SAS® GLOBAL FORUM 2020 #SASGF

Paper 4725-2020

# SAS® Packages - The Way To Share (a How To)

Bartosz Jabłoński, Warsaw University of Technology / Citibank Europe PLC Poland

## ABSTRACT

When working on Base SAS® code, especially when it becomes more and more complex, there is a point in time when a developer decides to break it into small pieces. The developer creates separate files for macros, formats/informats, and for functions or data too. Eventually the code is ready and tested and it is time for the deployment. The issue is that the code had been written on a local Windows machine and the deployment is on a remote Linux server. Folders[1] and files have to be created with the proper structure, code has to be run in the right order and not mixed up. Moreover it is not the developer who is deploying... Small challenge, isn't it?

How nice it would be to have it all (i.e. the code and its structure) wrapped up in a single file - a portable SAS package - which could be copied and deployed with a one-liner like: `%loadPackage(MyPackage)`?

In this article an idea of how to create such a SAS package in a fast and convenient way will be proposed/shared. We will discuss:

- a concept of how to build a package,
- the tools required to do so (framework), and
- a "how to" of the process (i.e. generating packages, loading, and using them).

The intended readers for the following document are intermediate SAS users (i.e. with good knowledge of Base SAS and practice in macro programming, see [**1**]) who want to learn how to share their code with others.

## INTRODUCTION and CONTEXT

In the world of programmers, software developers, and "computer people" the concept of a package is well known and common. To give an evidence of this statement let us consider four very popular examples: Linux, Python, TEX, and R software, and as an endorsement the following quotes.

According to [**6**]:

> "In Linux distributions, a *package* refers to a compressed file archive containing all of the files that come with a particular application. [...] Most packages also contain installation instructions for the OS, as well as a list of any other packages that are dependencies (prerequisites required for installation).
>
> Common types of Linux packages include `.deb`, `.rpm`, and `.tgz`. Since Linux packages do not usually contain the dependencies necessary to install them, many Linux distributions use package managers that automatically read dependencies files and download the packages needed before proceeding with the installation."

According to [**5**]:

> "[In Python] modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. Individual modules can then be cobbled together like building blocks to create a larger application. Packages allow for a hierarchical structuring of the module [...]."

[1] Folders - also known as Directories

According to [**4**]:

> "Many LaTeX commands [...] are not specific to a single class but can be used with several classes. A collection of such commands is called a package and you inform LaTeX about your use of certain packages in the document by placing one or more `\usepackage` commands after `\documentclass`.
>
> Just like the `\documentclass` declaration, `\usepackage` has a mandatory argument consisting of the name of the package and an optional argument that can contain a list of package options that modify the behaviour of the package."

As of January 2020, there were over 5,700 packages available on the Comprehensive TeX Archive Network! More than a dozen of TeX packages were used while writing this article.

According to [**3**]:

> "In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests, and is easy to share with others. As of January 2015, there were over 6,000 packages available on the Comprehensive R Archive Network, or CRAN, the public clearing house for R packages. This huge *variety of packages is one of the reasons that R is so successful*: the chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading their package."

As of January 2020, there were over 15,000 packages available on the CRAN!

The SAS/IML offers (limited) version of functionality similar to concept of a package in programming languages and environments mentioned above but such functionality is not available in the Base SAS. The main goal of this article is to propose, describe, explain, and discuss an original idea of a process and tools required to build SAS packages. In subsequent sections we introduce the concept of a SAS package from both user and developer point of view. What is worth to mention, and what is one of the biggest advantages of using SAS packages, is that the work to be done on the user's side to use a package is almost none. The last section provides an example of package creation.

## WHAT IS a SAS PACKAGE?

A **SAS package** [2] is an automatically generated, single, stand alone `zip` file containing organised and ordered code structures, created by the developer and extended with additional automatically generated "driving" files (i.e. description, metadata, load, unload, and help files).

The purpose of a package is to be a simple, and easy to access, code sharing medium, which will allow: on the one hand, to separate the code complex dependencies created by the developer from the user experience with the final product and, on the other hand, reduce developer's and user's unnecessary frustration related to a remote deployment process.

In this article we are presenting a *standalone Base SAS framework* which allows to develop and use SAS packages.

To create a package the developer must prepare the code files and a description file, fit them into a structured form (see next sections for details), download `%generatePackage.sas` file and execute the `%generatePackage()` macro (see THE CODE subsection for details on how to obtain it).

