Table of Contents

SheetJS js-xlsx

Parser and writer for various spreadsheet formats. Pure-JS cleanroom implementation from official specifications, related documents, and test files. Emphasis on parsing and writing robustness, cross-format feature compatibility with a unified JS representation, and ES3/ES5 browser compatibility back to IE6.

This is the community version. We also offer a pro version with performance enhancements and additional features by request.

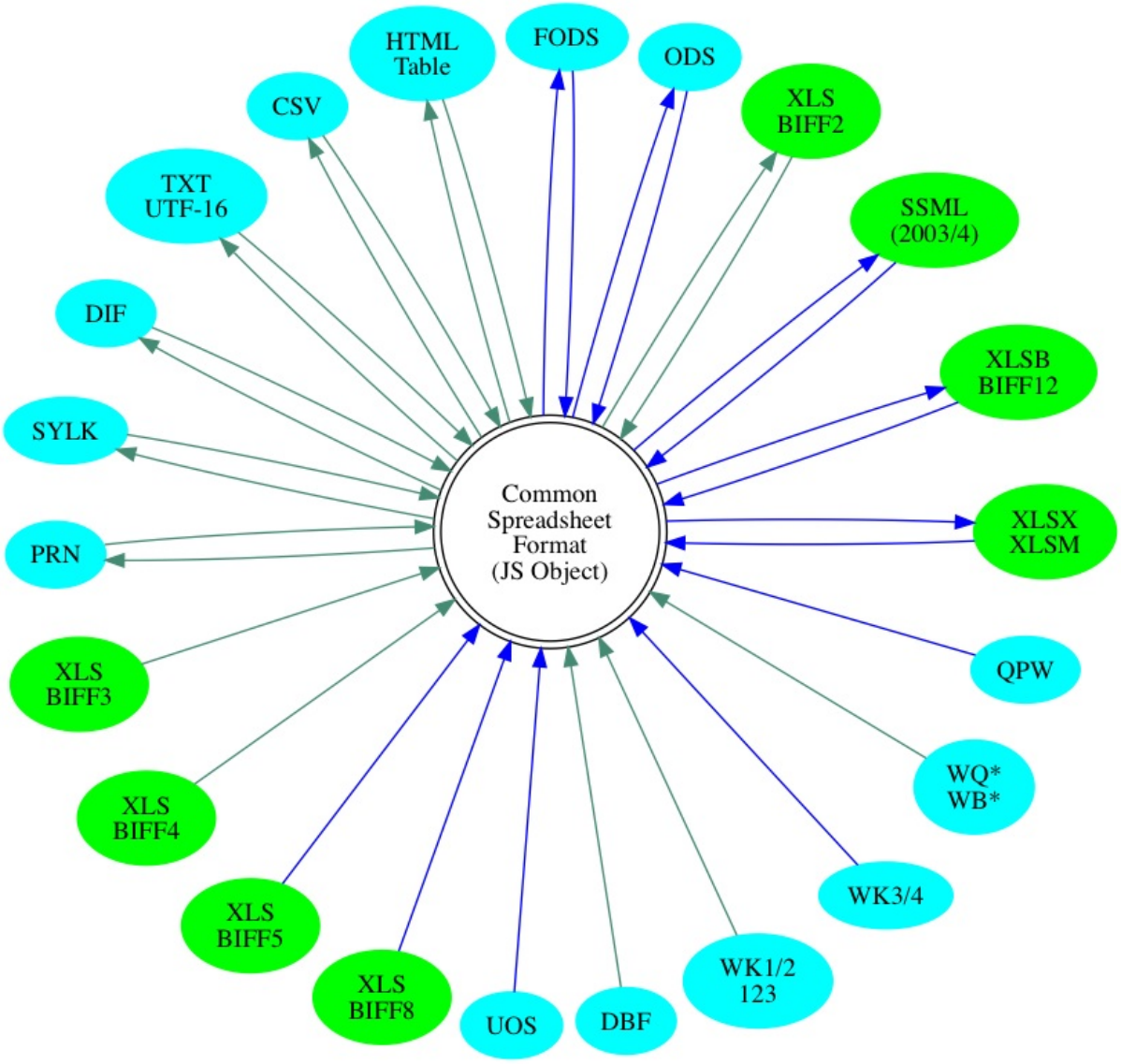
Pro Version

Commercial Support

In-Browser Demos

Source Code

File format support for known spreadsheet data formats:


HTML
Table
FODS
ODS
CSV
XLS
BIFF2
TXT
UTF-16
SSML
(2003/4)
DIF
XLSB
BIFF12
SYLK
Common
Spreadsheet
Format
(JS Object)
XLSX
XLSM
PRN
QPW
XLS
BIFF3
WQ*
WB*
XLS
BIFF4
WK3/4
XLS
BIFF5
WK1/2
123
XLS
BIFF8
UOS
DBF

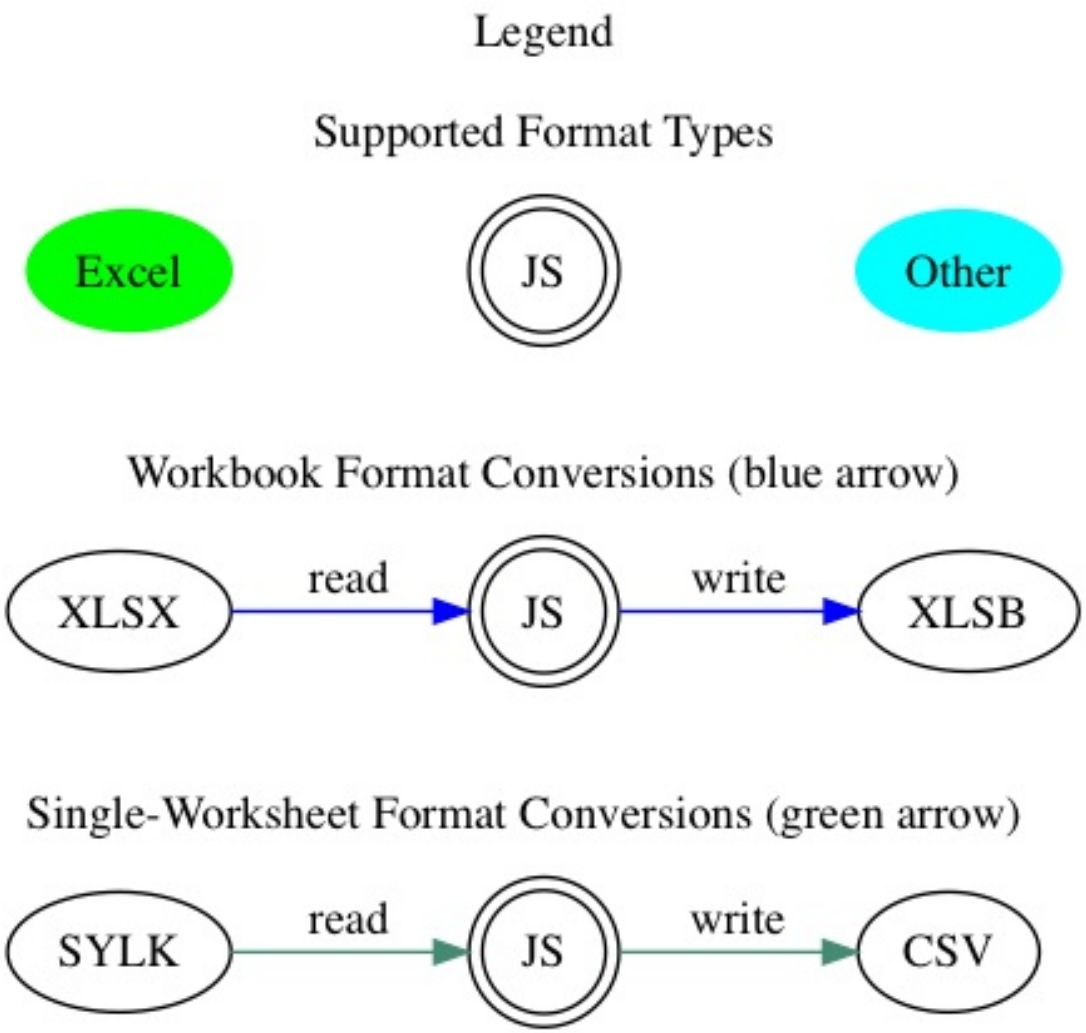


Table of Contents

Installation

With npm:

```
$ npm install xlsx
```

In the browser:

```
<script lang="javascript" src="dist/xlsx.core.min.js"></script>
```

