NAME

csv − CSV parser and writer library

SYNOPSIS

#include <libcsv/csv.h>

int csv_init(struct csv_parser *****p*, unsigned char** *options***);**
size_t csv_parse(struct csv_parser ****p*,**
const char ****s*,**
size_t *len***,**
void (**cb1***)(char *, size_t, void *),**
void (**cb2***)(char, void *),**
void ****data*);**
int csv_fini(struct csv_parser ****p*,**
void (**cb1***)(char *, size_t, void *),**
void (**cb2***)(char, void *),**
void ****data*);**
void csv_free(struct csv_parser ****p*);**

char csv_get_delim(struct csv_parser ****p*);**
char csv_get_quote(struct csv_parser ****p*);**
void csv_set_space_func(struct csv_parser ****p*, int (****f***)(char));**
void csv_set_term_func(struct csv_parser ****p*, int (****f***)(char));**

int csv_opts(struct csv_parser ****p*, unsigned char** *options***);**
int csv_error(struct csv_parser ****p*);**
char * csv_strerror(int *error***);**

size_t csv_write(char ****dest*, size_t** *dest_size***, const char ****src***,**
size_t *src_size***);**
int csv_fwrite(FILE ****fp*, const char ****src***, size_t** *src_size***);**

size_t csv_write2(char ****dest*, size_t** *dest_size***, const char ****src***,**
size_t *src_size***, char** *quote***);**
int csv_fwrite2(FILE ****fp*, const char ****src***, size_t** *src_size***, char** *quote***);**

DESCRIPTION

The CSV library provides a flexible, intuitive interface for parsing and writing csv data.

OVERVIEW

The idea behind parsing with **libcsv** is straight-forward: you initialize a parser object with **csv_init()** and feed data to the parser over one or more calls to **csv_parse()** providing callback functions that handle end-of-field and end-of-row events. **csv_parse()** parses the data provided calling rhe user-defined callback functions as it reads fields and rows. When complete, **csv_fini()** is called to finish processing the current field and make a final call to the callback functions if neccessary. **csv_free()** is then called to free the parser object. **csv_error()** and **csv_strerror()** provide information about errors encountered by the functions. **csv_write()** and **csv_fwrite()** provide a simple interface for converting raw data into CSV data and storing the result into a buffer or file respectively.

CSV is a binary format allowing the storage of arbitrary binary data, files opened for reading or writing CSV data should be opened in binary mode.

libcsv provides a default mode in which the parser will happily process any data as CSV without complaint, this is useful for parsing files which don’t adhere to all the traditional rules. A strict mode is also supported which will cause any violation of the imposed rules to cause a parsing failure.

ROUTINES

csv_init() initializes a pointer to a **csv_parser** structure. This structure contains housekeeping information such as the current state of the parser, the buffer, current size and position, etc. The **csv_init()** function returns 0 on success and a non-zero value upon failure. **csv_init()** will fail if the pointer passed to it is a null pointer or if there is insufficient memory to allocate the structure or its entry buffer. The *options* argument specifies the parser options, these may be changed later with the **csv_opts()** function. The **CSV_STRICT** option enables strict mode, **CSV_REPALL_NL** causes each instance of a carriage return or linefeed outside of a record to be reported. The **CSV_STRICT_FINI** option causes unterminated quoted fields encountered in **csv_fini()** to cause a parsing error (see below). Multiple options can be specified by OR-ing them together.

csv_parse() is the function that does the actual parsing, it takes 6 arguments. *p* is a pointer to an initialized **struct csv_parser**. *s* is a pointer to the data to read in, such as a dynamically allocated region of memory containing data read in from a call to **fread()**. *len* is the number of bytes of data to process, *cb1* is a pointer to the callback function that will be called from **csv_parse()** after an entire field has been read. *cb1* will be called with a pointer to the parsed data (which is NOT nul-terminated), the number of bytes in the data, and the pointer that was passed to **csv_parse()**. *cb2* is a pointer to the callback function that will be called when the end of a record is encountered, it will be called with the character that caused the record to end and the pointer that was passed to **csv_init()**. *data* is a pointer to user-defined data that will be passed to the callback functions when invoked. *cb1* and/or *cb2* may be **NULL** in which case no function will be called for the associated actions. *data* may also be **NULL** but the callback functions must be prepared to handle receiving a null pointer. By default *cb2* is not called when rows that do not contain any fields are encountered. This behavior is meant to accomodate files using only either a linefeed or a carriage return as a record seperator to be parsed properly while at the same time being able to parse files with rows terminated by multiple characters from resulting in blank rows after each actual row of data (for example, processing a text CSV file created that was created on a Windows machine on a Unix machine). The **CSV_REPALL_NL** option will cause **cb2** to be called once for every carraige return or linefeed encountered outside of a field. *cb2* is called with the character that prompted the call to the function, either **CSV_CR** for carriage return, **CSV_LF** for linefeed, or **0** for record termination from a call to **csv_fini()** (see below). A carriage return or linefeed within a non-quoted field always marks both the end of the field and the row.