To use a package the user should download the package `zip` file into the packages folder (containing downloaded `loadpackage.sas` file). And, in the SAS session, the user should run the following code:

```
filename packages "<directory/containing/packages>";
%include packages(loadpackage.sas);
%loadPackage(packageName)
```

to have the package available.

[2] The idea presented in this article should not be confused with other occurrences of "package" concept which could be found in the SAS ecosystem, e.g. Proc DS2 packages, SAS/IML packages, SAS ODS packages, or SAS Integration Technologies Publishing Framework packages.

## THE USER: HOW TO and THE RULES

**User's files and folders.** Since the idea of a SAS package is to take off (from the user's shoulders) the burden of "necessity to know how it is all connected and dependent" there are only a few simple steps to be done on the user's end. The user's part of work required to use a package starts with setting up some files and folders, but is very short and in practice only the last step is repeated more than once. The work goes as follows:

- Create a folder for your packages, e.g. under Windows OS family `C:/SAS_PACKAGES` or under Linux/UNIX OS family `/home/<username>/SAS_PACKAGES`.
- Download the `loadpackage.sas` file into the packages folder (see THE CODE subsection).
- Download the package `zip` file into the packages folder.

**User's session.** When all files and folders are ready the user, to enjoy the package in a SAS session, executes the following steps.

- To set up the SAS session for using packages:
  ```
  filename packages "<directory/containing/packages/>";
  %include packages(loadpackage.sas);
  ```
- To load the package:
  ```
  %loadPackage(packageName)
  ```
- To get help about the package printed in the log:
  - for general information about the package:
    ```
    %helpPackage(packageName)
    ```
  - for all available information about the package:
    ```
    %helpPackage(packageName,*)
    ```
  - for a particular element of the package, e.g. a function or a macro:
    ```
    %helpPackage(packageName, helpKeyword)
    ```
    where *helpKeyword* is a single word which is used for context search. "License" prints out license text.
- For removing (a.k.a. unloading) the package content:
  ```
  %unloadPackage(packageName)
  ```

For a detailed list of macros parameters see the Appendix B.

After loading a package for the first time it is a *good practice* to read the log to find out more about package content and the list of loaded elements.

**Note.** The code executed behind the scenes is designed to affect the user session environment in an "as small as possible" way. The required minimum is compilation of the `%loadPackage()`, `%unloadPackage()`, and `%helpPackage()` macros, and setting a global macrovariable `SYSloadedPackages`. When the `%loadPackage()` macro is executed for the first time the `SYSloadedPackages` macrovariable is created and its value is updated with package name and version for each new loaded package.

**Caution!** There is one important *restriction* regarding the SAS session! Words "`package`" and "`packages`" are restricted as a file reference for the `FILENAME` statement and the `FILENAME()` function. These words are file references used internally by the `%loadPackage()`, `%helpPackage()`, and `%unloadPackage()` macros. Using them may cause unexpected results and may jeopardise package stability!

**User's "under the hood".** The above steps are all that is necessary to use and work with a package. There are also some additional things happening in the background. This section explains them in more details.

- The `%loadPackage()` macro loads all components of the package as a primary job. Additionally information from the package description file is printed into the log. Whenever an element of the package is loaded an appropriate note is printed into the log. If there were any requirements provided they

will be tested at this point. If required element of the SAS system is missing the loading process is aborted with an error message. If required package is not loaded (`SYSloadedPackages` macrovariable is tested) SAS tries to load that required package, in case of failure loading process is aborted with an error message. If the loading process is successful the `SYSloadedPackages` macrovariable is updated with new entry. If a particular version of a package is required then the `%loadPackage()` macro has to be executed with additional parameter `requiredVersion`, e.g. `%loadPackage(PiPackage, requiredVersion=3.1415)`. The macro tests if the provided version is greater o equal then the required version, in case of failure loading is aborted with an error message (this condition *implicite* assumes that packages are backward compatible). The package may contain datasets as so called "lazy loading datasets". Such datasets are not automatically loaded when the `%loadPackage()` macro is executed. To load such dataset the user has to call the `%loadPackage()` macro with non missing `lazyData=` parameter, e.g `%loadPackage(PiPackage, lazyData=work.first1E6digits)` (list of multiple elements separated by space is allowed, an asterisk(*) means "load all data").

