1 Overview

Text Assembler, abreviated to TA or TXTASM, is a general purpose text/macro processor which takes a text file(s) as input and assembles them into an output text file(s). It does this by parsing the text and writing it into an output stream buffer. This buffer can then be saved to disk.

Rather than implement a custom language for any text generation logic, the open source Google V8 JavaScript Engine [1] has been integrated into the assembler. This allows full use of the ECMAScript® Language as defined in the ECMA-262 standard [2].

Standard JavaScript string variables can be defined and used to perform simple macro replacements. For more complex text generation tasks, a JSON data model can be loaded and used in conjunction with JavaScript code to generate the desired output.

The text assembler was created to generate HTML web pages, but it can also be used to generate any other kind of text file, for example, CSV [3] or even JavaScript files.

1.1 Invocation

The text assembler is invoked on the command line as follows:-

```
ta [options] -o <directory> <source-files>
```

By default, the output filename is the same as the input filename, so the output directory must be specified and should not be the same as the input file directory unless the file is saved with a different name. The following case-sensitive command line options are available.

Name	**Description**
-D name[=value]	Define a JavaScript global variable with optional value. If no value is specified, it is assumed to be assigned the boolean value 'true'. String variable values should be surrounded in single quotes, e.g. -D url='http://www.akiwi.co.uk'
-I <dir>	Add a directory to the include path.
-e <encoding>	Input source file character encoding, either 'UTF8' or 'WIN1252'. The default is 'UTF8'.
-o <directory>	Set the output directory.
-oc	Enable output compaction. Any runs of whitespace are reduced down to a single space.
-od	Enable output differencing. If a file of the same name already exists, the file contents are read into a buffer and compared with the contents of the output stream buffer. If they are identical, the save is skipped preserving the existing file and its timestamp.
-w	Enable warning messages.
-debug	Enable debug messages.
-help	Print a summary of all available command line options.
-version	Show version and copyright information.

2 Assembler Directives

A small number of assembler directives can be used within a document. All assembler directives begin with a # character. To use a literal # within a document, escape it with ## or turn parsing off then on again.

Name	Description

#off	Disables assembler parsing. Sometimes it's necessary to disable the parser if literal text would be misinterpreted.
	
	The #off and any trailing whitespace on the same line are not emitted by the parser. Any leading characters will be emitted. This directive is typically used on a line by itself, e.g.
	
	`#off`
	`Some text not to be parsed by the assembler`
	`#on`

#on	Normal assembler parsing is resumed. The #on and any trailing whitespace on the same line are not emitted by the parser. Any leading characters will be emitted.

3 Assembler Variables

Built-in assembler variables always begin with an underscore '_'. These should be considered as reserved words and not used for any other purpose.

Name	Description
_outdir	A string containing the output directory that was specified on the command line using the -o option.

4 JavaScript Functions

4.1 Core Functions

Here is a table summarising the most useful core functions available in ECMA-262 [2] for the purpose of generating text files. For full documentation, please refer to [2].

Name	**Description**
Number.toString([radix])	Returns a string representing this number.
Number.toFixed(fractionDigits)	Returns a string representing this number with the specfied number of decimal places.
String.indexOf(searchString, position)	Finds a substring within a string.
String.lastIndexOf(searchString, position)	Finds a substring within a string, backwards.
String.replace(searchValue, replaceValue)	Search and replace within a string.
String.search(regexp)	Search for a substring using a regular expression.
String.split(separator, limit)	Splits and array at each separator and returns an array of substrings excluding the separator.
String.substring(start, end)	Extracts a substring from a string.
String.toLowerCase()	Converts a string to lower case in place.
String.toUpperCase()	Converts a string to upper case in place.
String.trim()	Removes leading and trailing whitespace from a string in place.
Date.toUTCString()	Returns a string representation of this date and time in UTC format.
Date.toISOString()	Returns a string representation of this date and time in ISO format.
Date.toDateString()	Returns a string representation of the date part only.
Date.toTimeString()	Returns a string representation of the time part only.
JSON.parse(text [, reviver])	Parses a JSON formatted string and produces a JavaScript value.

4.2 Extensions

These JavaScript functions are available in TA and extend the standard set of JavaScript core functions.

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>$(s)</td><td>The inline string function - a special function. Unlike the other functions which must be called within a JavaScript block, this function actually creates its own JavaScript block implicitly and therefore can occur at any location within a text document.

It enters a JavaScript block, evaluates the string expression 's' and prints the result to the output stream. e.g.

<code><meta content="$(tohtml('food ' + '&' + ' drink');)"/></code>

would be written out as

<code><meta content="food & drink"/></code></td></tr>
<tr><td>erase(n)</td><td>Erases the last 'n' characters from the output stream buffer.</td></tr>
<tr><td>lastmod(filename)</td><td>To be implemented soon...
Returns a JavaScript Date object containing the time at which the specified file was last modified.

