Macros and Lua snippets

Helper macros mostly in Lua

Abstract
Described is a module containing a number of helper macros, many of them programmed in Lua.

Keywords
Lua, macro, file, font, list, date

Introduction

As already mentioned in a previous article in the MAPS, I am somewhat of a do-it-yourself'er. Either because the macro needed was not present, not on my radar or just because it seemed something nice to have and possibly useful in the future. These macros may be of use to others but it is up to them to decide that, of course. I just hope some might solve a problem here or there. They can be downloaded from my website at `https://www.hvandermeer.com`. From the homepage jump to the page *Publikaties and TeX modules* where they can be found as *ConTeXt module distribution*.

This module named `hvdm-ctx` is dependent on the companying module `hvdm-lua`. The latter contains most of the implementations of the caller macros in the former. The macros can be loosely categorized as file related macros and macros for the creation, maintenance and querying of Lua tables, as well as date and string manipulations plus some macros without much relation to each other.

Files in a Filevault

The Files group of macros is for manipulating file names, file content, combining files and probing the internet.

The idea for the *FileVault* macros arose from my use of XML for collecting in small notes the data found in various 17th and 18th century archives. This collection is nearing a thousand XML files with occasionally the addition of new ones or updated content. Some people will check the correctness of the XML before submitting their files, but that's not one of my habits. Therefore I decided to check them for validity in processing runs using the `XMLCheck` macro described below. But this implied reading them twice, once for the check and once for typesetting.

Why not read them once and keep the content as a string in Lua's memory? Modern memories have Gigabyte size and my thousand XML files are peanuts. I could have relied on the file caching of the OS, but it is not in my nature to forgo a chance to program something nice. Thus a small set of macros was developed to 'read once, use anywhere' so to speak. Use of this should be completely transparent, naturally, not needing special treatment. Switching between in memory content or rereading from disk to be done by simply turning it on and off. A bonus will be the possibility to run filters on the input before handing over to ConTEXt.

The above "not needing special treatment" could not be held on to entirely, pity. The culprit turned out to be the backslash. My notes can contain `<tex>` nodes, escapes to typesetting part of the text directly with ConTEXt. Typesetting files with for example `\loop ... \repeat` statements inevitable leads to a crash. In the underlying Lua code backslashes have a special meaning as escape string for control characters newline `\n`, `\t` or `\xxx` where xxx is a three digit number. TEX's use of the backslash doesn't fit in this scheme and things like `\def` fail miserably.
For possible solutions one can think of replacing `\` with its escaped equivalent `\\` but then they turned out to be neglected completely. Trickery with catcode changes is not encouraged in general and often extremely dangerous and disastrous.
Sad, but complete transparancy had to be relaxed a bit in order to cope with this situation. Its solution required the implementation of a macro that will leave the content of the file in an intermediate buffer in ConTEXt. Running code like

```
\xmlprocessbuffer{}{buffer-containing-file}{}
```

is then possible without causing a crash.

These are the functions implemented for the FileVault:

▷ `\FileVaultGetState`
`\FileVaultSetState[#1]` – handles the current state of the FileVault. The first macro returns the current state of the FileVault. In accordance with ConTEXt

custom this is `start` or `stop` whether the vault is active or inactive. Activation and deactivation is done with the second macro. The option values are `[state=start]` and `[state=stop]` respectively.
When the vault is in an inactive state all actions except these state macros and those reading files are suspended (see the example below for the latter). The latter will either return an error message or silently do nothing.
Examples: `\FileVaultGetState` shows the initial state of the vault as *stop*. After changing it with `\FileVaultSetState[state=start]` we can verify that the state is *start*.

▷ `\FileVaultFileString{#1}` – returns the file content as a string. If the file is nonexistent the macro silently returns an empty string.
Example: `\FileVaultFileString{example.txt}`

This is a very small example file.

What will happen if the vault is inactive? The same statement is executed after stopping the vault. The state is now *stop* and the same example gives

This is a very small example file.

Thus showing the transparency of the vault system for file reading.

▷ `\FileVaultFileBuffer[#1]{#2}` `\FileVaultFileDefaultBuffer{#1}` – returns the content of a file put into a ConTEXt-buffer. The parameter between braces `{}` specifies the file, the parameter between the option brackets `[]` must be the name of an existing buffer. The return of both macros will be the name of the buffer where the file content has been stored. With an explicite buffer name the contents of that buffer stays available, otherwise the internal buffer is reused each time. If the file is nonexistent the macro silently returns an empty buffer.
Example: `\typebuffer[...{otherexample.txt}]` results in

```
This is the other example file.
```

Since the underlying action is either file reading or retrieval, this action is transparent to the state of the filevault as for `\FileVaultFileString` has been shown above.