- The `%helpPackage()` macro prints out into the log help information attached to the package content. The `%helpPackage()` macro is independent from the `%loadPackage()` macro what means that it can be executed even if the `%loadPackage()` was not executed. The user can read about a package before loading it! When no second argument is provided only the package description, list of package elements, list of required components, and version of the `%generatePackage()` macro used are printed out. When the second argument is provided, if it is an asterisk ("`*`") all help content is printed out (for datasets also `proc contents` is run), if it is a `helpKeyword` then content search is executed based on its value and only selected parts of help content are printed out. If `helpKeyword` value is "License" then package license is printed out.
- The `%unloadPackage()` macro cleans up the session. All objects created by the package (except `execs`) are deleted. If `clean` files were provided their content is executed too. If the unloading process is successful the `SYSloadedPackages` macrovariable is updated. If a package required additional packages to be loaded then during unload a list of calls to `%unloadPackage()` for this additional packages is printed in the log, to allow their manual execution.

**User's "in case of emergency".** This section covers "emergency" situations that user may come across when for the first time loads a package.

The first one is *backward compatibility*. The package file is a `zip` file so at least SAS 9.4 is required to use it. But it is also possible to use earlier versions of SAS software if it is needed. When the SAS session does not support ZIP fileref the following solution could be used: unzip the `packagename.zip` file content into a `packagename.disk` folder and run loading, helping, and unloading macros with following options:

```
%loadPackage(packageName, zip=disk, options=)
%helpPackage(packageName, helpKeyword, zip=disk, options=)
%unloadPackage(packageName, zip=disk, options=)
```

The second one is *no access to the `loadpackage.sas` file*. When user has only the `packagename.zip` file with the package but does not have an access to the `loadpackage.sas` file with the `%loadPackage()` macro the following solution could be used: each package generated by `%generatePackage()` macro contains `%ICEloadPackage()` macro, which is a simplified version of the `%loadPackage()` macro, and can be used for an "emergency" load. To use it: a) make file reference to the package zip file, b) include `iceloadpackage.sas`, c) make file reference to the package folder, and d) load package.

```
filename packages ZIP "<directory/containing/packages>/packagename.zip";
%include packages(iceloadpackage.sas);
filename packages "<directory/containing/packages>/";
%ICEloadPackage(packageName)
```

## THE DEVELOPER: HOW TO and THE RULES

**Developer's files and folders.** The developer's part of work to build a package starts with preparing a set of files and folders. This part goes as follows:

- Create a folder for your package a.k.a. package folder (hint: name it the same as your package).
- Create a description file, named `description.sas`, and copy it into the package folder. The file is mandatory, has simple structure, and it contains package metadata and (short) description (compare [**3**]). The simple structure of the `description.sas` file can be seen in Figure 1.

```
/* This is the description file for the package.          */
/* The colon (:) is a field separator and is restricted   */
/* in lines of the header part.                           */

/* **HEADER** */ ❶
Type: Package                                              : ❷
Package: ThePackageName                                    : ❸
Title: Short description, single sentence.                 : ❹
Version: x.y                                               : ❺
Author: Fname1 Lname1 (xxx1@yyy.zz), Fname2 Lname2 (xxx2@yyy.zz) : ❻
Maintainer: Fname3 Lname3 (xxx3@yyy.zz)                    : ❼
License: XYZ17                                             : ❽
Encoding: UTF8                                             : ❾

Required: "Base SAS Software", "SAS/Xxx", "SAS/ACCESS Interface to Yyyy" : ①
ReqPackages: "somePackage (3.14)", "otherPackage (42)"                   : ②

/* **DESCRIPTION** */ ❿
/* All the text below will be used in help */
DESCRIPTION START:

Lorem ipsum dolor sit amet, ThePackageName consec tetur
adipis cingelit. Nullamdapibus lacus a elit congue
elementum. Suspendisse iaculis ipsum nec ante luctus
volutpat. Donec iaculis laoreet tristique.

DESCRIPTION END:
```

**Figure 1: Package description structure.**

The meaning of entries (a.k.a. tags) inside the `description.sas` file are the following (the dark bullet marks an element which is *mandatory*):

(❶) `/* **HEADER** */` - marks the header start. Mind the structure of the tag, i.e. slash, asterisk, *space*, and double asterisk (`/* **`). Each of the following lines is a `key:value` pair and such a pair must be a *single line of text*. The colon (`:`) is a field separator and is restricted in lines of the header.

(❷) `Type` - is a constant (i.e. "`Package`"), required, and not null value.

(❸) `Package` - is the package name. It is required, not null, up to 24 characters long, and shares naming restrictions like those for a SAS dataset name.

(❹) `Title` - is the short title of a package (i.e. one phrase). It is required and not null.