With bower:

```
$ bower install js-xlsx
```

CDNjs automatically pulls the latest version and makes all versions available at http://cdnjs.com/libraries/xlsx

JS Ecosystem Demos

The `demos` directory includes sample projects for:

- `angular`
- `browserify`
- `Adobe ExtendScript`
- `phantomjs`
- `requirejs`
- `systemjs`
- `webpack`

Optional Modules

The node version automatically requires modules for additional features. Some of these modules are rather large in size and are only needed in special circumstances, so they do not ship with the core. For browser use, they must be included directly:

```
<!-- international support from js-codepage -->
<script src="dist/cpexcel.js"></script>
```

An appropriate version for each dependency is included in the dist/ directory.

The complete single-file version is generated at `dist/xlsx.full.min.js`

Webpack and browserify builds include optional modules by default. Webpack can be configured to remove support with `resolve.alias`:

```
  /* uncomment the lines below to remove support */
  resolve: {
    alias: { "./dist/cpexcel.js": "" } // <-- omit international support
```

```
  }
```

ECMAScript 5 Compatibility

Since xlsx.js uses ES5 functions like `Array#forEach`, older browsers require Polyfills. This repo and the gh-pages branch include a shim

To use the shim, add the shim before the script tag that loads xlsx.js:

```
<script type="text/javascript" src="/path/to/shim.js"></script>
```

Parsing Workbooks

For parsing, the first step is to read the file. This involves acquiring the data and feeding it into the library. Here are a few common scenarios:

- node readFile:

```
if(typeof require !== 'undefined') XLSX = require('xlsx');
var workbook = XLSX.readFile('test.xlsx');
/* DO SOMETHING WITH workbook HERE */
```

- Browser DOM Table element:

```
var worksheet = XLSX.utils.table_to_book(document.getElementById('tableau'));
/* DO SOMETHING WITH workbook HERE */
```

- ajax (for a more complete example that works in older browsers, check the demo at http://oss.sheetjs.com/js-xlsx/ajax.html):

```
/* set up XMLHttpRequest */
var url = "test_files/formula_stress_test_ajax.xlsx";
var oReq = new XMLHttpRequest();
oReq.open("GET", url, true);
oReq.responseType = "arraybuffer";

oReq.onload = function(e) {
  var arraybuffer = oReq.response;
```

```
  /* convert data to binary string */
  var data = new Uint8Array(arraybuffer);
  var arr = new Array();
  for(var i = 0; i != data.length; ++i) arr[i] = String.fromCharCode(data[i]);
  var bstr = arr.join("");

  /* Call XLSX */
  var workbook = XLSX.read(bstr, {type:"binary"});

  /* DO SOMETHING WITH workbook HERE */
}

oReq.send();
```

- HTML5 drag-and-drop using readAsBinaryString or readAsArrayBuffer: note: readAsBinaryString and readAsArrayBuffer may not be available in every browser. Use dynamic feature tests to determine which method to use.

```
/* processing array buffers, only required for readAsArrayBuffer */
function fixdata(data) {
  var o = "", l = 0, w = 10240;
  for(; l<data.byteLength/w; ++l) o+=String.fromCharCode.apply(null,new Uint8Ar
ray(data.slice(l*w,l*w+w)));
  o+=String.fromCharCode.apply(null, new Uint8Array(data.slice(l*w)));
  return o;
}

var rABS = true; // true: readAsBinaryString ; false: readAsArrayBuffer
/* set up drag-and-drop event */
function handleDrop(e) {
  e.stopPropagation();
  e.preventDefault();
  var files = e.dataTransfer.files;
  var i,f;
  for (i = 0; i != files.length; ++i) {
    f = files[i];
    var reader = new FileReader();
    var name = f.name;
    reader.onload = function(e) {
      var data = e.target.result;

      var workbook;
      if(rABS) {
        /* if binary string, read with type 'binary' */
        workbook = XLSX.read(data, {type: 'binary'});
      } else {
        /* if array buffer, convert to base64 */
```

```
        var arr = fixdata(data);
        workbook = XLSX.read(btoa(arr), {type: 'base64'});
      }

      /* DO SOMETHING WITH workbook HERE */
    };
    if(rABS) reader.readAsBinaryString(f);
    else reader.readAsArrayBuffer(f);
  }
}
drop_dom_element.addEventListener('drop', handleDrop, false);
```

- HTML5 input file element using readAsBinaryString or readAsArrayBuffer:

```
/* fixdata and rABS are defined in the drag and drop example */
function handleFile(e) {
  var files = e.target.files;
  var i,f;
  for (i = 0; i != files.length; ++i) {
    f = files[i];
    var reader = new FileReader();
    var name = f.name;
    reader.onload = function(e) {
      var data = e.target.result;

      var workbook;
      if(rABS) {
        /* if binary string, read with type 'binary' */
        workbook = XLSX.read(data, {type: 'binary'});
      } else {
        /* if array buffer, convert to base64 */
        var arr = fixdata(data);
        workbook = XLSX.read(btoa(arr), {type: 'base64'});
      }

      /* DO SOMETHING WITH workbook HERE */
    };
    reader.readAsBinaryString(f);
  }
}
input_dom_element.addEventListener('change', handleFile, false);
```

Complete examples:

- http://oss.sheetjs.com/js-xlsx/ HTML5 File API / Base64 Text / Web Workers

Note that older versions of IE do not support HTML5 File API, so the base64 mode is used for testing. On OSX you can get the base64 encoding with:

```
$ <target_file base64 | pbcopy
```

On Windows XP and up you can get the base64 encoding using `certutil`:

```
> certutil -encode target_file target_file.b64
```

(note: You have to open the file and remove the header and footer lines)

- http://oss.sheetjs.com/js-xlsx/ajax.html XMLHttpRequest

Note on Streaming Read

The most common and interesting formats (XLS, XLSX/M, XLSB, ODS) are ultimately ZIP or CFB containers of files. Neither format puts the directory structure at the beginning of the file: ZIP files place the Central Directory records at the end of the logical file, while CFB files can place the FAT structure anywhere in the file! As a result, to properly handle these formats, a streaming function would have to buffer the entire file before commencing. That belies the expectations of streaming, so we do not provide any streaming read API. If you really want to stream, there are node modules like `concat-stream` that will do the buffering for you.

Working with the Workbook

The full object format is described later in this README.

This example extracts the value stored in cell A1 from the first worksheet:

```
var first_sheet_name = workbook.SheetNames[0];
var address_of_cell = 'A1';

/* Get worksheet */
var worksheet = workbook.Sheets[first_sheet_name];

/* Find desired cell */
var desired_cell = worksheet[address_of_cell];
```

```
/* Get the value */
var desired_value = (desired_cell ? desired_cell.v : undefined);
```

Complete examples:

- https://github.com/SheetJS/js-xlsx/blob/master/bin/xlsx.njs node

The node version installs a command line tool `xlsx` which can read spreadsheet files and output the contents in various formats. The source is available at `xlsx.njs` in the bin directory.

