NAME

archive_write_new, **archive_write_set_format_cpio**, **archive_write_set_format_pax**, **archive_write_set_format_pax_restricted**, **archive_write_set_format_shar**, **archive_write_set_format_shar_binary**, **archive_write_set_format_ustar**, **archive_write_get_bytes_per_block**, **archive_write_set_bytes_per_block**, **archive_write_set_bytes_in_last_block**, **archive_write_set_compression_bzip2**, **archive_write_set_compression_compress**, **archive_write_set_compression_gzip**, **archive_write_set_compression_none**, **archive_write_set_compression_program**, **archive_write_set_compressor_options**, **archive_write_set_format_options**, **archive_write_set_options**, **archive_write_open**, **archive_write_open_fd**, **archive_write_open_FILE**, **archive_write_open_filename**, **archive_write_open_memory**, **archive_write_header**, **archive_write_data**, **archive_write_finish_entry**, **archive_write_close**, **archive_write_finish** — functions for creating archives

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_write_new(void);

int
archive_write_get_bytes_per_block(struct archive *);

int
archive_write_set_bytes_per_block(struct archive *, int bytes_per_block);

int
archive_write_set_bytes_in_last_block(struct archive *, int);

int
archive_write_set_compression_bzip2(struct archive *);

int
archive_write_set_compression_compress(struct archive *);

int
archive_write_set_compression_gzip(struct archive *);

int
archive_write_set_compression_none(struct archive *);

int
archive_write_set_compression_program(struct archive *, const char * cmd);

int
archive_write_set_format_cpio(struct archive *);

int
archive_write_set_format_pax(struct archive *);

int
archive_write_set_format_pax_restricted(struct archive *);