Note: the first parameter of the *cb1* function is **char ***, not **const char ***; the pointer passed to the callback function is actually a pointer to the entry buffer inside the **csv_parser struct**, this data may safely be modified from the callback function (or any function that the callback function calls) but you must not attempt to access more than *len* bytes and you should not access the data after the callback function returns as the buffer is dynamically allocated and its location and size may change during calls to **csv_parse()**.

Note: Different callback functions may safely be specified during each call to **csv_parse()** but keep in mind that the callback functions may be called many times during a single call to **csv_parse()** depending on the amount of data being processed in a given call.

csv_parse() returns the number of bytes processed, on a successful call this will be *len*, if it is less then an error has occured. An error can occur, for example, if there is insufficient memory to store the contents of the current field in the entry buffer. An error can also occur if malformed data is encountered while running in strict mode.

The **csv_error()** function can be used to determine what the error is and the **csv_strerror()** function can be used to provide a textual description of the error. **csv_error()** takes a single argument, a pointer to a **struct csv_parser**, and returns one of the following values defined in **csv.h**:

CSV_EPARSE A parse error has occured while in strict mode

CSV_ENOMEM There was not enough memory while attempting to increase the entry buffer for the current field

CSV_ETOOBIG Continuing to process the current field would require a buffer of more than SIZE_MAX bytes

The value passed to **csv_strerror()** should be one returned from **csv_error()**. The return value of **csv_strerror()** is a pointer to a static string. The pointer may be used for the entire lifetime of the program and the contents will not change during execution but you must not attempt to modify the string it points to.

When you have finished submitting data to **csv_parse()**, you need to call the **csv_fini()** function. This function will call the *cb1* function with any remaining data in the entry buffer (if there is any) and call the *cb2* function unless we are already at the end of a row (the last byte processed was a newline character for example). It is neccessary to call this function because the file being processed might not end with a carriage return or newline but the data that has been read in to this point still needs to be submitted to the callback routines. If *cb2* is called from within **csv_fini()** it will be because the row was not terminated with a newline sequence, in this case *cb2* will be called with an argument of 0.

Note: A call to **csv_fini** implicitly ends the field current field and row. If the last field processed is a quoted field that ends before a closing quote is encountered, no error will be reported by default, even if CSV_STRICT is specified. To cause **csv_fini()** to report an error in such a case, set the CSV_STRICT_FINI option (new in version 1.0.1) in addition to the CSV_STRICT option.

csv_fini() also reinitializes the parser state so that it is ready to be used on the next file or set of data. **csv_fini()** does not alter the current buffer size. If the last set of data that was being parsed contained a very large field that increased the size of the buffer, and you need to free that memory before continuing, you must call **csv_free()** and then **csv_init()**. Like csv_parse, the callback functions provided to **csv_fini()** may be NULL. **csv_fini()** returns 0 on success and a non-zero value if you pass it a null pointer.

After calling **csv_fini()** you may continue to use the same struct csv_parser pointer without reinitializing it (in fact you must not call **csv_init()** with an initialized csv_parser object or the memory allocated for the original structure will be lost).

When you are finished using the csv_parser object you can free it along with any dynamically allocated memory associated with it by calling **csv_free()**. You may call **csv_free()** at any time, it need not be preceded by a call to **csv_fini()**. You must only call **csv_free()** with a value assigned to *p* from a successful call to **csv_init()**.

libcsv provides two functions to transform raw data into CSV formatted data: the **csv_write()** function which writes the result to a provided buffer, and the **csv_fwrite()** function which writes the result to a file. The functionality of both functions is straight-forward, they write out a single field including the opening and closing quotes and escape each encountered quote with another quote.

