_support_compression_all(struct archive *);

int
archive_read_support_compression_bzip2(struct archive *);

int
archive_read_support_compression_compress(struct archive *);

int
archive_read_support_compression_gzip(struct archive *);

int
archive_read_support_compression_lzma(struct archive *);

int
archive_read_support_compression_none(struct archive *);

int
archive_read_support_compression_xz(struct archive *);

int
archive_read_support_compression_program(struct archive *,
    const char *cmd);
```

int
archive_read_support_compression_program_signature(*struct archive *, const char *cmd, const void *signature, size_t signature_length*);

int
archive_read_support_format_all(*struct archive **);

int
archive_read_support_format_ar(*struct archive **);

int
archive_read_support_format_cpio(*struct archive **);

int
archive_read_support_format_empty(*struct archive **);

int
archive_read_support_format_iso9660(*struct archive **);

int
archive_read_support_format_mtree(*struct archive **);

int
archive_read_support_format_raw(*struct archive **);

int
archive_read_support_format_tar(*struct archive **);

int
archive_read_support_format_zip(*struct archive **);

int
archive_read_set_filter_options(*struct archive *, const char **);

int
archive_read_set_format_options(*struct archive *, const char **);

int
archive_read_set_options(*struct archive *, const char **);

int
archive_read_open(*struct archive *, void *client_data, archive_open_callback *, archive_read_callback *, archive_close_callback **);

int
archive_read_open2(*struct archive *, void *client_data, archive_open_callback *, archive_read_callback *, archive_skip_callback *, archive_close_callback **);

int
archive_read_open_FILE(*struct archive *, FILE *file*);

int
archive_read_open_fd(*struct archive *, int fd, size_t block_size*);

int
archive_read_open_filename(*struct archive *, const char *filename, size_t block_size*);

int
archive_read_open_memory(*struct archive **, *void *buff*, *size_t size*);

int
archive_read_next_header(*struct archive **, *struct archive_entry ***);

int
archive_read_next_header2(*struct archive **, *struct archive_entry **);

ssize_t
archive_read_data(*struct archive **, *void *buff*, *size_t len*);

int
archive_read_data_block(*struct archive **,*const void ***buff*,*size_t *len*,
*off_t *offset*);

int
archive_read_data_skip(*struct archive **);

int
archive_read_data_into_buffer(*struct archive **, *void **, *ssize_t len*);

int
archive_read_data_into_fd(*struct archive **, *int fd*);

int
archive_read_extract(*struct archive **,*struct archive_entry **,*int flags*);

int
archive_read_extract2(*struct archive *src*,*struct archive_entry **,
*struct archive *dest*);

void
archive_read_extract_set_progress_callback(*struct archive **,
*void (*func)(void **)*,*void *user_data*);

int
archive_read_close(*struct archive **);

int
archive_read_finish(*struct archive **);

DESCRIPTION

These functions provide a complete API for reading streaming archives. The general process is to first create the struct archive object, set options, initialize the reader, iterate over the archive headers and associated data, then close the archive and release all resources. The following summary describes the functions in approximately the order they would be used:

archive_read_new()
: Allocates and initializes a struct archive object suitable for reading from an archive.

archive_read_support_compression_bzip2(),
archive_read_support_compression_compress(),
archive_read_support_compression_gzip(),
archive_read_support_compression_lzma(),
archive_read_support_compression_none(),
archive_read_support_compression_xz()
: Enables auto-detection code and decompression support for the specified compression. Returns **ARCHIVE_OK** if the compression is fully supported, or **ARCHIVE_WARN** if the compression is supported only through an external program. Note that decompression using an external program

is usually slower than decompression through built-in libraries. Note that "none" is always enabled by default.

archive_read_support_compression_all()
Enables all available decompression filters.

archive_read_support_compression_program()
Data is fed through the specified external program before being dearchived. Note that this disables automatic detection of the compression format, so it makes no sense to specify this in conjunction with any other decompression option.

archive_read_support_compression_program_signature()
This feeds data through the specified external program but only if the initial bytes of the data match the specified signature value.

archive_read_support_format_all(), **archive_read_support_format_ar**(), **archive_read_support_format_cpio**(), **archive_read_support_format_empty**(), **archive_read_support_format_iso9660**(), **archive_read_support_format_mtree**(), **archive_read_support_format_tar**(), **archive_read_support_format_zip**()
Enables support---including auto-detection code---for the specified archive format. For example, **archive_read_support_format_tar**() enables support for a variety of standard tar formats, old-style tar, ustar, pax interchange format, and many common variants. For convenience, **archive_read_support_format_all**() enables support for all available formats. Only empty archives are supported by default.

archive_read_support_format_raw()
The "raw" format handler allows libarchive to be used to read arbitrary data. It treats any data stream as an archive with a single entry. The pathname of this entry is "data"; all other entry fields are unset. This is not enabled by **archive_read_support_format_all**() in order to avoid erroneous handling of damaged archives.

archive_read_set_filter_options(), **archive_read_set_format_options**(), **archive_read_set_options**()
Specifies options that will be passed to currently-registered filters (including decompression filters) and/or format readers. The argument is a comma-separated list of individual options. Individual options have one of the following forms:

option=value
The option/value pair will be provided to every module. Modules that do not accept an option with this name will ignore it.

option The option will be provided to every module with a value of "1".

!option
The option will be provided to every module with a NULL value.

module:option=value, *module:option*, *module:!option*
As above, but the corresponding option and value will be provided only to modules whose name matches *module*.

The return value will be **ARCHIVE_OK** if any module accepts the option, or **ARCHIVE_WARN** if no module accepted the option, or **ARCHIVE_FATAL** if there was a fatal error while attempting to process the option.

The currently supported options are:

Format iso9660

joliet Support Joliet extensions. Defaults to enabled, use **!joliet** to disable.

archive_read_open()
The same as **archive_read_open2**(), except that the skip callback is assumed to be NULL.

archive_read_open2()
: Freeze the settings, open the archive, and prepare for reading entries. This is the most generic version of this call, which accepts four callback functions. Most clients will want to use **archive_read_open_filename**(), **archive_read_open_FILE**(), **archive_read_open_fd**(), or **archive_read_open_memory**() instead. The library invokes the client-provided functions to obtain raw bytes from the archive.