Some helper functions in `XLSX.utils` generate different views of the sheets:

- `XLSX.utils.sheet_to_csv` generates CSV
- `XLSX.utils.sheet_to_json` generates an array of objects
- `XLSX.utils.sheet_to_formulae` generates a list of formulae

Writing Workbooks

For writing, the first step is to generate output data. The helper functions `write` and `writeFile` will produce the data in various formats suitable for dissemination. The second step is to actual share the data with the end point. Assuming `workbook` is a workbook object:

- nodejs write to file:

```
/* output format determined by filename */
XLSX.writeFile(workbook, 'out.xlsx');
/* at this point, out.xlsx is a file that you can distribute */
```

- browser generate binary blob and "download" to client (using FileSaver.js for download):

```
/* bookType can be 'xlsx' or 'xlsm' or 'xlsb' or 'ods' */
var wopts = { bookType:'xlsx', bookSST:false, type:'binary' };

var wbout = XLSX.write(workbook,wopts);

function s2ab(s) {
  var buf = new ArrayBuffer(s.length);
  var view = new Uint8Array(buf);
```

```
  for (var i=0; i!=s.length; ++i) view[i] = s.charCodeAt(i) & 0xFF;
  return buf;
}

/* the saveAs call downloads a file on the local machine */
saveAs(new Blob([s2ab(wbout)],{type:"application/octet-stream"}), "test.xlsx");
```

Complete examples:

- http://sheetjs.com/demos/writexlsx.html generates a simple file
- http://git.io/WEK88Q writing an array of arrays in nodejs
- http://sheetjs.com/demos/table.html exporting an HTML table

Streaming Write

The streaming write functions are available in the `XLSX.stream` object. They take the same arguments as the normal write functions but return a readable stream. They are only exposed in node.

- `XLSX.stream.to_csv` is the streaming version of `XLSX.utils.sheet_to_csv`.
- `XLSX.stream.to_html` is the streaming version of the HTML output type.

https://github.com/sheetjs/sheetaki pipes write streams to nodejs response.

Interface

`XLSX` is the exposed variable in the browser and the exported node variable

`XLSX.version` is the version of the library (added by the build script).

`XLSX.SSF` is an embedded version of the format library.

Parsing functions

`XLSX.read(data, read_opts)` attempts to parse `data`.

`XLSX.readFile(filename, read_opts)` attempts to read `filename` and parse.

Parse options are described in the Parsing Options section.

Writing functions

`XLSX.write(wb, write_opts)` attempts to write the workbook `wb`

`XLSX.writeFile(wb, filename, write_opts)` attempts to write `wb` to `filename`

`XLSX.writeFileAsync(filename, wb, o, cb)` attempts to write `wb` to `filename`. If `o` is omitted, the writer will use the third argument as the callback.

`XLSX.stream` contains a set of streaming write functions.

Write options are described in the Writing Options section.

Utilities

Utilities are available in the `XLSX.utils` object:

Importing:

- `aoa_to_sheet` converts an array of arrays of JS data to a worksheet.

Exporting:

- `sheet_to_json` converts a worksheet object to an array of JSON objects. `sheet_to_row_object_array` is an alias that will be removed in the future.
- `sheet_to_csv` generates delimiter-separated-values output.
- `sheet_to_formulae` generates a list of the formulae (with value fallbacks).

Exporters are described in the Utility Functions section.

Cell and cell address manipulation:

- `format_cell` generates the text value for a cell (using number formats)
- `{en,de}code_{row,col}` convert between 0-indexed rows/cols and A1 forms.
- `{en,de}code_cell` converts cell addresses
- `{en,de}code_range` converts cell ranges

Common Spreadsheet Format

js-xlsx conforms to the Common Spreadsheet Format (CSF):

General Structures

Cell address objects are stored as `{c:C, r:R}` where `C` and `R` are 0-indexed column and row numbers, respectively. For example, the cell address `B5` is represented by the object `{c:1, r:4}`.

Cell range objects are stored as `{s:S, e:E}` where `S` is the first cell and `E` is the last cell in the range. The ranges are inclusive. For example, the range `A3:B7` is represented by the object `{s:{c:0, r:2}, e:{c:1, r:6}}`. Utils use the following pattern to walk each of the cells in a range:

```
for(var R = range.s.r; R <= range.e.r; ++R) {
  for(var C = range.s.c; C <= range.e.c; ++C) {
    var cell_address = {c:C, r:R};
  }
}
```

Cell Object

Key	Description
`v`	raw value (see Data Types section for more info)
`w`	formatted text (if applicable)
`t`	cell type: `b` Boolean, `n` Number, `e` error, `s` String, `d` Date
`f`	cell formula encoded as an A1-style string (if applicable)
`F`	range of enclosing array if formula is array formula (if applicable)
`r`	rich text encoding (if applicable)
`h`	HTML rendering of the rich text (if applicable)
`c`	comments associated with the cell
`z`	number format string associated with the cell (if requested)
`l`	cell hyperlink object (.Target holds link, .Tooltip is tooltip)
`s`	the style/theme of the cell (if applicable)

Built-in export utilities (such as the CSV exporter) will use the `w` text if it is available. To change a value, be sure to delete `cell.w` (or set it to `undefined`) before attempting to export. The utilities will regenerate the `w` text from the number format (`cell.z`) and the raw value if possible.

The actual array formula is stored in the `f` field of the first cell in the array range. Other cells in the range will omit the `f` field.

Data Types

The raw value is stored in the `v` field, interpreted based on the `t` field.

Type `b` is the Boolean type. `v` is interpreted according to JS truth tables.

Type `e` is the Error type. `v` holds the number and `w` holds the common name:

Value	Error Meaning
`0x00`	`#NULL!`
`0x07`	`#DIV/0!`
`0x0F`	`#VALUE!`
`0x17`	`#REF!`
`0x1D`	`#NAME?`
`0x24`	`#NUM!`
`0x2A`	`#N/A`
`0x2B`	`#GETTING_DATA`

Type `n` is the Number type. This includes all forms of data that Excel stores as numbers, such as dates/times and Boolean fields. Excel exclusively uses data that can be fit in an IEEE754 floating point number, just like JS Number, so the `v` field holds the raw number. The `w` field holds formatted text. Dates are stored as numbers by default and converted with `XLSX.SSF.parse_date_code`.

Type `d` is the Date type, generated only when the option `cellDates` is passed. Since JSON does not have a natural Date type, parsers are generally expected to store ISO 8601 Date strings like you would get from `date.toISOString()`. On the

other hand, writers and exporters should be able to handle date strings and JS Date objects. Note that Excel disregards timezone modifiers and treats all dates in the local timezone. js-xlsx does not correct for this error.

Type `s` is the String type. `v` should be explicitly stored as a string to avoid possible confusion.

Type `z` represents blank stub cells. These do not have any data or type, and are not processed by any of the core library functions. By default these cells will not be generated; the parser `sheetStubs` option must be set to `true`.

Dates

By default, Excel stores dates as numbers with a format code that specifies date processing. For example, the date `19-Feb-17` is stored as the number `42785` with a number format of `d-mmm-yy`. The `SSF` module understands number formats and performs the appropriate conversion.

XLSX also supports a special date type `d` where the data is an ISO 8601 date string. The formatter converts the date back to a number.

The default behavior for all parsers is to generate number cells. Setting `cellDates` to true will force the generators to store dates.