▷ `\FileVaultList` `\FileVaultListWithCount` – shows the content of the FileVault, the second caller adds the number of times each file has been accessed. Between the filenames in the list `\crlf`'s are added.
Example: After reading the second file twice again, the current access counts are:

example.txt 1
otherexample.txt 3

An inactive vault will return the following appropriate return:

ERROR inactive filevault

▷ `\FileVaultReturnTransform[#1]{#2}` – applies the transformation specified in `[#1]` on the file named in `#2` and delivers the value returned by that transformation. The first parameter of the user defined transformation function will receive the contents of the file as a Lua string. Since the data in the filevault are left intact, this macro will work transparently in either an active or inactive filevault.
The number of applications one can think of are manyfold. To name some: returning the length of the file, changing everything to uppercase, checking xml for correctness.
The argument `[#1]` must begin with the name of the Lua function performing the transformation. After a comma there may follow any number of comma separated arguments that are to be passed to the transformation function.[1]
To clarify this a complete example is presented of a function that will return the length of a file, either counted as bytes or counted as UTF8 characters. *Nota bene:* see how our Lua function is registered in the function registry of the filevault, because without this registration it will not be found. File *length.txt* contains

German ü and è in French.

```
\startluacode
-- Define our namespace as xmpl.
xmpl = xmpl or {}
-- Return the (utf8) length of a file.
xmpl.filelength = function (filedata, encoding)
  if encoding and encoding == "utf8" then
    return utf8.len(filedata)
  else
    return string.len(filedata)
  end
end
-- Register this function as "filelength".
hvdm.registerfunction
  ("filelength", xmpl.filelength)
\stopluacode
```

Input `[filelength]{length.txt}` to `\FileVaultReturnTransform` results in 28 bytes and `[filelength,utf8]{length.txt}` in 26 UTF8 characters.[2]

▷ `\FileVaultTransform[#1]{#2}` – nearly same as the previous macro but instead of leaving the data in the vault untouched, it will replace them by the result of the transformation. Because of this the macro is necessarily inoperative when the filevault is inactive and it will do so silently except for a

message in the log.
The return of this macro depends on the behaviour of the transformation function. Functions in Lua may return more than one value and this macro is programmed to accept one or two return values. The first value is used to replace the data in the filevault and the second value will be sent to the caller. In this manner it is possible to deliver a message, for instance to inform the caller of the cause that made an operation fail. Being more specific: an absent or nil second value returns nothing, otherwise that second value is returned. An example of its use might be the stripping of ignorable whitespace from xml: returning the stripped data twice will both change the data in the filevault as well as having it available for immediate processing.
Extending the previous example we will use this macro to replace the *length.txt* file by its length as UTF8 string. Note that the original file on disk will not be affected! Thus after `\FileVaultTransform[filelength,utf8]{length.txt}` calling `\FileVaultFileString{length.txt}` demonstrates that its value in the filevault has become

26

▷ `\FileVaultClear`
`\FileVaultClearFile[#1]` – respectively clears the FileVault completely or clears file `#1`; if that file is not in the vault nothing happens.
Example: after `\FileVaultClearFile[example.txt]` the vault contains:

length.txt *otherexample.txt*

and after `\FileVaultClear` we will receive an empty string "".

Information on Files
It sometimes is of interest to know if a file has a certain suffix. Perhaps in order to differentiate between files with the same basename but of different type: text, xml, images, pdf. Next follows macros used to query filenames and to retrieve the content of a directory.

▷ `\FileExist{#1}` – returns true or false on existence of the file `#1`.
Example: called on the source of this article naturally results in *true*.

▷ `\FileBaseName{#1}` – returns the filename stripped of prefixed directories.
Example: `./Documents/Letters/lastletter.doc` becomes *lastletter.doc*.

▷ `\FileDirectoryName{#1}` – returns the prefixed directories of the filename.
Example: `./Documents/Letters/lastletter.doc` becomes *./Documents/Letters/*.

▷ `\FileSuffix{#1}` – returns the suffix of the filename.
Example: `./Documents/Letters/lastletter.doc` returns *doc*.