(❺) `Version` - is the package version, it is required, not null, and a positive number (i.e. $> 0$). The preferred form is: an integer value for a stable version, a decimal value for a non-stable one. Since the `%loadPackage()` macro contains a dedicated macrovariable `requiredVersion` to tests if the provided version of a package is *greater or equal* then the required version - a very important,

in line with the "SAS way", assumption must be highlighted! The assumption is: *packages are assumed to be backward compatible*.

(❻) and (❼) `Author, Maintainer` - are comma separated lists of package author(s) and maintainer(s). Elements of lists are of the form: "`Firstname Lastname (email@address.com)`".

(❽) `License` - is the license under which the package is distributed. It is required, not null, possible values are: MIT, GPL2, BSD, PROPRIETARY, etc. The license text itself should be inserted into the `license.sas` file (see further steps).

(❾) `Encoding` - is the information about SAS sessions encoding the package files were created in. It is required and not null. Possible values are: UTF8, WLATIN1, LATIN2, etc. and the values should satisfy requirements for the `encoding=` option of the `filename` statement.

(❿) `/* **DESCRIPTION** */` - is the last required part. It is the package description. It is a free text bounded between the "`DESCRIPTION START:`" and the "`DESCRIPTION END:`" tags. It could be multi-line. It should elaborate about the package and its components (e.g. macros, functions, datasets, etc.) Text outside the tags is ignored.

(①) `Required` - is a quoted and comma separated list of licensed SAS products required for the SAS session under which the package will be used. Possible values inserted into the list should be the same as these the `proc setinit` prints in the log, e.g. `"Base SAS Software", "SAS/IML", "SAS/ACCESS Interface to Teradata"`. The "`Required`" tag is optional, when it is empty or not provided in the description the testing code is not generated. Though, it is recommended to add this one.

(②) `ReqPackages` - is a quoted and comma separated list of names and versions (in parentheses) of other SAS packages required for the package to work. Possible values inserted into the list should be formatted like e.g. `"SQLinDS (0.1)"`. The "`ReqPackages`" tag is optional, when it is empty or not provided in the description the testing code is not generated.

Based on the header information, the following internal macrovariables are generated: `packageName`, `packageVersion`, `packageTitle`, `packageAuthor`, `packageMaintainer`, `packageEncoding`, `packageLicense`.

- Inside the package folder create subfolders for the code files. A subfolder name has to be structured as follows:
  a) it contains only lower case letters, digits, and underscore ("_")
  b) it is composed of two parts separated by an underscore ("_"), i.e.
     - the first part is a series of digits (with leading zeros, e.g. `001`, `002`, ..., `123`, `124`, ...); its purpose is to keep execution sequence in case the code must be ordered to run properly;
     - the second part, called folder *type*, indicates subfolder content. The *type* has to be one of the following:
       `libname` (for libraries assignments),
       `macro` (for macros),
       `function` (for `proc fcmp` functions),
       `format` (for formats and informats),
       `data` (for the code generating datasets),
       `lazydata` (for the code generating datasets which will be loaded on demand, so called "lazy loading datasets"),
       `exec` (for so called "free code"),
       `clean` (for the code cleaning up the session after `exec`s) or
       `test` (for developer code with package tests).

  An example of a package subfolders structure can be found in Figure 2. In case the order of code execution is irrelevant the first part (i.e. digits and underscore) may be skipped.

  In case when the order of code execution is important, e.g. format `$efg.` must be defined before function `abc()`, two folders of *type* `format` and `function` with two different sequences of digits have

to be created in a way that digits indicate execution order, e.g. `017_format` for the code of the format `$efg.` and `042_function` for the code of the function `abc()`.

**Note.** The list of types may be extended in the future if need be.

- Copy the files with the code into package subfolders in accordance with *types* and the following set of rules:
  - One-file-one-object, e.g. macro `%abc()` definition has to be contained in a single file without any definitions of other objects. The only exception are formats/informats, in this case one file has to contain all definitions of formats/informats sharing the same name, e.g. numeric format `abc.`, character format `$abc.`, numeric informat `abc.`, and character informat `$abc.` all have to be kept in one file.
  - An object name is a file name, e.g. a definition of a macro named `%abc()` has to be contained in a file named `abc.sas`.
  - A definition of a function has to be enclosed in the following template of the `FCMP` procedure:

```
proc fcmp
  inlib  = work.&packageName.fcmp /* optional */
  outlib = work.&packageName.fcmp.package
  <... other options ...>
;
    <... function or subroutine body ...>
run;
quit;
```