Sheet Objects

Each key that does not start with `!` maps to a cell (using `A-1` notation)

`sheet[address]` returns the cell object for the specified address.

Special sheet keys (accessible as `sheet[key]`, each starting with `!`):

- `sheet['!ref']`: A-1 based range representing the sheet range. Functions that work with sheets should use this parameter to determine the range. Cells that are assigned outside of the range are not processed. In particular, when writing a sheet by hand, cells outside of the range are not included

  Functions that handle sheets should test for the presence of `!ref` field. If the `!ref` is omitted or is not a valid range, functions are free to treat the sheet as empty or attempt to guess the range. The standard utilities that ship with this library treat sheets as empty (for example, the CSV output is empty string).

When reading a worksheet with the `sheetRows` property set, the ref parameter will use the restricted range. The original range is set at `ws['!fullref']`

- `sheet['!margins']`: Object representing the page margins. The default values follow Excel's "normal" preset. Excel also has a "wide" and a "narrow" preset but they are stored as raw measurements. The main properties are listed below:

key	description	"normal"	"wide"	"narrow"
`left`	left margin (inches)	`0.7`	`1.0`	`0.25`
`right`	right margin (inches)	`0.7`	`1.0`	`0.25`
`top`	top margin (inches)	`0.75`	`1.0`	`0.75`
`bottom`	bottom margin (inches)	`0.75`	`1.0`	`0.75`
`header`	header margin (inches)	`0.3`	`0.5`	`0.3`
`footer`	footer margin (inches)	`0.3`	`0.5`	`0.3`

```
/* Set worksheet sheet to "normal" */
sheet["!margins"] = { left:0.7, right:0.7, top:0.75, bottom:0.75, header:0.3, f
ooter:0.3 }
/* Set worksheet sheet to "wide" */
sheet["!margins"] = { left:1.0, right:1.0, top:1.0, bottom:1.0, header:0.5, foo
ter:0.5 }
/* Set worksheet sheet to "narrow" */
sheet["!margins"] = { left:0.25, right:0.25, top:0.75, bottom:0.75, header:0.3,
 footer:0.3 }
```

Worksheet Object

In addition to the base sheet keys, worksheets also add:

- `ws['!cols']`: array of column properties objects. Column widths are actually stored in files in a normalized manner, measured in terms of the "Maximum Digit Width" (the largest width of the rendered digits 0-9, in pixels). When parsed, the column objects store the pixel width in the `wpx` field, character width in the `wch` field, and the maximum digit width in the `MDW` field.

- `ws['!merges']` : array of range objects corresponding to the merged cells in the worksheet. Plaintext utilities are unaware of merge cells. CSV export will write all cells in the merge range if they exist, so be sure that only the first cell (upper-left) in the range is set.

- `ws['protect']` : object of write sheet protection properties. The `password` key specifies the password. The writer uses the XOR obfuscation method. The following keys control the sheet protection (same as ECMA-376 18.3.1.85):

key	functionality disabled if value is true
`selectLockedCells`	Select locked cells
`selectUnlockedCells`	Select unlocked cells
`formatCells`	Format cells
`formatColumns`	Format columns
`formatRows`	Format rows
`insertColumns`	Insert columns
`insertRows`	Insert rows
`insertHyperlinks`	Insert hyperlinks
`deleteColumns`	Delete columns
`deleteRows`	Delete rows
`sort`	Sort
`autoFilter`	Filter
`pivotTables`	Use PivotTable reports
`objects`	Edit objects
`scenarios`	Edit scenarios

- `ws['!autofilter']` : AutoFilter object following the schema:

```
type AutoFilter = {
    ref:string; // A-1 based range representing the AutoFilter table range
}
```

Chartsheet Object

Chartsheets are represented as standard sheets. They are distinguished with the `!type` property set to `"chart"`.

The underlying data and `!ref` refer to the cached data in the chartsheet. The first row of the chartsheet is the underlying header.

Workbook Object

`workbook.SheetNames` is an ordered list of the sheets in the workbook

`wb.Sheets[sheetname]` returns an object representing the worksheet.

`wb.Props` is an object storing the standard properties. `wb.Custprops` stores custom properties. Since the XLS standard properties deviate from the XLSX standard, XLS parsing stores core properties in both places.

`wb.WBProps` includes more workbook-level properties:

- Excel supports two epochs (January 1 1900 and January 1 1904), see 1900 vs. 1904 Date System. The workbook's epoch can be determined by examining the workbook's `wb.WBProps.date1904` property.

Workbook File Properties

The various file formats use different internal names for file properties. The workbook `Props` object normalizes the names:

JS Name	Excel Description
Title	Summary tab "Title"
Subject	Summary tab "Subject"
Author	Summary tab "Author"
Manager	Summary tab "Manager"
Company	Summary tab "Company"
Category	Summary tab "Category"
Keywords	Summary tab "Keywords"

Comments	Summary tab "Comments"
LastAuthor	Statistics tab "Last saved by"
CreatedDate	Statistics tab "Created"

For example, to set the workbook title property:

```
if(!wb.Props) wb.Props = {};
wb.Props.Title = "Insert Title Here";
```

Custom properties are added in the workbook `Custprops` object:

```
if(!wb.Custprops) wb.Custprops = {};
wb.Custprops["Custom Property"] = "Custom Value";
```

Writers will process the `Props` key of the options object:

```
/* force the Author to be "SheetJS" */
XLSX.write(wb, {Props:{Author:"SheetJS"}});
```

Workbook-Level Attributes

`wb.Workbook` stores workbook level attributes.

Defined Names

`wb.Workbook.Names` is an array of defined name objects which have the keys:

Key	Description
`Sheet`	Name scope. Sheet Index (0 = first sheet) or `null` (Workbook)
`Name`	Case-sensitive name. Standard rules apply **
`Ref`	A1-style Reference (e.g. `"Sheet1!$A$1:$D$20"`)
`Comment`	Comment (only applicable for XLS/XLSX/XLSB)

Excel allows two sheet-scoped defined names to share the same name. However, a sheet-scoped name cannot collide with a workbook-scope name. Workbook writers may not enforce this constraint.

Document Features

Even for basic features like date storage, the official Excel formats store the same content in different ways. The parsers are expected to convert from the underlying file format representation to the Common Spreadsheet Format. Writers are expected to convert from CSF back to the underlying file format.

Formulae

The A1-style formula string is stored in the `f` field. Even though different file formats store the formulae in different ways, the formats are translated. Even though some formats store formulae with a leading equal sign, CSF formulae do not start with `=`.

The worksheet representation of A1=1, A2=2, A3=A1+A2:

```
{
    "!ref": "A1:A3",
    A1: { t:'n', v:1 },
    A2: { t:'n', v:2 },
    A3: { t:'n', v:3, f:'A1+A2' }
}
```

Shared formulae are decompressed and each cell has the formula corresponding to its cell. Writers generally do not attempt to generate shared formulae.

Cells with formula entries but no value will be serialized in a way that Excel and other spreadsheet tools will recognize. This library will not automatically compute formula results! For example, to compute `BESSELJ` in a worksheet:

```
{
    "!ref": "A1:A3",
    A1: { t:'n', v:3.14159 },
    A2: { t:'n', v:2 },
    A3: { t:'n', f:'BESSELJ(A1,A2)' }
}
```

Array Formulae

Array formulae are stored in the top-left cell of the array block. All cells of an array formula have a `F` field corresponding to the range. A single-cell formula can be distinguished from a plain formula by the presence of `F` field.