int
archive_write_set_format_shar(struct archive *);
```

int
archive_write_set_format_shar_binary(*struct archive* *);

int
archive_write_set_format_ustar(*struct archive* *);

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

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

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

int
archive_write_open(*struct archive* *,*void* **client_data*,
archive_open_callback *,*archive_write_callback* *,
archive_close_callback *);

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

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

int
archive_write_open_filename(*struct archive* *, *const char* **filename*);

int
archive_write_open_memory(*struct archive* *,*void* **buffer*,
size_t bufferSize,*size_t* **outUsed*);

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

ssize_t
archive_write_data(*struct archive* *, *const void* *, *size_t*);

int
archive_write_finish_entry(*struct archive* *);

int
archive_write_close(*struct archive* *);

int
archive_write_finish(*struct archive* *);

DESCRIPTION

These functions provide a complete API for creating streaming archive files. The general process is to first create the struct archive object, set any desired options, initialize the archive, append entries, then close the archive and release all resources. The following summary describes the functions in approximately the order they are ordinarily used:

archive_write_new()
: Allocates and initializes a struct archive object suitable for writing a tar archive.

archive_write_set_bytes_per_block()
: Sets the block size used for writing the archive data. Every call to the write callback function, except possibly the last one, will use this value for the length. The third parameter is a boolean

that specifies whether or not the final block written will be padded to the full block size. If it is zero, the last block will not be padded. If it is non-zero, padding will be added both before and after compression. The default is to use a block size of 10240 bytes and to pad the last block. Note that a block size of zero will suppress internal blocking and cause writes to be sent directly to the write callback as they occur.

archive_write_get_bytes_per_block()
Retrieve the block size to be used for writing. A value of -1 here indicates that the library should use default values. A value of zero indicates that internal blocking is suppressed.

archive_write_set_bytes_in_last_block()
Sets the block size used for writing the last block. If this value is zero, the last block will be padded to the same size as the other blocks. Otherwise, the final block will be padded to a multiple of this size. In particular, setting it to 1 will cause the final block to not be padded. For compressed output, any padding generated by this option is applied only after the compression. The uncompressed data is always unpadded. The default is to pad the last block to the full block size (note that **archive_write_open_filename**() will set this based on the file type). Unlike the other "set" functions, this function can be called after the archive is opened.

archive_write_get_bytes_in_last_block()
Retrieve the currently-set value for last block size. A value of -1 here indicates that the library should use default values.

archive_write_set_format_cpio(), **archive_write_set_format_pax**(), **archive_write_set_format_pax_restricted**(), **archive_write_set_format_shar**(), **archive_write_set_format_shar_binary**(), **archive_write_set_format_ustar**()
Sets the format that will be used for the archive. The library can write POSIX octet-oriented cpio format archives, POSIX-standard "pax interchange" format archives, traditional "shar" archives, enhanced "binary" shar archives that store a variety of file attributes and handle binary files, and POSIX-standard "ustar" archives. The pax interchange format is a backwards-compatible tar format that adds key/value attributes to each entry and supports arbitrary filenames, linknames, uids, sizes, etc. "Restricted pax interchange format" is the library default; this is the same as pax format, but suppresses the pax extended header for most normal files. In most cases, this will result in ordinary ustar archives.

archive_write_set_compression_bzip2(), **archive_write_set_compression_compress**(), **archive_write_set_compression_gzip**(), **archive_write_set_compression_none**()
The resulting archive will be compressed as specified. Note that the compressed output is always properly blocked.

archive_write_set_compression_program()
The archive will be fed into the specified compression program. The output of that program is blocked and written to the client write callbacks.

archive_write_set_compressor_options(), **archive_write_set_format_options**(), **archive_write_set_options**()
Specifies options that will be passed to the currently-enabled compressor and/or format writer. 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:

Compressor gzip

compression-level
The value is interpreted as a decimal integer specifying the gzip compression level.

Compressor xz

compression-level
The value is interpreted as a decimal integer specifying the compression level.

Format mtree

cksum, **device**, **flags**, **gid**, **gname**, **indent**, **link**, **md5**, **mode**, **nlink**, **rmd160**, **sha1**, **sha256**, **sha384**, **sha512**, **size**, **time**, **uid**, **uname**
Enable a particular keyword in the mtree output. Prefix with an exclamation mark to disable the corresponding keyword. The default is equivalent to "device, flags, gid, gname, link, mode, nlink, size, time, type, uid, uname".

all Enables all of the above keywords.

use-set
Enables generation of **/set** lines that specify default values for the following files and/or directories.

indent XXX needs explanation XXX

archive_write_open()
Freeze the settings, open the archive, and prepare for writing entries. This is the most generic form of this function, which accepts pointers to three callback functions which will be invoked by the compression layer to write the constructed archive.

archive_write_open_fd()
A convenience form of **archive_write_open**() that accepts a file descriptor. The **archive_write_open_fd**() function is safe for use with tape drives or other block-oriented devices.

archive_write_open_FILE()
A convenience form of **archive_write_open**() that accepts a *FILE* * pointer. Note that **archive_write_open_FILE**() is not safe for writing to tape drives or other devices that require correct blocking.

archive_write_open_file()
A deprecated synonym for **archive_write_open_filename**().

archive_write_open_filename()
A convenience form of **archive_write_open**() that accepts a filename. A NULL argument indicates that the output should be written to standard output; an argument of "-" will open a file

with that name. If you have not invoked **archive_write_set_bytes_in_last_block**(), then **archive_write_open_filename**() will adjust the last-block padding depending on the file: it will enable padding when writing to standard output or to a character or block device node, it will disable padding otherwise. You can override this by manually invoking **archive_write_set_bytes_in_last_block**() before calling **archive_write_open**(). The **archive_write_open_filename**() function is safe for use with tape drives or other block-oriented devices.

archive_write_open_memory()
A convenience form of **archive_write_open**() that accepts a pointer to a block of memory that will receive the archive. The final *size_t* * argument points to a variable that will be updated after each write to reflect how much of the buffer is currently in use. You should be careful to ensure that this variable remains allocated until after the archive is closed.

archive_write_header()
Build and write a header using the data in the provided struct archive_entry structure. See archive_entry(3) for information on creating and populating struct archive_entry objects.

archive_write_data()
Write data corresponding to the header just written. Returns number of bytes written or -1 on error.

archive_write_finish_entry()
Close out the entry just written. In particular, this writes out the final padding required by some formats. Ordinarily, clients never need to call this, as it is called automatically by **archive_write_next_header**() and **archive_write_close**() as needed.

archive_write_close()
Complete the archive and invoke the close callback.

archive_write_finish()
Invokes **archive_write_close**() if it was not invoked manually, then releases all resources. Note that this function was declared to return *void* in libarchive 1.x, which made it impossible to detect errors when **archive_write_close**() was invoked implicitly from this function. This is corrected beginning with libarchive 2.0.

More information about the *struct archive* object and the overall design of the library can be found in the libarchive(3) overview.

IMPLEMENTATION

Compression support is built-in to libarchive, which uses zlib and bzlib to handle gzip and bzip2 compression, respectively.

CLIENT CALLBACKS

To use this library, you will need to define and register callback functions that will be invoked to write data to the resulting archive. These functions are registered by calling **archive_write_open**():

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

The open callback is invoked by **archive_write_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**.

typedef ssize_t **archive_write_callback**(*struct archive* *, *void* **client_data*, *const void* **buffer*, *size_t length*)

The write callback is invoked whenever the library needs to write raw bytes to the archive. For correct blocking, each call to the write callback function should translate into a single `write`(2) system call. This is especially critical when writing archives to tape drives. On success, the write callback should return the number of bytes actually written. On error, the callback should invoke **`archive_set_error`**() to register an error code and message and return -1.

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

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