archive_read_open_FILE()
: Like **archive_read_open**(), except that it accepts a *FILE* * pointer. This function should not be used with tape drives or other devices that require strict I/O blocking.

archive_read_open_fd()
: Like **archive_read_open**(), except that it accepts a file descriptor and block size rather than a set of function pointers. Note that the file descriptor will not be automatically closed at end-of-archive. This function is safe for use with tape drives or other blocked devices.

archive_read_open_file()
: This is a deprecated synonym for **archive_read_open_filename**().

archive_read_open_filename()
: Like **archive_read_open**(), except that it accepts a simple filename and a block size. A NULL filename represents standard input. This function is safe for use with tape drives or other blocked devices.

archive_read_open_memory()
: Like **archive_read_open**(), except that it accepts a pointer and size of a block of memory containing the archive data.

archive_read_next_header()
: Read the header for the next entry and return a pointer to a struct archive_entry. This is a convenience wrapper around **archive_read_next_header2**() that reuses an internal struct archive_entry object for each request.

archive_read_next_header2()
: Read the header for the next entry and populate the provided struct archive_entry.

archive_read_data()
: Read data associated with the header just read. Internally, this is a convenience function that calls **archive_read_data_block**() and fills any gaps with nulls so that callers see a single continuous stream of data.

archive_read_data_block()
: Return the next available block of data for this entry. Unlike **archive_read_data**(), the **archive_read_data_block**() function avoids copying data and allows you to correctly handle sparse files, as supported by some archive formats. The library guarantees that offsets will increase and that blocks will not overlap. Note that the blocks returned from this function can be much larger than the block size read from disk, due to compression and internal buffer optimizations.

archive_read_data_skip()
: A convenience function that repeatedly calls **archive_read_data_block**() to skip all of the data for this archive entry.

archive_read_data_into_buffer()
: This function is deprecated and will be removed. Use **archive_read_data**() instead.

archive_read_data_into_fd()
: A convenience function that repeatedly calls **archive_read_data_block**() to copy the entire entry to the provided file descriptor.

archive_read_extract(), **archive_read_extract_set_skip_file**()
: A convenience function that wraps the corresponding archive_write_disk(3) interfaces. The first call to **archive_read_extract**() creates a restore object using archive_write_disk_new(3) and archive_write_disk_set_standard_lookup(3), then transparently invokes

archive_write_disk_set_options(3), archive_write_header(3), archive_write_data(3), and archive_write_finish_entry(3) to create the entry on disk and copy data into it. The *flags* argument is passed unmodified to archive_write_disk_set_options(3).

archive_read_extract2()
: This is another version of **archive_read_extract**() that allows you to provide your own restore object. In particular, this allows you to override the standard lookup functions using archive_write_disk_set_group_lookup(3), and archive_write_disk_set_user_lookup(3). Note that **archive_read_extract2**() does not accept a *flags* argument; you should use **archive_write_disk_set_options**() to set the restore options yourself.

archive_read_extract_set_progress_callback()
: Sets a pointer to a user-defined callback that can be used for updating progress displays during extraction. The progress function will be invoked during the extraction of large regular files. The progress function will be invoked with the pointer provided to this call. Generally, the data pointed to should include a reference to the archive object and the archive_entry object so that various statistics can be retrieved for the progress display.

archive_read_close()
: Complete the archive and invoke the close callback.

archive_read_finish()
: Invokes **archive_read_close**() if it was not invoked manually, then release all resources. Note: In libarchive 1.x, this function was declared to return *void*, which made it impossible to detect certain errors when **archive_read_close**() was invoked implicitly from this function. The declaration is corrected beginning with libarchive 2.0.

Note that the library determines most of the relevant information about the archive by inspection. In particular, it automatically detects gzip(1) or bzip2(1) compression and transparently performs the appropriate decompression. It also automatically detects the archive format.

A complete description of the struct archive and struct archive_entry objects can be found in the overview manual page for libarchive(3).

CLIENT CALLBACKS

The callback functions must match the following prototypes:

typedef ssize_t **archive_read_callback**(*struct archive *, void *client_data, const void **buffer*)

typedef int **archive_skip_callback**(*struct archive *, void *client_data, size_t request*)

typedef int **archive_open_callback**(*struct archive *, void *client_data*)

typedef int **archive_close_callback**(*struct archive *, void *client_data*)

The open callback is invoked by **archive_open**(). It should return **ARCHIVE_OK** if the underlying file or data source is successfully opened. If the open fails, it should call **archive_set_error**() to register an error code and message and return **ARCHIVE_FATAL**.

The read callback is invoked whenever the library requires raw bytes from the archive. The read callback should read data into a buffer, set the const void **buffer argument to point to the available data, and return a count of the number of bytes available. The library will invoke the read callback again only after it has consumed this data. The library imposes no constraints on the size of the data blocks returned. On end-of-file, the read callback should return zero. On error, the read callback should invoke **archive_set_error**() to register an error code and message and return -1.

The skip callback is invoked when the library wants to ignore a block of data. The return value is the number of bytes actually skipped, which may differ from the request. If the callback cannot skip data, it should return zero. If the skip callback is not provided (the function pointer is `NULL`), the library will invoke the read function instead and simply discard the result. A skip callback can provide significant performance gains when reading uncompressed archives from slow disk drives or other media that can skip quickly.

The close callback is invoked by archive_close when the archive processing is complete. The callback should return **`ARCHIVE_OK`** on success. On failure, the callback should invoke **`archive_set_error`**() to register an error code and message and return **`ARCHIVE_FATAL.`**

EXAMPLE

The following illustrates basic usage of the library. In this example, the callback functions are simply wrappers around the standard `open`(2), `read`(2), and `close`(2) system calls.

