The CSV format

by Paul Hsieh

Introduction

This specification is based on partial specification by Robert J. Lynch and description by Creativyst Software. This specification explands on these and makes an attempt to very specifically and completely nail down the format in a way that is compatible with existing practice for CSV file generation. This description should provide a way to make CSV a deterministically portable format.

What is the CSV file format?

The Comma Seperated Values, or CSV, format is an encoding format for a finite two dimensional collection of raw data (originally developed by Microsoft for use with its Excel product). The individual datum encodings are any arbitrary contents that can be encoded into a sequence of octets (this includes any arbitrary binary data or things like Unicode's UTF-8, or in fact, CSV encodings themselves). The CSV format encodes a sequence of Fields each of which is a sequence of Entries. CSV imposes no limits on Entry length, Field length or Field count.

Generally, the format is a merely a sequence of octets with the comma (,) quote (") and carriage return (\r) characters taking special meaning that provides the underlying structure of the data. All octets appearing between a pair of bracketting quotes have no special meaning besides being raw data except for quotes themselves which should be doubled up to guarantee correct parsing. Otherwise the comma octet seperates Entries in a Field, and the carriage return octet seperates Fields. Each entry is either a quoted or unquoted octet data sequence -- surrounding whitespace is considered irrelevant and trimmed upon consumption.

Example 1:

A CSV file with the following contents:

```
1234,The Big Ol' Bear
```

Would be parsed as a single field with two data entries, with the contents "1234" and "The Big Ol' Bear". Note that the CSV file format does not dictate any meta-information, so the fact that the first field seems to be a 4-digit number is not something that the CSV file format implicitely encodes.

The next example:

```
"1234 Harrington St, Northwest",Suite 17 Stop 3
```

Shows an example of a quoted entry. The quotes are required because of the embedded comma (,) in the first entry.

In the following example:

```
"1234 West ""Q"" St.",   0
```

The first entry (1234 West "Q" St.) contains quote characters that must be doubled to be rendered properly. The second entry (0) has a leading white space which is ignored.

The next example:

```
"1234 West "Q" St.", 0
```

is an example of legacy formed CSV data resulting from a bug the program "Paradox". A sufficiently forgiving parser should inferr that the quote characters before and after Q character are octet data in sequence with the rest of the octet data for the first entry rather than a quoting escape sequence. This makes this CSV data identical to the example before it. The way that the CSV parser is able to deduce that these quotes are interior data is that they are not followed by an optionally whitespace prefexed comma, carriage return or EOF. These improper encodings are strongly discouraged since they don't provide ways of encoding an entry which contains a quote octet followed by either a comma or carriage return octet.

The next example:

```
"Thomas Aquinus, Esq.
Prosecutor for the Pope
St. Luke's Dungeon
Somewhere in Italy"
```

Shows a single entry which contains carriage return octets.

The following example:

```
    ,Thos.,"",Aquinus,Esq,Pros.forPope,,"Somewhere..."
```

shows that an empty entry may be expressed either as 0-octets between
delimiters, or an empty quote pair.

Grammar
=======

Note that the CSV file format is described in terms of octets which embeds
7-bit ASCII as subset of values expressable by each octet. Such octets will
generally referred to as ascii-8 format. So CSV does not have a direct
encoding in incompatible formats such as EBCDIC, UTF-16 or Base-64. The
correctly parsable CSV grammar is described as follows:

```
  WSChar        = [ \v\f\t]
  NoWSChar      = <ascii-8 - [,"\r\n \v\f\t]>
  RawChar       = <ascii-8 - [,"\r\n]>
  NoQuoteChar   = <ascii-8 - ["]>
  ParadoxBug    = ["] WSChar* NoWSChar NoQuoteChar*
  QuotableText  = NoQuoteChar* (["] ["] NoQuoteChar* | ParadoxBug)*
  NonBlankEntry = NoWSChar (RawChar* NoWSChar)?        # Not bracketted by WS
                | ["] QuotableText ["]
  Entry         = []
                | NonBlankEntry
  Field         = WSChar*                              # 0 entries
                | WSChar* NonBlankEntry WSChar*        # 1 entry
                | WSChar* Entry WSChar* [,] Field      # 2 or more entries
  CSVFile       = Field
                | Field [\r] [\n]? CSVFile
```

Legend:

- []   Represents the empty string.
- ascii-8 Represents one arbitrary octet.
- [...] Represents one character selected from the sequence of characters
           between the brackets. Non-text characters are described in a C
           language character compatible way. The [character is represented
           as \[.
- <A - B> Represents one octet from the set A that is not in the set B.
- A B   Represents the concatenation of rule A with the rule B.
- A*   Represents the rule A concatenated with itself 0 or more times.
- A?   Represents either the rule A or [].
- A | B   Represents either the rule A or the rule B.
- (A)   Represents the rule A taken as a whole.

Words which start with a capital letter are non-terminal rules.

Notes:

- To accomodate for improperly formated quoted entries (such as is output by
  Borland's Paradox), any enclosed single quote which is not followed by any
  amount of white space then either a "," or carriage-return should be
  considered a stand-alone double quote. Going forward, implementations
  should not output CSV data using the ParadoxBug rule.