    The `inlib=` and `outlib=` options are, literally, set to: "`work.&packageName.fcmp`" and "`work.&packageName.fcmp.package`" respectively. In case when functions from other packages are required to be used the `inlib=` option may be extended e.g. `inlib=(work.&packageName.fcmp work.OnePackagefcmp work.SecondPackagefcmp)`.
  - A definition of a format/informat has to be enclosed in the following template of the `FORMAT` procedure:

```
proc format
  lib = work.&packageName.format
  <... other options ...>
;
  <... numeric format definition ...>
  <... character format definition ...>
  <... numeric informat definition ...>
  <... character informat definition ...>
run;
```

    The `lib=` option is, literally, set to: "`work.&packageName.format`".
  - `exec` folders are for so-called "free code", i.e. if a package, to be ready and usable, requires some additional code to be run (code not fitting provided *types*) - this code has to be inserted into a file inside one of the `exec` subfolders.
  - `clean` folders are for cleaning after `exec`s, i.e. if a code from one of `exec` folders creates some object (e.g. a catalog, a macro, or a dataset) the appropriate code inside a `clean` subfolder has to be developed to remove that created object.
  - Parts of code files which are to be used to generate help information must be enclosed between following text tags: "`/*** HELP START ***/`" and "`/*** HELP END ***/`". The tags are not mandatory but if they are missing in a file a warning is printed into the log during package generation. If help tags are present in a file they have to be in proper order (start - end) and cannot be nested or overlap. If such situation takes place an error is printed into the log during

package generation and the process is aborted. An example could be the following file containing a macro code, a help text, and other comment:

```
/*** HELP START ***/
/* >>> %ABC() macro: <<<
 *
 * Main macro which allows to do
 * this and that...
 * Recommended for SAS 9.4 and higher.
**/
/*** HELP END ***/

/* macro definition */
/*** HELP START ***/
%MACRO ABC(
  param1 /* parameter 1 is used for ... */
 ,param2 /* parameter 2 is used for ... */
);
/*** HELP END ***/
    <... body of a macro ...>
    <...                 ...>
    <... body of a macro ...>
%MEND ABC;

/*** HELP START ***/
/* EXAMPLE 1: use in datastep

   data class;
     set sashelp.class;
     %ABC(age, weight)
   run;

**/
/*** HELP END ***/
```

only the following parts of text will be extracted for help purpose:

```
/* >>> %ABC() macro: <<<
 *
 * Main macro which allows to do
 * this and that...
 * Recommended for SAS 9.4 and higher.
 *
**/
%MACRO ABC(
  param1 /* parameter 1 is used for ... */
 ,param2 /* parameter 2 is used for ... */
);
/* EXAMPLE 1: use in datastep

   data class;
     set sashelp.class;
     %ABC(age, weight)
   run;

**/
```

- Create a `license.sas` file containing license information for the package. Place the file in the package folder (together with the `description.sas` and subfolders). If no file is provided the `license.sas` will

be generated with standard MIT license (read "GENERATING PACKAGE IN PRACTICE - a USECASE" section and Appendix A to see the MIT license text).

- Create a folder for packages, e.g. under Windows OS family `C:/SAS_PACKAGES` or under Linux/UNIX OS family `/home/<username>/SAS_PACKAGES` and copy the `generatepackage.sas` and `loadpackage.sas` files into this folder.

**Developer's session.** When all files and folders are ready the developer runs SAS session and executes the following code:

```
filename packages "<directory/containing/packages/>";
%include packages(generatepackage.sas);
/*ods html;*/
%generatePackage(filesLocation=<directory/with/package/files/>)
```

When the `%generatePackage` macro ends its execution the `packagename.zip` file, containing all package content inside it, is created inside the "`<directory/with/package/files/>`".

**Developer's "under the hood".** Before reading this subsection further we highly recommend (for a better view) to have subsections "User's files and folders" and "User's session" of the "THE USER: HOW TO and THE RULES" section read.

When the `packagename.zip` file is created, by the `%generatePackage()` macro, a lot of things is happening behind the scenes. This section explains them in more details.

The first information the developer receives after the process ends is a summary report displaying basic information about the package content. In this summary the following elements are displayed: the package location (i.e. folder), developer's `&sysuserid.`, creation timestamp, SAS version, the package encoding information (based on the `description.sas` file), and current SAS session encoding. From the `description.sas` file the package name, version, and license type are extracted and printed. Also the list of required elements is printed if any were provided. The last part of the summary is a table displaying a list of files used to build up the package.