For example, setting the cell `C1` to the array formula `{=SUM(A1:A3*B1:B3)}`:

```
worksheet['C1'] = { t:'n', f: "SUM(A1:A3*B1:B3)", F:"C1:C1" };
```

For a multi-cell array formula, every cell has the same array range but only the first cell specifies the formula. Consider `D1:D3=A1:A3*B1:B3`:

```
worksheet['D1'] = { t:'n', F:"D1:D3", f:"A1:A3*B1:B3" };
worksheet['D2'] = { t:'n', F:"D1:D3" };
worksheet['D3'] = { t:'n', F:"D1:D3" };
```

Utilities and writers are expected to check for the presence of a `F` field and ignore any possible formula element `f` in cells other than the starting cell. They are not expected to perform validation of the formulae!

Formula Output

The `sheet_to_formulae` method generates one line per formula or array formula. Array formulae are rendered in the form `range=formula` while plain cells are rendered in the form `cell=formula or value`. Note that string literals are prefixed with an apostrophe `'`, consistent with Excel's formula bar display.

Formulae File Format Details

Storage Representation	Formats	Read	Write
A1-style strings	XLSX	O	O
RC-style strings	XLML and plaintext	O	O
BIFF Parsed formulae	XLSB and all XLS formats	O	
OpenFormula formulae	ODS/FODS/UOS	O	O

Since Excel prohibits named cells from colliding with names of A1 or RC style cell references, a (not-so-simple) regex conversion is possible. BIFF Parsed formulae have to be explicitly unwound. OpenFormula formulae can be converted with regexes for the most part.

Column Properties

Excel internally stores column widths in a nebulous "Max Digit Width" form. The Max Digit Width is the width of the largest digit when rendered. The internal width must be an integer multiple of the the width divided by 256. ECMA-376 describes a formula for converting between pixels and the internal width.

Given the constraints, it is possible to determine the MDW without actually inspecting the font! The parsers guess the pixel width by converting from width to pixels and back, repeating for all possible MDW and selecting the MDW that minimizes the error. XLML actually stores the pixel width, so the guess works in the opposite direction.

The `!cols` array in each worksheet, if present, is a collection of `ColInfo` objects which have the following properties:

```
type ColInfo = {
    MDW?:number;  // Excel's "Max Digit Width" unit, always integral
    width:number; // width in Excel's "Max Digit Width", width*256 is integral
    wpx?:number;  // width in screen pixels
    wch?:number;  // intermediate character calculation
};
```

Even though all of the information is made available, writers are expected to follow the priority order:

1) use `width` field if available 2) use `wpx` pixel width if available 3) use `wch` character count if available

Hyperlinks

Hyperlinks are stored in the `l` key of cell objects. The `Target` field of the hyperlink object is the target of the link, including the URI fragment. Tooltips are stored in the `Tooltip` field and are displayed when you move your mouse over the text.

For example, the following snippet creates a link from cell `A3` to http://sheetjs.com with the tip `"Find us @ SheetJS.com!"`:

```
ws['A3'].l = { Target:"http://sheetjs.com", Tooltip:"Find us @ SheetJS.com!" };
```

Note that Excel does not automatically style hyperlinks -- they will generally be displayed as normal text.

Cell Comments

Cell comments are objects stored in the `c` array of cell objects. The actual contents of the comment are split into blocks based on the comment author. The `a` field of each comment object is the author of the comment and the `t` field is the plaintext representation.

For example, the following snippet appends a cell comment into cell `A1`:

```
if(!ws.A1.c) ws.A1.c = [];
ws.A1.c.push({a:"SheetJS", t:"I'm a little comment, short and stout!"});
```

Note: XLSB enforces a 54 character limit on the Author name. Names longer than 54 characters may cause issues with other formats.

Sheet Visibility

Excel enables hiding sheets in the lower tab bar. The sheet data is stored in the file but the UI does not readily make it available. Standard hidden sheets are revealed in the unhide menu. Excel also has "very hidden" sheets which cannot be revealed in the menu. It is only accessible in the VB Editor!

The visibility setting is stored in the `Hidden` property of the sheet props array. The values are:

Value	Definition
0	Visible
1	Hidden
2	Very Hidden

With https://rawgit.com/SheetJS/test_files/master/sheet_visibility.xlsx:

```
> wb.Workbook.Sheets.map(function(x) { return [x.name, x.Hidden] })
[ [ 'Visible', 0 ], [ 'Hidden', 1 ], [ 'VeryHidden', 2 ] ]
```

Non-Excel formats do not support the Very Hidden state. The best way to test if a sheet is visible is to check if the `Hidden` property is logical truth:

```
> wb.Workbook.Sheets.map(function(x) { return [x.name, !x.Hidden] })
[ [ 'Visible', true ], [ 'Hidden', false ], [ 'VeryHidden', false ] ]
```

Parsing Options

The exported `read` and `readFile` functions accept an options argument:

Option Name	Default	Description
type		Input data encoding (see Input Type below)
cellFormula	true	Save formulae to the .f field
cellHTML	true	Parse rich text and save HTML to the .h field
cellNF	false	Save number format string to the .z field
cellStyles	false	Save style/theme info to the .s field
cellDates	false	Store dates as type `d` (default is `n`)
sheetStubs	false	Create cell objects of type `z` for stub cells
sheetRows	0	If >0, read the first `sheetRows` rows **
bookDeps	false	If true, parse calculation chains
bookFiles	false	If true, add raw files to book object **
bookProps	false	If true, only parse enough to get book metadata **
bookSheets	false	If true, only parse enough to get the sheet names

bookVBA	false	If true, expose vbaProject.bin to `vbaraw` field **
password	""	If defined and file is encrypted, use password **
WTF	false	If true, throw errors on unexpected file features **

- Even if `cellNF` is false, formatted text will be generated and saved to `.w`
- In some cases, sheets may be parsed even if `bookSheets` is false.
- `bookSheets` and `bookProps` combine to give both sets of information
- `Deps` will be an empty object if `bookDeps` is falsy
- `bookFiles` behavior depends on file type:
  - `keys` array (paths in the ZIP) for ZIP-based formats
  - `files` hash (mapping paths to objects representing the files) for ZIP
  - `cfb` object for formats using CFB containers
- `sheetRows-1` rows will be generated when looking at the JSON object output (since the header row is counted as a row when parsing the data)
- `bookVBA` merely exposes the raw vba object. It does not parse the data.
- Currently only XOR encryption is supported. Unsupported error will be thrown for files employing other encryption methods.
- WTF is mainly for development. By default, the parser will suppress read errors on single worksheets, allowing you to read from the worksheets that do parse properly. Setting `WTF:1` forces those errors to be thrown.

The defaults are enumerated in bits/84_defaults.js

Input Type

Strings can be interpreted in multiple ways. The `type` parameter for `read` tells the library how to parse the data argument:

`type`	expected input
`"base64"`	string: base64 encoding of the file
`"binary"`	string: binary string (`n` -th byte is `data.charCodeAt(n)`)
`"buffer"`	nodejs Buffer
`"array"`	array: array of 8-bit unsigned int (`n` -th byte is `data[n]`)
`"file"`	string: filename that will be read and processed (nodejs only)

Guessing File Type

Excel and other spreadsheet tools read the first few bytes and apply other heuristics to determine a file type. This enables file type punning: renaming files with the `.xls` extension will tell your computer to use Excel to open the file but Excel will know how to handle it. This library applies similar logic:

Byte 0	Raw File Type	Spreadsheet Types
`0xD0`	CFB Container	BIFF 5/8 or password-protected XLSX/XLSB or WQ3/QPW
`0x09`	BIFF Stream	BIFF 2/3/4/5
`0x3C`	XML/HTML	SpreadsheetML / Flat ODS / UOS1 / HTML / plaintext
`0x50`	ZIP Archive	XLSB or XLSX/M or ODS or UOS2 or plaintext
`0x49`	Plain Text	SYLK or plaintext
`0x54`	Plain Text	DIF or plaintext
`0xFE`	UTF16 Encoded	SpreadsheetML or Flat ODS or UOS1 or plaintext
`0x00`	Record Stream	Lotus WK* or Quattro Pro or plaintext