- An input line which is just white space is 0 entries, not one single
  blank entry. A single blank entry must be explicitely quoted.

- The LF (\n) character is ignored as a Field seperator (i.e., its considered
  white space) but NOT ignored as data within a quoted entry. It is not legal
  to include this character as non-quoted data.

Carriage Return versus Line Feed

In some systems, CSV file data may be read in text mode (this is a data losing
operation) in which case carriage returns (\r) are never read but instead
removed and replaced by linefeed (\n) if one is not already immediately
following. In this case, the grammar would have to be modified in the
CSVFile production rule:

```
  CSVFile       = Field
                | Field [\n] CSVFile
```

Note that this does not solve all CSV structure problems when dealing with
moving CSV files from platform to platform which use different conventions
for text file delimiters (note that Windows/DOS, Unix and Mac OS are all
different.) In particular in some systems a carriage return followed by a
linefeed may be interpreted as two linefeeds.

Because of these anomolies and because it is inherently data losing, it is

highly recommended that where possible CSV data not be consumed as files opened in text mode, but only as binary data.

Discussion

Implementations

An incremental CSV parser can be written which is nothing more than a state machine parser. (This means that it is possible to encode the CSV grammar into a regular expression.) So it is not necessary to perform backwards or bidirectional parsing, nor is an arbitrary amount of storage required to perform the parsing operations. This also means that CSV file parsing can be done with very high performance.

Many naive implementations assume that Fields correspond exactly to lines parsed from the CSV file. This is not the case because of the posibility of quoting the CR octet. Trying to read CSV file in a line by line manner will generally not be an effective way of attempting to parse CSV files.

The C/C++ programming language is a notorious cess pool of "fixed array length" kind of solutions. Programmers should not by trapped by such ideas in an attempt to parse generic CSV files. There are some well known libraries than can be used in various ways to assist in the parsing of CSV files that escape the fixed array length problem for the C programming language. These include:

    The Better String Library - http://bstring.sf.net/
    SGLIB   - http://www.xref-tech.com/sglib/main.html

C++ programmers should familliarize themselves with the Standard Template Library; in particular the std::string class and std::vector<> template, rather than using fixed length arrays.

The binary data in each entry needs to be properly "unpacked" from its CSV encoding (to compress the double quotes to single quotes.) This binary data is also clearly location independent and thus should be intrinsically pre-serialized. This means that CSV can contain data in any arbitrary format so long as it can be encoded in a sequence of octets. This includes Unicode UTF-16 mode data. The problem is that octet encoded UTF-16 may introduce extra comma or quote octets even if they do not represent those characters in the actual text. If, instead, UTF-8 encoding is used, then the appearance of comma or quote octets in the encoding is exactly one to one with their appearance as actual code points in the unicode text itself. Thus UTF-8 encoding has the advantage that quote usage detection and pre-doubling can be performed either at the UTF-8 level or the CSV encoding level, whereas with UTF-16 this can only be done at the CSV encoding level. UTF-8 enabled text editors can also be used on CSV files directly.

The original implementor of CSV output files is, of course, Microsoft. However, Microsoft has chosen not to openly document the format, thus this specification has been largely deduced. There is some confusion about how Microsoft Excel's behavior is related to the CSV format. In particular, Excel will remove white space and leading 0s from numeric entries. In this specification, the position taken is that this is not an inherent attribute of the CSV format, but rather just how Excel choses to deal with data after it reads it from CSV files.

Some CSV file users insist that CSV files have a uniform column structure dictated by the first Field. Like Microsoft Excel's interpretation, such layout constraints are entirely application specific and not dictated by the CSV file format itself.

Possible Future Growth of the CSV standard

CSV is a very old historical standard. There have not been any known efforts to clearly extend the capabilities of the CSV format. The syntax for CSV is a nearly complete format, however one clear opportunity for expansion lies in unquoted entries which contain quote characters. For example if an entry were encoded as something like:

    ext"Extension-Name:extension-data

This would normally simply fall outside of the CSV grammar. But detecting it still falls into the realm of a state machine implementation. So such a strategy may represent a way to extend the standard. This specification does not propose any extensions, nor does it take a position on the possible use of this or other mechanisms for extending the CSV format beyond the grammar described.

Advantages

The format is fairly straight forward to parse (compared to XML), and is

universally portable to just about any system. The CSV format can encode recursively (a single data entry may itself be a quoted CSV encoding), meaning that it can be used to encode arbitrarily convoluted structures. The CSV format can encode both ASCII and UTF-8 directly, meaning that it can be read and manipulated directly with any sufficiently enabled text editor (some text editors may unify CR and LF characters that can result in lost data). No schema or template is assumed which allows it to be freeform, and thus general enough to map to any deterministic static data format. CSV files and their parsers have very low overhead versus other more general formats such as XML.

Disadvantages

Because it is freeform, the data format essentially only supports sequential processing (as opposed to random access). Historically, the CSV file has been output by programs that deviated from the specification described above making parsing them non-deterministic. The format does not contain any provisions for annotation, or meta-data (compare to XML). The CSV file format has been created by Microsoft, and thus is technically proprietary despite the existence of this specification -- this can represent a real problem if, in the future, Microsoft chooses to extend the format in ways that are not easy to deduce.