This function can be used to create XML sitemaps with accurate lastmod elements.</td></tr>
<tr><td>load(filename)</td><td>Loads a UTF-8 text file into memory, converts it to UTF-16 and then stores the result in a JavaScript string variable.

This function can be used to load external strings and data for use during text assembly.

<pre>/* Load some JSON data */
var s = load('data.json');
var d = JSON.parse(s);</pre></td></tr>
<tr><td>parse(filename)</td><td>Include a text file at the current location and parse it. If the file cannot be found, then the -I path list is searched for the named file.</td></tr>
<tr><td>print(s)</td><td>Write a string to stdout. This can be used to generate status messages during text processing operations.</td></tr>
<tr><td>save(filename [, flags])</td><td>Save the output stream buffer to a named file relative to the output directory. After the save, the buffer length is set to zero.

This functions behaviour is affected by the command line options -oc and -od which enable output compaction and differencing respectively.

If a flags string is specified, then it overrides the global settings. To enable or disable an option, the flag should be preceded by a '+' or '-' character respectively. The following flags are currently available.

<table><tr><td>c</td><td>Output compaction</td></tr></table></td></tr>
</table>

	<table><tr><td>d</td><td>Output differencing</td></tr></table> Returns true if the file was written out to disk successfully, otherwise false.

tocsv(s [, separator])	Return the string 's' converted to a CSV field, properly double quoted and escaped if necessary. The string is only enclosed in double quotes if it contains a separator, double quote or newline character(s).
	
	The integer separator character value is optional and defaults to a comma = 44.
	
	`/* Write out a tsv field */`
	`write(tocsv(s,9)+'\t');`

tohtml(s)	Return a string in HTML format, replacing <, >, &, ', " characters with their corresponding entity references <, >, &, ', ". If an & character starts a recognised HTML entity reference, then no replacement is made. Any characters contained within a CDATA section delimited by '<![CDATA[' and ']]>' are output verbatim. The CDATA start and end sequences are removed from the string.
topath(s)	Return a valid path name, replacing any illegal path name characters within 's' with an underscore. Runs of underscores are reduced to a single occcurance.
toplain(s)	Return 's' as plain text without any HTML character entity references.
write(s)	Append a string to the output stream buffer.

5 Text Generation

There are various mechanisms available to generate output text. These are discussed below in the relevant section.

5.1 Script Blocks

Text assembler scripts within a document must be enclosed in a script block with type "application/vnd.akiwi.ta". The inline string function $() is the only function that may be called outside a script block.

```
<script type="application/vnd.akiwi.ta">
   /* Put your JavaScript here */
</script>
```

5.2 The parse() Function

The parse() function can be used to concatenate text files. Each file is loaded and parsed sequentially. If there are no text assembler directives or scripts contained within the file, then the input file is written to the output stream verbatim.

For example, if we have 3 plain HTML files, header.html, body.html and footer.html, these can be concatenated and output as a single document using the following script.

File: myfile.html

```
<script type="application/vnd.akiwi.ta">
parse('header.html');
parse('body.html');
parse('footer.html');
</script>
```

5.3 The write() Function

The write() function is used to write a string to the output stream.

File: myfile.html

```
<script type="application/vnd.akiwi.ta">
   write('Hello World\n');
</script>
```

5.4 Conditional Text Generation

A standard JavaScript if statement can be used to conditionally generate text. For example, to simulate the C preprocessors #ifdef directive, we can use the JavaScript typeof operator as follows.

```
<script type="application/vnd.akiwi.ta">
if( typeof myvar === 'undefined' )
   write('a');
else
   write('b');
</script>
```

Assuming the string variable myvar exists and can take on several values, then we can also use the following script to conditionally generate text.

```
<script type="application/vnd.akiwi.ta">
if( myvar === 'foo' )
```

```
   write('a');
else if( myvar === 'bar' )
   write('b');
else
   write('c');
</script>
```

5.5 Iterative Text Generation

Standard JavaScript loop statements can be used to iteratively generate text, e.g.

```
<script type="application/vnd.akiwi.ta">
var a = new Array("Mon","Tue","Wed","Thu","Fri","Sat","Sun");
for( var n=0; n<a.length; n++ )
    write('<th>'+a[n]+'</th>\n');
</script>
```

6 References

[1]	V8 JavaScript Engine. Google Inc. http://code.google.com/p/v8/
[2]	Standard ECMA-262. ECMAScript Language Specification. Edition 5.1 (June 2011) http://www.ecma-international.org/publications/standards/Ecma-262.htm
[3]	RFC-4180. Common Format and MIME Type for Comma-Separated Values (CSV) Files. The Internet Society. 2005. http://tools.ietf.org/html/rfc4180
	Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation 16 August 2006, edited in place 29 September 2006. http://www.w3.org/TR/xml11/

7 Disclaimer

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.