But the summary is only the tip of an iceberg. The following steps are executed when the macro runs. At the beginning the `description.sas` file is tested for existence and when the result is positive the file is read otherwise process is stopped with error. The following macrovariables (descriptors): `packageName`, `packageVersion`, `packageTitle`, `packageAuthor`, `packageMaintainer`, `packageEncoding`, `packageLicense`, `packageRequired` (optional), `packageReqPackages` (optional) are created and obligatory ones (i.e. first seven) are tested for values. If at least one of the descriptors is missing the process is aborted with an error. If the package name is more than *24 characters* or contains illegal symbols (non alphanumeric or underscore) the process is aborted with an error too. Value of the package version should be a positive number hence this is also tested.

If a `zip` file with package name exists inside the package folder the `zip` file is deleted and the new fileref is generated.

In the next step the package folder is scanned and structure of files and subfolders is extracted. Since files and subfolders with code have to be named only with lower case letters - it is tested, if the test fails the process is aborted with an error.

At this point the summary mentioned above is generated.

Further steps create so called "driving" files. The `description.sas` is copied into the `zip` file. The `license.sas` is either copied or MIT license is generated. The `packagemetadata.sas` file with descriptors macrovariables is created.

At this point the `iceloadpackage.sas` file containing the `%ICEloadPackage()` "emergency" macro (described in the "User's "in case of emergency" section) is generated.

The next one is the `load.sas` file. If `packageRequired` or `packageReqPackages` macrovariables are present two parts of code for testing requirements are generated, respectively. Both codes are design to set up the `packageRequiredErrors` macrovariable to 1 (one) if requirements are not met. The first test compares the `proc setinit` output with provided list of required licensed SAS products. The

second test compares the SYSloadedPackages macrovariable with provided list of required packages. If a package from the list is not in the SYSloadedPackages then, to try to load required package, the %loadPackage() macro is called. If the packageRequiredErrors is positive the loading of the package is aborted with the following error message "ERROR: Loading package &packageName. will be aborted! Required SAS components are missing."

After requirements testing all "includes" are generated. In case of functions or formats/informats, if the first one is detected, a code to update cmplib or fmtsearch is generated. If the subfolders of the exec type are detected a snippet code to print their content is added. As the final part of the load.sas code for creating/updating the SYSloadedPackages global macrovariable is added.

After that a code snippet for loading "lazy datasets" is generated into a lazydata.sas code file.

The process continues with generation of the unload.sas code. As the first part the code to print and execute the clean type is assembled. As the second step the code for the macros and formats/informats deletion is generated. Deletion of functions follows after. And the last is the code for libraries unassignment and the SYSloadedPackages update.

In the third file, help.sas, following snippets are generated. The first to print out content of the description.sas file, namely the description part and list of required components if any are provided. The second to print out content of the license.sas file. The third snippet creates a data step used for content search and print out of the help text from the package files (macros, functions, etc.)

The final part of the %generatePackage macro is a series of data steps copying package code files into the zip file and testing existence, parity, and potential overlapping of help tags. If no tags are found a warning is printed in the log. If tags are mismatched or overlapping an error is printed.

Eventually within the package zip file we will find:

- Copies of all files from package subfolders but with modified names, what is needed to keep the ordering in place. Each code file name is extended with a prefix of a form: underscore, subfolder name, and dot. For example if a file name is abc.sas and a subfolder name is 007_macro then the new name is _007_macro.abc.sas.
- The description.sas file (the one described earlier) and the license.sas file.
- The packagemetadata.sas file containing definitions of internal macrovariables used by the %loadPackage(), %helpPackage(), and %unloadPackage() macro.
- The load.sas file containing the code executed by the %loadPackage() macro. The file content is built based on the subfolders and files structure provided by the developer. The file is a series of requirements tests, %includes (with additional automatic note comments in %put statements), and, if need be, set of options modifications e.g. inserts to fmtsearch option for formats/informats or appends to cmplib option for functions. If files of type exec are inside the package a code printing out their content into the log is also attached.
- The help.sas file containing the code executed by the %helpPackage() macro. The file contains 1) code which displays general package description, 2) code which searches for a content based on helpKeyword and prints out the information, and 3) code which, if helpKeyword is "License", prints out the license text.
- The unload.sas file containing the code executed by the %unloadPackage() macro. The file content is built based on the subfolders and files structure provided by the developer. Code inside the file removes macros, functions, formats, datasets and libraries created during loading process. It restores fmtsearch and cmplib options. If clean type subfolder is provided files from within the folder are %included (they are executed at the beginning).