DBF files are detected based on the first byte as well as the third and fourth bytes (corresponding to month and day of the file date)

Plaintext format guessing follows the priority order:

Format	Test
HTML	starts with \<html
XML	starts with \<
DSV	starts with `/sep=.$/`, separator is the specified character
TSV	one of the first 1024 characters is a tab char `"\t"`
CSV	one of the first 1024 characters is a comma char `","`

PRN	(default)

Writing Options

The exported `write` and `writeFile` functions accept an options argument:

Option Name	Default	Description
type		Output data encoding (see Output Type below)
cellDates	`false`	Store dates as type `d` (default is `n`)
bookSST	`false`	Generate Shared String Table **
bookType	`"xlsx"`	Type of Workbook (see below for supported formats)
sheet	`""`	Name of Worksheet for single-sheet formats **
compression	`false`	Use ZIP compression for ZIP-based formats **
Props		Override workbook properties when writing **
themeXLSX		Override theme XML when writing XLSX/XLSB/XLSM **

- `bookSST` is slower and more memory intensive, but has better compatibility with older versions of iOS Numbers
- The raw data is the only thing guaranteed to be saved. Features not described in this README may not be serialized.
- `cellDates` only applies to XLSX output and is not guaranteed to work with third-party readers. Excel itself does not usually write cells with type `d` so non-Excel tools may ignore the data or blow up in the presence of dates.
- `Props` is an object mirroring the workbook `Props` field. See the table from the Workbook File Properties section.
- if specified, the string from `themeXLSX` will be saved as the primary theme for XLSX/XLSB/XLSM files (to `xl/theme/theme1.xml` in the ZIP)

Supported Output Formats

For broad compatibility with third-party tools, this library supports many output formats. The specific file type is controlled with `bookType` option:

bookType	file ext	container	sheets	Description
`xlsx`	`.xlsx`	ZIP	multi	Excel 2007+ XML Format
`xlsm`	`.xlsm`	ZIP	multi	Excel 2007+ Macro XML Format
`xlsb`	`.xlsb`	ZIP	multi	Excel 2007+ Binary Format
`biff2`	`.xls`	none	single	Excel 2.0 Worksheet format
`xlml`	`.xls`	none	multi	Excel 2003-2004 (SpreadsheetML)
`ods`	`.ods`	ZIP	multi	OpenDocument Spreadsheet
`fods`	`.fods`	none	multi	Flat OpenDocument Spreadsheet
`csv`	`.csv`	none	single	Comma Separated Values
`txt`	`.txt`	none	single	UTF-16 Unicode Text (TXT)
`sylk`	`.sylk`	none	single	Symbolic Link (SYLK)
`html`	`.html`	none	single	HTML Document
`dif`	`.dif`	none	single	Data Interchange Format (DIF)
`prn`	`.prn`	none	single	Lotus Formatted Text

- `compression` only applies to formats with ZIP containers.
- Formats that only support a single sheet require a `sheet` option specifying the worksheet. If the string is empty, the first worksheet is used.
- `writeFile` will automatically guess the output file format based on the file extension if `bookType` is not specified. It will choose the first format in the aforementioned table that matches the extension.

Output Type

The `type` argument for `write` mirrors the `type` argument for `read`:

`type`	output
`"base64"`	string: base64 encoding of the file
`"binary"`	string: binary string (`n` -th byte is `data.charCodeAt(n)`)
`"buffer"`	nodejs Buffer
`"file"`	string: name of file to be written (nodejs only)

Utility Functions

The `sheet_to_*` functions accept a worksheet and an optional options object.

The `*_to_sheet` functions accept a data object and an optional options object.

The examples are based on the following worksheet:

```
XXX| A | B | C | D | E | F | G |
---+---+---+---+---+---+---+---+
 1 | S | h | e | e | t | J | S |
 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
```

Array of Arrays Input

`XLSX.utils.aoa_to_sheet` takes an array of arrays of JS values and returns a worksheet resembling the input data. Numbers, Booleans and Strings are stored as the corresponding styles. Dates are stored as date or numbers. Array holes and explicit `undefined` values are skipped. `null` values may be stubbed. All other values are stored as strings. The function takes an options argument:

Option Name	Default	Description
dateNF	fmt 14	Use specified date format in string output
cellDates	false	Store dates as type `d` (default is `n`)

sheetStubs	false	Create cell objects of type `z` for `null` values

To generate the example sheet:

```
var ws = XLSX.utils.aoa_to_sheet([
    "SheetJS".split(""),
    [1,2,3,4,5,6,7],
    [2,3,4,5,6,7,8]
]);
```

HTML Table Input

`XLSX.utils.table_to_sheet` takes a table DOM element and returns a worksheet resembling the input table. Numbers are parsed. All other data will be stored as strings.

`XLSX.utils.table_to_book` produces a minimal workbook based on the worksheet.

To generate the example sheet, start with the HTML table:

```
<table id="sheetjs">
<tr><td>S</td><td>h</td><td>e</td><td>e</td><td>t</td><td>J</td><td>S</td></tr>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
</table>
```

To process the table:

```
var tbl = document.getElementById('sheetjs');
var wb = XLSX.utils.table_to_book(tbl);
```

Note: `XLSX.read` can handle HTML represented as strings.

Formulae Output

`XLSX.utils.sheet_to_formulae` generates an array of commands that represent how a person would enter data into an application. Each entry is of the form `A1-cell-address=formula-or-value`. String literals are prefixed with a `'` in accordance with Excel. For the example sheet:

```
> var o = XLSX.utils.sheet_to_formulae(ws);
> o.filter(function(v, i) { return i % 5 === 0; });
[ 'A1=\'S', 'F1=\'J', 'D2=4', 'B3=3', 'G3=8' ]
```

Delimiter-Separated Output

As an alternative to the `writeFile` CSV type, `XLSX.utils.sheet_to_csv` also produces CSV output. The function takes an options argument:

Option Name	Default	Description
FS	`","`	"Field Separator" delimiter between fields
RS	`"\n"`	"Record Separator" delimiter between rows
dateNF	fmt 14	Use specified date format in string output
strip	false	Remove trailing field separators in each record **
blankrows	true	Include blank lines in the CSV output

- `strip` will remove trailing commas from each line under default `FS/RS`
- blankrows must be set to `false` to skip blank lines.

For the example sheet:

```
> console.log(XLSX.utils.sheet_to_csv(ws));
S,h,e,e,t,J,S
1,2,3,4,5,6,7
2,3,4,5,6,7,8
> console.log(XLSX.utils.sheet_to_csv(ws, {FS:"\t"}));
S    h    e    e    t    J    S
1    2    3    4    5    6    7
2    3    4    5    6    7    8
> console.log(XLSX.utils.sheet_to_csv(ws,{FS:":",RS:"|"}));
S:h:e:e:t:J:S|1:2:3:4:5:6:7|2:3:4:5:6:7:8|
```

UTF-16 Unicode Text

The `txt` output type uses the tab character as the field separator. If the codepage library is available (included in the full distribution but not core), the output will be encoded in codepage `1200` and the BOM will be prepended.