▷ `\FileSuffixList{#1}{#2}` – returns the suffix of the filename if that suffix is in comma separated list `#2`.
Example: suffix list `{pdf,doc,txt}` returns *doc*, while `{pdf,txt}` will return an empty string.

▷ `\FileDirectoryList[#1]{#2}{#3}` – collects in a list all filenames in directory `#2` (empty is current directory) having suffix `#3` (empty suffix lists all files). The result is not directly returned but saved into a Lua table named `#1`. That list then can be queried by the table macros described below. With empty `#1` the name of the list is `FILEDIRECTORYLIST` by default. The information is added to the list if it already exists, except for the default list which is cleared before the operation.
Example: On this run there are *12 file(s)* in the current directory of which *5 tex-file(s)*.

Operations on Files
The purpose of the macros below is to execute file operations that otherwise would need intervening actions outside the ConTEXt run. The operations are concatenation of files, removal of files and querying for existence on the internet.

▷ `\FileConcatFile{#1}{#2}` – concatenates all files in directory `#1` having suffix `#2` and returns the concatenated contents. Before returning the intermediate temporary is removed.

▷ `\FileConcatName{#1}{#2}` – nearly the same as `\FileConcatFile` but here the temporary is not removed and its name is returned to the caller.

▷ `\FileConcatFilePrePost{#1}{#2}{#3}{#4}` – same as `\FileConcatFile` but `#3` precedes the concatenated contents while `#4` is affixed to the end.

▷ `\FileConcatNamePrePost{#1}{#2}{#3}{#4}` – same as `\FileConcatFilePrePost` but returning the filename instead of the content.

▷ `\FileRemove[#1]` – remove the file by name.

▷ `\URIReturnCode{#1}` – reaches into the internet with a `socket.http.request` to check if file given can be accessed. Returns the standard return code 200 on success and 404 for file not found. Waits 5 seconds for reaction from the internet before giving up.

Care is taken to change spaces in the URL to the mandatory %20.

Creation of Lua Lists
Collection of macros to produce and manipulate Lua tables. The tables or lists as named in the macros, are kept inside the module in an invisible table functioning as the holder of those created by the caller.
The macros in this section arose from the need to collect various information on the fly, storing it and present it afterwards in various forms.

Where lists are involved their name is always in `#1`.

List creation and removal

▷ `ListCreate[#1]` – creates a named list. That name is used in later accesses to the list. The list is created empty. Beware: reusing a list is silently inhibited, first delete an existing list before reusing the name.

▷ `\ListExist[#1]` – queries the existence of list `#1` and returns `true` or `false` accordingly.
Example: `\ListCreate[test]` then `\ListExist[test]` returns *true*.

▷ `\ListDelete[#1]` – removes the list by setting the reference to the list `nil`.
Example: given the above ListCreate the sequence `\ListExist[test]`, `\Delete{test}`, `\ListExist[test]` returns first *true* and then *false*.

▷ `\ListClear[#1]` – removes the content of the list, leaving it empty.

▷ `\ListCount[#1]` – returns the number of elements in the list. Lua tables have both an array and a key based section. The count is done over both these sections.
Example: empty list should return zero `\ListCreate[zero]` then `\ListCount[zero]` returns 0.

Addition of List Elements

▷ `\ListAdd[#1]{#2}` – adds `#2` as next element in the array section of the Lua table. Successive additions of the same element become successive elements in the list.
Example: add string "one" to the array section of list zero created above, the reported element count is 0 before and 1 after.

▷ `\ListAddKey[#1]{#2}{#3}` – adds element with key `#2` and value `#3` to the key section of the Lua table. Successive additions with the same key silently overwrite the previous value.
Example: add string "two" with key "second" to the same list with `\ListAddKey[zero]{second}{two}` and observe the incremented element count 2.

▷ `\ListAddSubKey[#1]{#2}{#3}{#4}` – adds element with key `#3` and value `#4` to a sublist keyed by `#2`, creating that sublist if necessary.

▷ `\ListAddTo[#1]{#2}` – adds element with key `#2` to the list setting its count to 1. Successive additions at the same key increment the counter value. The list therefore keeps a count of how often that key has been added.

▷ `\ListAddToKey[#1]{#2}{#3}` – adds to the array section of the Lua table a subtable `{#2,#3}` as a key-value pair.

Retrievial of List Elements

▷ `\ListArrayValue[#1]{#2}` – returns the element with index `#2`. If the index is outside the range of stored elements an empty string is returned.
Example: retrieve the first element in the array section of list zero `\ListArrayValue[zero]{1}` is *one*.