As the final part of the process the testing of the package is executed. Details about the testing process are provided in the next section.

```
<packageName>
 ..
  |
  +-000_libname [one file one libname]
  |           |
  |           +-abc.sas [a file with a code creating libname ABC]
  |
  +-001_macro [one file one macro]
  |         |
  |         +-hij.sas [a file with a code creating macro HIJ]
  |         |
  |         +-klm.sas [a file with a code creating macro KLM]
  |
  +-002_function [one file one function,
  |            |  option OUTLIB= should be: work.&packageName.fcmp.package
  |            |  option INLIB=  should be: work.&packageName.fcmp
  |            |  (both literally with macrovariable name and "fcmp" sufix)]
  |            |
  |            +-efg.sas [a file with a code creating function EFG]
  |
  +-003_format [one file one format,
  |          |  option LIB= should be: work.&packageName.format
  |          |  (literally with macrovariable name and "format" sufix)]
  |          |
  |          +-efg.sas [a file with a code creating format EFG and informat EFG]
  |
  +-004_data [one file one dataset]
  |        |
  |        +-abc.efg.sas [a file with a code creating dataset EFG in library ABC]
  |
  +-005_exec [so called "free code", content of these files will be printed
  |        |  to the log before execution]
  |        |
  |        +-<no file, in this case folder may be skipped>
  |
  +-006_format [if your codes depend on each other you can order them in folders,
  |          |  e.g. code from 003_... will be executed before 006_...]
  |          |
  |          +-abc.sas [a file with a code creating format ABC,
  |                     using the definition of the format EFG]
  |
  +-007_lazydata [one file one dataset]
  |            |
  |            +-klm.sas [a file with a code creating dataset KLM in the WORK library
  |                       dataset will be created on demand by user with the following
  |                       call: %loadPackage(packagename, lazyData=klm)]
  |
  +-...<sequential number>_<type [in lower case]>
  |
  +-00n_clean [if you need to clean something up after an exec file execution,
  |         |  content of these files will be printed to the log before execution]
  |         |
  |         +-<no file, in this case folder may be skipped>
  |
  +-00n+1_test [code for testing the package, used only in the developer's session]
  ...
```

**Figure 2: Example of a package's subfolders structure.**

**Developer's tests.** When all elements of the package are ready and zipped into a `zip` file a good practice is to execute tests which will verify if and how the package works. This subsection describes what possibilities of testing are provided by the `%generatePackage()` macro and how to use them.

**NOTE.** The developer has to download the `loadpackage.sas` file into the same folder as `generatePackage.sas`!

All tests are executed in clean and separate SAS sessions hence the `XCMD` option has to be turned on. A *separate session* means that a `systask` statement is used to call SAS binary and run new session in a batch mode. A *clean session* means that the default configuration file, located in the `!SASROOT` folder, is used to run the testing session.

By default the `%generatePackage()` macro executes test for loading, helping, and unloading a package. The `%loadPackage()`, `%helpPackage()`, and `%unloadPackage()` macros are called in such way that location of the package is the one provided by the `filesLocation` macroparameter. If there are any dependencies (i.e. additional packages are required), unless the `packages` fileref is assigned, the developer has to use the `packages=` macroparameter to point to the folder containing required packages and also a copy of the `loadpackage.sas` file!

Results of this test (and all other provided by the developer), i.e. log and listing, are stored in a subfolder of the maim session `WORK` library, the subfolder name has the following form: `test_YYYYMMDDtHHMMSS`. When the test is done the log is scanned for any error or warning messages. As a summary the developer gets short output with a table presenting: exit status of the `systask` statement, value of the `sysrc` macrovariable after the `systask`, the number of error messages from the log, and the number of warnings messages from the log.

If the developer wants to execute additional tests the following "files and folders" steps have to be performed:

- Create a subfolder of type `test` in the package folder, e.g `999_test`.
- Copy all files with the code for tests into the subfolder.

The test file does not require a code to load package, this code is provided in a separate `autoexec.sas` file so only the testing code should be in the file. There is no formal form of the test file but a good practice would be to call explicit error or warning if the test fails.

Since each file in the `test` subfolder is executed in a separate session the developer can chose to keep all tests in one file or splits tests into separate files.

To suppress any testing the developer has to set up the `testPackage=` macroparmeter to any value different than "Y", e.g. "N" is preferred.

The following list presents file references and libnames created during the testing process in the main developer's session: `sasroot`, `currdir`, `packages`, `TEST`, `TESTWORK`.