JSON

`XLSX.utils.sheet_to_json` and the alias `XLSX.utils.sheet_to_row_object_array` generate different types of JS objects. The function takes an options argument:

Option Name	Default	Description
raw	`false`	Use raw values (true) or formatted strings (false)
range	from WS	Override Range (see table below)
header		Control output format (see table below)
dateNF	fmt 14	Use specified date format in string output
defval		Use specified value in place of null or undefined
blankrows	**	Include blank lines in the output **

- `raw` only affects cells which have a format code (`.z`) field or a formatted text (`.w`) field.
- If `header` is specified, the first row is considered a data row; if `header` is not specified, the first row is the header row and not considered data.
- When `header` is not specified, the conversion will automatically disambiguate header entries by affixing `_` and a count starting at `1` . For example, if three columns have header `foo` the output fields are `foo` , `foo_1` , `foo_2`
- `null` values are returned when `raw` is true but are skipped when false.
- If `defval` is not specified, null and undefined values are skipped normally. If specified, all null and undefined points will be filled with `defval`
- When `header` is `1` , the default is to generate blank rows. `blankrows` must be set to `false` to skip blank rows.
- When `header` is not `1` , the default is to skip blank rows. `blankrows` must be truthy to generate blank rows

`range` is expected to be one of:

`range`	Description
(number)	Use worksheet range but set starting row to the value
(string)	Use specified range (A1-style bounded range string)
(default)	Use worksheet range (`ws['!ref']`)

`header` is expected to be one of:

`header`	Description
`1`	Generate an array of arrays ("2D Array")
`"A"`	Row object keys are literal column labels
array of strings	Use specified strings as keys in row objects
(default)	Read and disambiguate first row as keys

If header is not `1`, the row object will contain the non-enumerable property `__rowNum__` that represents the row of the sheet corresponding to the entry.

For the example sheet:

```
> console.log(XLSX.utils.sheet_to_json(ws));
[ { S: 1, h: 2, e: 3, e_1: 4, t: 5, J: 6, S_1: 7 },
  { S: 2, h: 3, e: 4, e_1: 5, t: 6, J: 7, S_1: 8 } ]

> console.log(XLSX.utils.sheet_to_json(ws, {header:1}));
[ [ 'S', 'h', 'e', 'e', 't', 'J', 'S' ],
  [ '1', '2', '3', '4', '5', '6', '7' ],
  [ '2', '3', '4', '5', '6', '7', '8' ] ]

> console.log(XLSX.utils.sheet_to_json(ws, {header:"A"}));
[ { A: 'S', B: 'h', C: 'e', D: 'e', E: 't', F: 'J', G: 'S' },
  { A: '1', B: '2', C: '3', D: '4', E: '5', F: '6', G: '7' },
  { A: '2', B: '3', C: '4', D: '5', E: '6', F: '7', G: '8' } ]
> console.log(XLSX.utils.sheet_to_json(ws, {header:["A","E","I","O","U","6","9"
]}));
[ { '6': 'J', '9': 'S', A: 'S', E: 'h', I: 'e', O: 'e', U: 't' },
  { '6': '6', '9': '7', A: '1', E: '2', I: '3', O: '4', U: '5' },
  { '6': '7', '9': '8', A: '2', E: '3', I: '4', O: '5', U: '6' } ]
```

Example showing the effect of `raw`:

```
> ws['A2'].w = "3";                          // set A2 formatted string value
> console.log(XLSX.utils.sheet_to_json(ws, {header:1}));
[ [ 'S', 'h', 'e', 'e', 't', 'J', 'S' ],
  [ '3', '2', '3', '4', '5', '6', '7' ],     // <-- A2 uses the formatted string

  [ '2', '3', '4', '5', '6', '7', '8' ] ]
> console.log(XLSX.utils.sheet_to_json(ws, {header:1, raw:true}));
[ [ 'S', 'h', 'e', 'e', 't', 'J', 'S' ],
  [ 1, 2, 3, 4, 5, 6, 7 ],                   // <-- A2 uses the raw value
  [ 2, 3, 4, 5, 6, 7, 8 ] ]
```

File Formats

Despite the library name `xlsx`, it supports numerous spreadsheet file formats:

Format	Read	Write
Excel Worksheet/Workbook Formats	:-----:	:-----:
Excel 2007+ XML Formats (XLSX/XLSM)	O	O
Excel 2007+ Binary Format (XLSB BIFF12)	O	O
Excel 2003-2004 XML Format (XML "SpreadsheetML")	O	O
Excel 97-2004 (XLS BIFF8)	O	
Excel 5.0/95 (XLS BIFF5)	O	
Excel 4.0 (XLS/XLW BIFF4)	O	
Excel 3.0 (XLS BIFF3)	O	
Excel 2.0/2.1 (XLS BIFF2)	O	O
Excel Supported Text Formats	:-----:	:-----:
Delimiter-Separated Values (CSV/TXT)	O	O
Data Interchange Format (DIF)	O	O
Symbolic Link (SYLK/SLK)	O	O

Lotus Formatted Text (PRN)	⭕	⭕
UTF-16 Unicode Text (TXT)	⭕	⭕
Other Workbook/Worksheet Formats	:-----:	:-----:
OpenDocument Spreadsheet (ODS)	⭕	⭕
Flat XML ODF Spreadsheet (FODS)	⭕	⭕
Uniform Office Format Spreadsheet (标文通 UOS1/UOS2)	⭕	
dBASE II/III/IV / Visual FoxPro (DBF)	⭕	
Lotus 1-2-3 (WKS/WK1/WK2/WK3/WK4/123)	⭕	
Quattro Pro Spreadsheet (WQ1/WQ2/WB1/WB2/WB3/QPW)	⭕	
Other Common Spreadsheet Output Formats	:-----:	:-----:
HTML Tables	⭕	⭕

Excel 2007+ XML (XLSX/XLSM)

XLSX and XLSM files are ZIP containers containing a series of XML files in accordance with the Open Packaging Conventions (OPC). The XLSM filetype, almost identical to XLSX, is used for files containing macros.

The format is standardized in ECMA-376 and later in ISO/IEC 29500. Excel does not follow the specification, and there are additional documents discussing how Excel deviates from the specification.

Excel 2.0-95 (BIFF2/BIFF3/BIFF4/BIFF5)

BIFF 2/3 XLS are single-sheet streams of binary records. Excel 4 introduced the concept of a workbook (`XLW` files) but also had single-sheet `XLS` format. The structure is largely similar to the Lotus 1-2-3 file formats. BIFF5/8/12 extended the format in various ways but largely stuck to the same record format.

There is no official specification for any of these formats. Excel 95 can write files in these formats, so record lengths and fields were backsolved by writing in all of the supported formats and comparing files. Excel 2016 can generate BIFF5 files, enabling a full suite of file tests starting from XLSX or BIFF2.

Excel 97-2004 Binary (BIFF8)

BIFF8 exclusively uses the Compound File Binary container format, splitting some content into streams within the file. At its core, it still uses an extended version of the binary record format from older versions of BIFF.

The `MS-XLS` specification covers the basics of the file format, and other specifications expand on serialization of features like properties.

Excel 2003-2004 (SpreadsheetML)

Predating XLSX, SpreadsheetML files are simple XML files. There is no official and comprehensive specification, although MS has released whitepapers on the format. Since Excel 2016 can generate SpreadsheetML files, backsolving is pretty straightforward.

Excel 2007+ Binary (XLSB, BIFF12)

Introduced in parallel with XLSX, the XLSB filetype combines BIFF architecture with the content separation and ZIP container of XLSX. For the most part nodes in an XLSX sub-file can be mapped to XLSB records in a corresponding sub-file.

The `MS-XLSB` specification covers the basics of the file format, and other specifications expand on serialization of features like properties.