The **csv_write()** function takes a pointer to a source buffer (*src*) and processes at most *src_size* characters from *src*. **csv_write()** will write at most *dest_size* characters to *dest* and returns the number of characters that would have been written if *dest* was large enough. This can be used to determine if all the characters were written and, if not, how large *dest* needs to be to write out all of the data. **csv_write()** may be called with a null pointer for the *dest* argument in which case no data is written but the size required to write out the data will be returned. The space needed to write out the data is the size of the data + number of quotes appearing in data (each one will be escaped) + 2 (the leading and terminating quotes). *csv_write()* and *csv_fwrite()* always surround the output data with quotes. If *src_size* is very large (SIZE_MAX/2 or greater) it is possible that the number of bytes needed to represent the data, after inserting escaping quotes, will be greater than SIZE_MAX. In such a case, csv_write will return SIZE_MAX which should be interpreted as meaning the data is too large to write to a single field. The **csv_fwrite()** function is not similiarly limited.

csv_fwrite() takes a FILE pointer (which should have been opened in binary mode) and converts and writes the data pointed to by *src* of size *src_size*. It returns **0** on success and **EOF** if there was an error writing to the file. **csv_fwrite()** doesn't provide the number of characters processed or written. If this functionality is required, use the **csv_write()** function combined with **fwrite()**.

csv_write2() and **csv_fwrite2()** work similiarly but take an additional argument, the quote character to use when composing the field.

The **csv_set_delim()** and **csv_set_quote()** functions provide a means to change the characters that the parser will consider the delimiter and quote characters respetively. **csv_get_delim()** and **csv_get_delim()** return the current delimiter and quote characters respectively. When **csv_init()** is called the delimiter is set to **CSV_COMMA** and the quote to **CSV_QUOTE**. Note that the rest of the CSV conventions still apply when these functions are used to change the delimiter and/or quote characters, fields containing the new quote character or delimiter must be quoted and quote characters must be escaped with an immediately preceeding instance of the same character. Additionally, the **csv_set_space_func()** and **csv_set_term_func()** allow a user-defined function to be provided which will be used determine what constitutes a space character and what constitutes a record terminator character. The space characters determine which characters are removed from the beginning and end of non-quoted fields and the terminator characters govern when a record ends. When **csv_init()** is called, the effect is as if these functions were each called with a NULL argument in which case no function is called and CSV_SPACE and CSV_TAB are used for space characters, and CSV_CR and CSV_LF are used for terminator characters.

THE CSV FORMAT

Although quite prevelant there is no standard for the CSV format. There are however, a set of traditional conventions used by many applications. **libcsv** follows the conventions described at http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm which seem to reflect the most common usage of the format, namely:

Fields are seperated with commas.

Rows are delimited by newline sequences (see below).

Fields may be surrounded with quotes.

Fields that contains a comma, quote, and newline characters MUST be quoted.

Each instance of a quote character must be escaped with an immediately preceding quote character.

Leading and trailing spaces and tabs are removed from non-quoted fields.

The final line need not contain a newline sequence.

In strict mode, any detectable violation of these rules results in an error.

RFC 4180 is an informational memo which attempts to document the CSV format, especially with regards to its use as a MIME type. There are a several parts of the description documented in this memo which either do not accurately reflect widely used conventions or artificially limit the usefulness of the format. The differences between the RFC and **libcsv** are:

"Each line should contain the same number of fields throughout the file"
: **libcsv** doesn't care if every record contains a different number of fields, such a restriction could easily be enforced by the application itself if desired.

"Spaces are considered part of a field and should not be ignored"
: Leading and trailing spaces that are part of non-quoted fields are ignored as this is by far the most common behavior and expected by many applications.

abc , def

is considered equivalent to:

"abc", "def"

"The last field in the record must not be followed by a comma"

The meaning of this statement is not clear but if the last character of a record is a comma, **libcsv** will interpret that as a final empty field, i.e.:

"abc", "def",

will be interpreted as 3 fields, equivalent to:

"abc", "def", ""

RFC 4180 limits the allowable characters in a CSV field, **libcsv** allows any character to be present in a field provided it adheres to the conventions mentioned above. This makes it possible to store binary data in CSV format, an attribute that many application rely on.

RFC 4180 states that a Carriage Return plus Linefeed combination is used to delimit records, **libcsv** allows any combination of Carriage Returns and Linefeeds to signify the end of a record. This is to increase portability among systems that use different combinations to denote a newline sequence.