```
#ifdef __linux__
#define _FILE_OFFSET_BITS 64
#endif
#include <sys/stat.h>
#include <archive.h>
#include <archive_entry.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

struct mydata {
       const char *name;
       int fd;
};

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

  mydata->fd = open(mydata->name, O_WRONLY | O_CREAT, 0644);
  if (mydata->fd >= 0)
    return (ARCHIVE_OK);
  else
    return (ARCHIVE_FATAL);
}

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

  return (write(mydata->fd, buff, n));
}

int
```

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

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

void
write_archive(const char *outname, const char **filename)
{
  struct mydata *mydata = malloc(sizeof(struct mydata));
  struct archive *a;
  struct archive_entry *entry;
  struct stat st;
  char buff[8192];
  int len;
  int fd;

  a = archive_write_new();
  mydata->name = outname;
  archive_write_set_compression_gzip(a);
  archive_write_set_format_ustar(a);
  archive_write_open(a, mydata, myopen, mywrite, myclose);
  while (*filename) {
    stat(*filename, &st);
    entry = archive_entry_new();
    archive_entry_copy_stat(entry, &st);
    archive_entry_set_pathname(entry, *filename);
    archive_write_header(a, entry);
    fd = open(*filename, O_RDONLY);
    len = read(fd, buff, sizeof(buff));
    while ( len > 0 ) {
        archive_write_data(a, buff, len);
        len = read(fd, buff, sizeof(buff));
    }
    archive_entry_free(entry);
    filename++;
  }
  archive_write_finish(a);
}

int main(int argc, const char **argv)
{
        const char *outname;
        argv++;
        outname = argv++;
        write_archive(outname, argv);
        return 0;
}
```

RETURN VALUES

Most functions return **ARCHIVE_OK** (zero) on success, or one of several non-zero error codes for errors. Specific error codes include: **ARCHIVE_RETRY** for operations that might succeed if retried, **ARCHIVE_WARN** for unusual conditions that do not prevent further operations, and **ARCHIVE_FATAL** for serious errors that make remaining operations impossible. The **archive_errno**() and **archive_error_string**() functions can be used to retrieve an appropriate error code and a textual error message.

archive_write_new() returns a pointer to a newly-allocated struct archive object.

archive_write_data() returns a count of the number of bytes actually written. On error, -1 is returned and the **archive_errno**() and **archive_error_string**() functions will return appropriate values. Note that if the client-provided write callback function returns a non-zero value, that error will be propagated back to the caller through whatever API function resulted in that call, which may include **archive_write_header**(), **archive_write_data**(), **archive_write_close**(), or **archive_write_finish**(). The client callback can call **archive_set_error**() to provide values that can then be retrieved by **archive_errno**() and **archive_error_string**().

SEE ALSO

tar(1), libarchive(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

There are many peculiar bugs in historic tar implementations that may cause certain programs to reject archives written by this library. For example, several historic implementations calculated header checksums incorrectly and will thus reject valid archives; GNU tar does not fully support pax interchange format; some old tar implementations required specific field terminations.

The default pax interchange format eliminates most of the historic tar limitations and provides a generic key/value attribute facility for vendor-defined extensions. One oversight in POSIX is the failure to provide a standard attribute for large device numbers. This library uses “SCHILY.devminor” and “SCHILY.devmajor” for device numbers that exceed the range supported by the backwards-compatible ustar header. These keys are compatible with Joerg Schilling’s **star** archiver. Other implementations may not recognize these keys and will thus be unable to correctly restore device nodes with large device numbers from archives created by this library.