▷ `\ListKeyValue[#1]{#2}` – returns the value of the element at key `#2`. If the element is not present then an empty string is returned.
Example: retrieve the element with key "second" from list zero `\ListKeyValue[zero]{second}` is *two*.

▷ `\ListKeyValueWithDefault[#1]{#2}{#3}` – same as `\ListKeyValue` but instead of returning an empty string for an absent element, `#4` is returned as default instead.
Example: an element with key "third" has not been added thus `\ListKeyValue[zero]{third}` returns "", an empty string. The next call shows the return of a default `\ListKeyValueWithDefault[zero]{third}{three}` results in *three*.

▷ `\ListSubKeyValue[#1]{#2}{#3}` – returns element `#3` from sublist `#2`, empty string if absent.

▷ `\ListSubKeyValueWithDefault[#1]{#2}{#3}{#4}` – returns element `#3` from sublist `#2`, default `#4` if absent.

▷ `\ListValueKey[#1]{#2}` – find and return from the list the first key having `#2` as its value.

▷ `\ListValueSubKey[#1]{#2}{#3}` – find and return from the list the first key in sublist `#2` having `#3` as its value.

▷ `\ListValueSubKeyAll[#1]{#2}` – find and return from the list and all of its sublists the first key having `#2` as its value.

▷ `ListPrint[#1]` – simple printer for the contents of list `#1`.
The underlying Lua tables harbour two sections: (1) an array section indexed upwards from 1 (by default), and (2) a section with key-value pairs.

Both sections are printed unsorted, just as the entries are encountered by table traversal. Line endings are set to `\crlf` to accommodate ConTEXt. Example: a list has been made with two items in both the array and the key-value section.
`\ListPrint[Test]` prints:
1 = array item 2 added first
2 = array item 1 added last
1 = array item 2 added first
2 = array item 1 added last
secondkey = value two
firstkey = value one

XML-related operations

The purpose of next macros is the production of content to be handled by an XML processor. They provide for the contruction of (embedded) nodes, sorting lists of nodes, processing and checking of XML from various sources.

▷ `XMLContentToNode{#1}{#2}` – returns the string `<node>content</node>` where node = `#1` and content is `#2`. Convenient when a list must be filled with XML nodes to be processed later.

▷ `\ListToNodes[#1]{#2}`
`\ListToNodesSorted[#1]{#2}{#3}` – returns the content of the key section of Lua table `#1` as concatenated pairs <name><key>thekey</key><value>the-value</value>...</name> where `name` is `#1`, the name of the list. In the second macro the nodes are sorted to the keys with `#3` is `normal` for ascending (default) and `reverse` for descending keys.[3] This macro is meant for lists filled with `\ListAdd`.

▷ `\ListValuesToNodes[#1]{#2}`
`\ListValuesToNodesSorted[#1]{#2}{#3}` – same as `\ListToNodes` except here the sorting is for lists filled with `\ListAddKey` where the elements are tables containing a key-value pair.

▷ `\XMLProcessBuffer{#1}` – processes buffer `#2` containing a valid XML file with macro call `\xmlprocessbuffer{id}{#2}{}`. The `id` is filled by the called function.

▷ `\XMLProcessFile{#1}` – same as above with file `#1`. As a bonus the macro discerns the presence of embedded TEX and switches processing as described in the section on the FileVault.

▷ `\XMLProcessFolder{#1}` – same as above for all files in directory `#1` having the `xml` extension.

▷ `\XMLProcessString{#1}` – same as above with the content of argument `#1`.

▷ `\XMLEntitiesRead[#1]` – register entity declarations for XML processing from a dtd file `#1` with the ConTEXt procedure `xml.registerentity()`.

▷ `\XMLCheck[#1]{#2}` – checks the validity of the XML tree `#2` located in `#1` being `file`, `folder`, `buffer` or `string`. If the XML is correct an empty string is returned otherwise the string contains the nodes remaining after removal of the correct nodes. Example: `<root><node att="abc">error</root>` is missing the xml-header (not considered a problem) and a closing `</node>`. Checking results in
`>>> <root><node></root>`
which should be helpful in the repair. The check works by deleting correct nodes, starting within and working outwards. At the end of the reduction correct XML leaves nothing but an empty string, otherwise something is amiss as can be seen in the example.