PARSING MALFORMED DATA

libcsv should correctly parse any CSV data that conforms to the rules discussed above. By default, however, **libcsv** will also attempt to parse malformed CSV data such as data containing unescaped quotes or quotes within non-quoted fields. For example:

a"c, "d"f"

would be parsed equivalently to the correct form:

"a""c", "d""f"

This is often desirable as there are some applications that do not adhere to the specifications previously discussed. However, there are instances where malformed CSV data is ambigious, namely when a comma or newline is the next non-space character following a quote such as:

"Sally said "Hello", Wally said "Goodbye""

This could either be parsed as a single field containing the data:

Sally said "Hello", Wally said "Goodbye"

or as 2 seperate fields:

Sally said "Hello and **Wally said "Goodbye""**

Since the data is malformed, there is no way to know if the quote before the comma is meant to be a literal quote or if it signifies the end of the field. This is of course not an issue for properly formed data as all quotes are be escaped. **libcsv** will parse this example as 2 seperate fields.

libcsv provides a strict mode that will return with a parse error if a quote is seen inside a non-quoted field or if a non-escaped quote is seen whose next non-space character isn't a comma or newline sequence.

PARSER DETAILS

A field is considered quoted if the first non-space character for a new field is a quote.

If a quote is encountered in a quoted field and the next non-space character is a comma, the field ends at the

closed quote and the field data is submitted when the comma is encountered. If the next non-space character after a quote is a newline character, the row has ended and the field data is submitted and the end of row is signalled (via the appropriate callback function). If two quotes are immediately adjacent, the first one is interpreted as escaping the second one and one quote is written to the field buffer. If the next non-space character following a quote is anything else, the quote is interpreted as a non-escaped literal quote and it and what follows are written to the field buffer, this would cause a parse error in strict mode.

Example 1
"abc"""
Parses as: **abc"**
The first quote marks the field as quoted, the second quote escapes the following quote and the last quote ends the field. This is valid in both strict and non-strict modes.

Example 2
"ab"c
Parses as: **ab"c**
The first qute marks the field as quoted, the second quote is taken as a literal quote since the next non-space character is not a comma, or newline and the quote is not escaped. The last quote ends the field (assuming there is a newline character following). A parse error would result upon seeing the character c in strict mode.

Example 3
"abc" "
Parses as: **abc"**
In this case, since the next non-space character following the second quote is not a comma or newline character, a literal quote is written, the space character after is part of the field, and the last quote terminated the field. This demonstrates the fact that a quote must immediately precede another quote to escape it. This would be a strict-mode violation as all quotes are required to be escaped.

If the field is not quoted, any quote character is taken as part of the field data, any comma terminated the field, and any newline character terminated the field and the record.

Example 4
ab""c
Parses as: **ab""c**
Quotes are not considered special in non-quoted fields. This would be a strict mode violation since quotes may not exist in non-quoted fields in strict mode.

EXAMPLES

The following example prints the number of fields and rows in a file. This is a simplified version of the csvinfo program provided in the examples directory. Error checking not related to **libcsv** has been removed for clarity, the csvinfo program also provides an option for enabling strict mode and handles multiple files.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include "libcsv/csv.h"

struct counts {
  long unsigned fields;
  long unsigned rows;
};

void cb1 (char *s, size_t len, void *data) {
```

```
  ((struct counts *)data)->fields++; }
void cb2 (char c, void *data) {
  ((struct counts *)data)->rows++; }

int main (int argc, char *argv[]) {
  FILE *fp;
  struct csv_parser *p;
  char buf[1024];
  size_t bytes_read;
  struct counts c = {0, 0};

  if (csv_init(&p, 0) != 0) exit(EXIT_FAILURE);
  fp = fopen(argv[1], "rb");
  if (!fp) exit(EXIT_FAILURE);

  while ((bytes_read=fread(buf, 1, 1024, fp)) > 0)
    if (csv_parse(p, buf, bytes_read, cb1, cb2, &c) != bytes_read) {
      fprintf(stderr, "Error while parsing file: %s\n",
      csv_strerror(csv_error(p)) );
      exit(EXIT_FAILURE);
    }

  csv_fini(p, cb1, cb2, &c);

  fclose(fp);
  printf("%lu fields, %lu rows\n", c.fields, c.rows);

  csv_free(p);
  exit(EXIT_SUCCESS);
}
```

See the examples directory for several complete example programs.

AUTHOR

Written by Robert Gamble.

BUGS

Please send questions, comments, bugs, etc. to:

rgamble@sourceforge.net