```
void
list_archive(const char *name)
{
  struct mydata *mydata;
  struct archive *a;
  struct archive_entry *entry;

  mydata = malloc(sizeof(struct mydata));
  a = archive_read_new();
  mydata->name = name;
  archive_read_support_compression_all(a);
  archive_read_support_format_all(a);
  archive_read_open(a, mydata, myopen, myread, myclose);
  while (archive_read_next_header(a, &entry) == ARCHIVE_OK) {
    printf("%s\n",archive_entry_pathname(entry));
    archive_read_data_skip(a);
  }
  archive_read_finish(a);
  free(mydata);
}

ssize_t
myread(struct archive *a, void *client_data, const void **buff)
{
  struct mydata *mydata = client_data;

  *buff = mydata->buff;
  return (read(mydata->fd, mydata->buff, 10240));
}

int
myopen(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  mydata->fd = open(mydata->name, O_RDONLY);
  return (mydata->fd >= 0 ? ARCHIVE_OK : ARCHIVE_FATAL);
}
```

```
int
myclose(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  if (mydata->fd > 0)
    close(mydata->fd);
  return (ARCHIVE_OK);
}
```

RETURN VALUES

Most functions return zero on success, non-zero on error. The possible return codes include: **ARCHIVE_OK** (the operation succeeded), **ARCHIVE_WARN** (the operation succeeded but a non-critical error was encountered), **ARCHIVE_EOF** (end-of-archive was encountered), **ARCHIVE_RETRY** (the operation failed but can be retried), and **ARCHIVE_FATAL** (there was a fatal error; the archive should be closed immediately). Detailed error codes and textual descriptions are available from the **archive_errno**() and **archive_error_string**() functions.

archive_read_new() returns a pointer to a freshly allocated struct archive object. It returns NULL on error.

archive_read_data() returns a count of bytes actually read or zero at the end of the entry. On error, a value of **ARCHIVE_FATAL**, **ARCHIVE_WARN**, or **ARCHIVE_RETRY** is returned and an error code and textual description can be retrieved from the **archive_errno**() and **archive_error_string**() functions.

The library expects the client callbacks to behave similarly. If there is an error, you can use **archive_set_error**() to set an appropriate error code and description, then return one of the non-zero values above. (Note that the value eventually returned to the client may not be the same; many errors that are not critical at the level of basic I/O can prevent the archive from being properly read, thus most I/O errors eventually cause **ARCHIVE_FATAL** to be returned.)

SEE ALSO

tar(1), archive(3), archive_util(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

Many traditional archiver programs treat empty files as valid empty archives. For example, many implementations of tar(1) allow you to append entries to an empty file. Of course, it is impossible to determine the format of an empty file by inspecting the contents, so this library treats empty files as having a special “empty” format.