Date and Time formatting

Dates can be given in the formats yyyy-mm-dd, yymmdd (20th century only), yyyymmdd, dd-mm-yyyy, d-m-yyyy, dd-m-yyyy, d-mm-yyyy. Negative dates or dates with BC get a negative year. Use yyyy for year only and yyyy-yyyy for a year range. Dates are checked for validity. Dates containing other characters than digits, dashes and possible BC are taken as is and considered dates already in final format. The formatters return the string `"DATE ERROR"` when something is amiss.

▷ `\DateCheck{#1}` – returns string `"true"` if a valid date has been found, `"false"` otherwise. A range of years yyyy-yyyy and a date already formatted return `"date"`.
Example: `\DateCheck{1-1-1900}` is *true* and for `13-13-1313` and `30-2-2020` it is *false* and *false*.

▷ `\DateFormat[options]{#1}` – formats the date in European format dd-mm-yyyy where days and months less than 10 are typeset with one digit only. There are both single and key=value options. The formatting options are `zero` (zero fill), `short` (abbreviated month name), `long` (full month name), `julian` (Julian date), `text` (no formatting), default is a compact format. A language option switches the translation (see the example below) default is the value of `\currentlanguage`.
Examples:
`\DateFormat[]{200201}` = *1-2-2020*
`\DateFormat[zero]{200201}` = *01-02-2020*
`\DateFormat[short]{200201}` = *1 Feb 2020*
`\DateFormat[long]{200201}` = *1 February 2020*
`\..[long,language=fr]{200201}` = *1 février 2020*
`\DateFormat[julian]{200201}` = *2458881*
`\DateFormat[text]{200201}` = *200201*

▷ `\DateCurrent` – the date of today *19-3-2021* formatted

with `\DateFormat`. Also one can obtain *19 March 2021* from `\DateFormat[long]{\DateCurrent}`.

▷ `\DateJulian{#1}` – same as `DateFormat[julian]{#1}`. The date converted to a Julian date with range limited from 4713 BC to 3628 AD. Today is *2459293* in Julian. The Julian date is useful when things have to be sorted on date.

▷ `\TimeFormatMinutes{#1}`
`\TimeFormatSeconds{#1}` — format time duration given as minutes or seconds respectively, into hh:mm:ss. Input with one or more :'s or otherwise not a number is considered formatted.
Examples:
`\TimeFormatMinutes{59}` = *00:59:00*
`\TimeFormatMinutes{120}` = *02:00:00*
`\TimeFormatSeconds{10}` = *00:00:10*
`\TimeFormatSeconds{3599}` = *00:59:59*
`\TimeFormatSeconds{3600}` = *01:00:00*

▷ `\TimeCurrent` – the time from the current clock as of the moment of typesetting is 10:04:00. Note that the underlying macro `\the\normaltime` returns the time in minutes since mdidnight.

Case Change and Character Selection

Changing case is notoriously difficult, at least in my experience. I always had trouble with `\uppercase` and friends. ConTEXt provides macros `\Word`, etc. but doing it yourself is a challenge I am not always able to resist. The implementation of CamelCase transformations came as a bonus.

▷ `\ChangeLower{#1}` – changes all letters in `#1` to lower case. For instance the french Était becomes était.

▷ `\ChangeUpper{#1}` – changes first letter to upper case, était thus becomes Était.

▷ `\ChangeUPPER{#1}` – changes all letters in `#1` to upper case, était becomes ÉTAIT.

▷ `\ChangeCamel{#1}` – changes all letters following whitespace in `#1` to upper case.
Example: *sample text word-combination* becomes: *Sample Text Word-combination*

▷ `\ChangeCamelPlus{#1}{#2}` – changes to uppercase in `#1` all letters following whitespace and those from `#2`. Beware: the - for example is special in Lua search patterns and therefore must be preceded by a %. Note that `\letterpercent` is how to insert it.
The same example with the - added:
Sample Text Word-Combination

▷ `\TypeCheck{#1}` returns the Lua type of `#1`. Useful to test if `#1` is a string that Lua can convert into a number.
Example: "5" has type *number* and "a5b" type *string*.

The macros below are intended to extract characters and truncate strings to substrings. For example in operating systems long filenames are sometimes truncated by removing parts in the middle.