Delimiter-Separated Values (CSV/TXT)

Excel CSV deviates from RFC4180 in a number of important ways. The generated CSV files should generally work in Excel although they may not work in RFC4180 compatible readers. The parser should generally understand Excel CSV.

Excel TXT uses tab as the delimiter and codepage 1200.

Other Workbook Formats

Support for other formats is generally far XLS/XLSB/XLSX support, due in large part to a lack of publicly available documentation. Test files were produced in the respective apps and compared to their XLS exports to determine structure. The main focus is data extraction.

Lotus 1-2-3 (WKS/WK1/WK2/WK3/WK4/123)

The Lotus formats consist of binary records similar to the BIFF structure. Lotus did release a whitepaper decades ago covering the original WK1 format. Other features were deduced by producing files and comparing to Excel support.

Quattro Pro (WQ1/WQ2/WB1/WB2/WB3/QPW)

The Quattro Pro formats use binary records in the same way as BIFF and Lotus. Some of the newer formats (namely WB3 and QPW) use a CFB enclosure just like BIFF8 XLS.

OpenDocument Spreadsheet (ODS/FODS)

ODS is an XML-in-ZIP format akin to XLSX while FODS is an XML format akin to SpreadsheetML. Both are detailed in the OASIS standard, but tools like LO/OO add undocumented extensions.

Uniform Office Spreadsheet (UOS1/2)

UOS is a very similar format, and it comes in 2 varieties corresponding to ODS and FODS respectively. For the most part, the difference between the formats lies in the names of tags and attributes.

Other Single-Worksheet Formats

Many older formats supported only one worksheet:

dBASE and Visual FoxPro (DBF)

DBF is really a typed table format: each column can only hold one data type and each record omits type information. The parser generates a header row and inserts records starting at the second row of the worksheet.

Multi-file extensions like external memos and tables are currently unsupported, limited by the general ability to read arbitrary files in the web browser.

Symbolic Link (SYLK)

There is no real documentation. All knowledge was gathered by saving files in various versions of Excel to deduce the meaning of fields.

Lotus Formatted Text (PRN)

There is no real documentation, and in fact Excel treats PRN as an output-only file format. Nevertheless we can guess the column widths and reverse-engineer the original layout.

Data Interchange Format (DIF)

There is no unified definition. Visicalc DIF differs from Lotus DIF, and both differ from Excel DIF. Where ambiguous, the parser/writer follows the expected behavior from Excel.

HTML

Excel HTML worksheets include special metadata encoded in styles. For example, `mso-number-format` is a localized string containing the number format. Despite the metadata the output is valid HTML, although it does accept bare `&` symbols.

Testing

Node

`make test` will run the node-based tests. By default it runs tests on files in every supported format. To test a specific file type, set `FMTS` to the format you want to test. Feature-specific tests are avaialble with `make test_misc`

```
$ make test_misc   # run core tests
$ make test        # run full tests
$ make test_xls    # only use the XLS test files
$ make test_xlsx   # only use the XLSX test files
$ make test_xlsb   # only use the XLSB test files
$ make test_xml    # only use the XML test files
$ make test_ods    # only use the ODS test files
```

To enable all errors, set the environment variable `WTF=1`:

```
$ make test        # run full tests
$ WTF=1 make test  # enable all error messages
```

Flow and JSHint/JSCS checks are available:

```
$ make lint        # JSHint and JSCS checks
$ make flow        # make lint + Flow checking
```

Browser

The core in-browser tests are available at `tests/index.html` within this repo. Start a local server and navigate to that directory to run the tests. `make ctestserv` will start a server on port 8000.

`make ctest` will generate the browser fixtures. To add more files, edit the `tests/fixtures.lst` file and add the paths.

To run the full in-browser tests, clone the repo for oss.sheetjs.com and replace the xlsx.js file (then fire up the browser and go to `stress.html`):

```
$ cp xlsx.js ../SheetJS.github.io
$ cd ../SheetJS.github.io
$ simplehttpserver # or "python -mSimpleHTTPServer" or "serve"
$ open -a Chromium.app http://localhost:8000/stress.html
```

Tested Environments

- NodeJS 0.8, 0.9, 0.10, 0.11, 0.12, 4.x, 5.x, 6.x, 7.x
- IE 6/7/8/9/10/11 (IE6-9 browsers require shims for interacting with client)
- Chrome 24+
- Safari 6+
- FF 18+

Tests utilize the mocha testing framework. Travis-CI and Sauce Labs links:

- https://travis-ci.org/SheetJS/js-xlsx for XLSX module in nodejs
- https://semaphoreci.com/sheetjs/js-xlsx for XLSX module in nodejs
- https://travis-ci.org/SheetJS/SheetJS.github.io for XLS* modules
- https://saucelabs.com/u/sheetjs for XLS* modules using Sauce Labs

Test Files

Test files are housed in another repo.

Running `make init` will refresh the `test_files` submodule and get the files.

Contributing

Due to the precarious nature of the Open Specifications Promise, it is very important to ensure code is cleanroom. Consult CONTRIBUTING.md

OSX/Linux

The xlsx.js file is constructed from the files in the `bits` subdirectory. The build script (run `make`) will concatenate the individual bits to produce the script. Before submitting a contribution, ensure that running make will produce the xlsx.js file exactly. The simplest way to test is to add the script:

```
$ git add xlsx.js
$ make clean
$ make
$ git diff xlsx.js
```

To produce the dist files, run `make dist`. The dist files are updated in each version release and *should not be committed between versions*.

Windows

The included `make.cmd` script will build `xlsx.js` from the `bits` directory. Building is as simple as:

```
> make.cmd
```

To prepare dev environment:

```
> npm install -g mocha
> npm install
> mocha -t 30000
```

The normal approach uses a variety of command line tools to grab the test files. For windows users, please download the latest version of the test files snapshot from github

Latest test files snapshot:
https://github.com/SheetJS/test_files/releases/download/20170409/test_files.zip

Download and unzip to the `test_files` subdirectory.

License

Please consult the attached LICENSE file for details. All rights not explicitly granted by the Apache 2.0 License are reserved by the Original Author.

References

ISO/IEC 29500:2012(E) "Information technology — Document description and processing languages — Office Open XML File Formats"

OSP-covered specifications:

- [MS-XLSB]: Excel (.xlsb) Binary File Format
- [MS-XLSX]: Excel (.xlsx) Extensions to the Office Open XML SpreadsheetML File Format
- [MS-OE376]: Office Implementation Information for ECMA-376 Standards Support
- [MS-CFB]: Compound File Binary File Format
- [MS-XLS]: Excel Binary File Format (.xls) Structure Specification
- [MS-ODATA]: Open Data Protocol (OData)
- [MS-OFFCRYPTO]: Office Document Cryptography Structure
- [MS-OLEDS]: Object Linking and Embedding (OLE) Data Structures
- [MS-OLEPS]: Object Linking and Embedding (OLE) Property Set Data Structures
- [MS-OSHARED]: Office Common Data Types and Objects Structures
- [MS-ODRAW]: Office Drawing Binary File Format
- [MS-ODRAWXML]: Office Drawing Extensions to Office Open XML Structure

- [MS-OVBA]: Office VBA File Format Structure
- [MS-CTXLS]: Excel Custom Toolbar Binary File Format
- [MS-XLDM]: Spreadsheet Data Model File Format
- [MS-EXSPXML3]: Excel Calculation Version 2 Web Service XML Schema
- [XLS]: Microsoft Office Excel 97-2007 Binary File Format Specification
- [MS-OI29500]: Office Implementation Information for ISO/IEC 29500 Standards Support

Open Document Format for Office Applications Version 1.2 (29 September 2011)

Worksheet File Format (From Lotus) December 1984

Badges