▷ `\StringFirstCharacters{#1}{#2}` – truncates string `#1` to a length of `#2` by removing the excess characters at the end.
Example: string has more than 40 characters *This was a very long string more than th....*

▷ `\StringLastCharacters{#1}{#2}` – truncates string `#1` to a length of `#2` by removing the excess characters at the front.
Example: string has more than 40 characters *...ong string more than the available space.*

▷ `\StringFirstLastCharacters{#1}{#2}` – truncates string `#1` to a length of `#2` by removing the excess characters in the middle.
Example: string has more than 40 characters *This was a very lo...he available space.*

▷ `\CharacterSelect[#1]{#2}{#3}` – returns from string `#2` the `#3`-th character or an empty string if that character is not found. The option `#1` is a selector from `alphanum`, `alpha`, `digit`, `punct`. A recognized option returns the n-th character from the corresponding set, while an empty selector returns just the n-th character. The function does not know about UTF8 characters when a selector is given.
Example: `\CharacterSelect[]{the 7 dwarfs.}{7}` = d
Example: `\CharacterSelect[alphanum]{..}{7}` = a
Example: `\CharacterSelect[alpha]{..}{7}` = r
Example: `\CharacterSelect[digit]{..}{1}` = 7
Example: `\CharacterSelect[punct]{..}{1}` = .
Example: `\CharacterSelect[]{pygmée}{5}` = é

Numbers and Number Series

▷ `\RandomSeed{#1}` – initializes the Lua random generator with seed `#1`.

▷ `\RandomValue` – returns next random value from the Lua random generator.
Example: 0.85193537224893.

▷ `\RandomRange{#1}` – returns a random value within range 1-`#1` from the Lua random generator.

▷ `\Series[options]{#1}` – returns a series of `#1` numbers. The direction of the values can be ascending (default) or descending, the corresponding key options being `normal` (or empty) and `reverse`.

The start value and the stepsize follow from `start=number` and `step=number`, both having default 1. The blank separator between the list items can be changed with the option `separator=value` or `separator={value}`.[4] See the last example below where the default space separator is replaced by space + space.
Examples:
`\Series[]{10}` *1 2 3 4 5 6 7 8 9 10*
`\Series[reverse]{10}` *10 9 8 7 6 5 4 3 2 1*
`\Series[reverse,start=0]{10}` *9 8 7 6 5 4 3 2 1 0*
`\Series[start=0,step=-0.5]{4}` *0 -0.5 -1.0 -1.5*
`\Series[separator={ + }]{4}` *1 + 2 + 3 + 4*

A more elaborate example is the following:[5]

```
\leavevmode
\setupframed[extras=\space,width=5mm,height=5mm]
\ProcessCommaList{framed[framecolor=blue]}
    {\Series[separator={,}]{5}}}
```

1 2 3 4 5

A few remarks are in order. Macro `\ProcessCommaList` is a wrapper around `\processcommalist` which would otherwise crash as used here.[6]

▷ `\RandomSeries[#1]{#2}` – returns `#2` numbers randomly from the range 0 to 1. With option `#1` is `range=integer_number` the values are integers drawn from the interval 1..range. The option `#1` will receive an item separator as in the above example.
Example: `\RandomSeries[range=10]{4}` *4, 7, 2, 8*

▷ `\MDfive{#1}` – converts the string `#1` into MD5 hash value. Although nowadays not strong enough for a secure hash, it is sufficient to fingerprint (long) strings. Stored in a list useful to detect if these strings were encountered before.
Example: `\MDfive{[[ MD5 ]]}` is *6b3663a615846322674e3abf4fd59672*

▷ `\SafeNumber{#1}{#2}` – Return `#1` if the Lua function tonumber succeeds, otherwise return `#2` as default.
Example: strings 207 and 207a with default NAN, return respectively *207* and *NAN*.

Availability

The module and its supporting modules can be downloaded from my site `hvandermeer.com/publications.html`. The TEX-stuff resides in section "Articles on TeX", downloads are a little below in the link "ConTeXt module distribution".

Notes

1. Courtesy of the fact that Lua functions can accept a variable number of parameters.
2. For those who would expect 27 and 25 as answers: the newline at the end of the line is included in the count.
3. The tablesorter is derived from *Programming in Lua* by Roberto Ierusalimschy, 3rd edition, section 20.2 page 197.
4. The braces are mandatory in case certain characters are present in the option value, especially spaces, commas and Lua special pattern matching characters. Without the braces the underlying Lua function does not treat the option value as intended. The Lua special characters are `^$()%.[]*+-?`
5. The `\leavevmode` is needed to suppress the newlines that otherwise appear between the frames.
6. All parameters to the `\framed` could have been placed inside the `[]`'s, but it would clutter this presentation too much.

Hans van der Meer
havdmeer@ziggo.nl