## GENERATING PACKAGE IN PRACTICE - a USECASE

The practical **example** will be build based on one of the author's favourite SAS article, namely Mike Rhoads' "*Use the Full Power of SAS in Your Function-Style Macros*" [**2**], which introduces the macro-function-sandwich programming approach. The idea is to allow user to execute SQL "select" code within a data step, e.g.

```
data class_subset;
  set %SQL(select
             name
           , sex
           , height
           from
             sashelp.class
           where
             age > 12
          )
          (rename=(height=heightInch));

  heightCm = 2.54 * heightInch;
run;
```

Thus the package name will be *SQLinDS* and it will be providing the `%SQL()` macro which will allow users to write queries like the one above. Internally the `%SQL()` macro uses a user defined function, another macro, and stores intermediate data (views) inside a predefined library (pointing to a subdirectory of the `work`). Package will be built with 5 files: `description.sas`, two `macros`, one `function`, and one `library`. Let's assume we have created the following package folder `C:/SAS_PACKAGES/SQLinDS/` and we have copied the `generatePackage.sas` file and the `loadpackage.sas` file into the `C:/SAS_PACKAGES/` directory. The structure of subfolders created for the package is presented in Figure 3.

```
<C:/SAS_PACKAGES/SQLinDS/>
 ..
  |
  +-description.sas ❶
  |
  +-000_libname
  |           |
  |           +-dssql.sas ❷
  |
  +-001_macro
  |         |
  |         +-dssql_inner.sas ❸
  |         |
  |         +-sql.sas ❺
  |
  +-002_function
  |            |
  |            +-dssql.sas ❹
  |
  +-license.sas ❻
```

**Figure 3: SQLinDS package - subfolders structure.**

Details of the package files (including license) can be found in the Appendix A or on the web (see "THE CODE" section).

When all files are placed inside proper subfolders we start a new SAS session and we execute the following code:

```
filename packages "C:/SAS_PACKAGES/";
%include packages(generatepackage.sas);
/*ods html;*/
%generatePackage(filesLocation=C:/SAS_PACKAGES/SQLinDS/)
```

As a result, extra the summary report, we receive (inside the `C:/SAS_PACKAGES/SQLinDS/` folder) the `sqlinds.zip` file. Our package is prepared. And ready for sharing!

## THE CODE

If you are interested in testing the approach presented above yourself and want to play a bit with the code and data you can download SAS programs which were the motivation for this paper under the following "world wild web" address:

`http://www.mini.pw.edu.pl/~bjablons/SASpublic/SAS_PACKAGES`

or from authors GitHub:

`https://github.com/yabwon/SAS_PACKAGES`

Also the future versions of this article will be uploaded there.

## REFERENCES


[1] Art Carpenter, "Carpenter's Guide to Innovative SAS Techniques", SAS Press

[2] Mike Rhoads, "Use the Full Power of SAS in Your Function-Style Macros", SAS Global Forum 2012 Proceedings, `https://support.sas.com/resources/papers/proceedings12/004-2012.pdf`

[3] Hadley Wickham, "R Packages: Organize, Test, Document, and Share Your Code", O'Reilly Media 2015, `http://r-pkgs.had.co.nz/description.html`

[4] Frank Mittelbach, Michel Goossens, "The LaTeX Companion, Second Edition", Addison-Wesley 2004, ISBN 0-201-36299-6

[5] `https://realpython.com/python-modules-packages/` as of October 2019

[6] `https://www.internetblog.org.uk/post/1520/what-is-a-linux-package/` as of October 2019


## ACKNOWLEDGMENTS


Author would like to acknowledge **Filip Kulon**, **Krzysztof Socki**, **Allan Bowe**, **Quentin McMullen**, **Piotr Wójcik**, and **Michał Wojtasiewicz** for their contribution and effort to make this paper looks and feel as it should.

Author would like to acknowledge the Citibank Europe PLC Poland for the support.


## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at one of the following e-mail address:


`yabwon`✉`gmail.com`

`bartosz1.jablonski`✉`citi.com`


or via the following LinkedIn profile:

`www.linkedin.com/in/yabwon`

## Appendix A

The content of files composing the `SQLinDS` package.

❶

```
/* This is the description file for the package.          */
/* The colon (:) is a field separator and is restricted  */
/* in lines of the header part.                           */

/* **HEADER** */
Type: Package                                    :/*required, not null*/
Package: SQLinDS                                 :/*required, not null*/
Title: SQL queries in Data Step                  :/*required, not null*/
Version: 1.0                                     :/*required, not null*/
Author: Mike Rhoads (RhoadsM1@Westat.com)         :/*required, not null*/
Maintainer: Bartosz Jablonski (yabwon@gmail.com) :/*required, not null*/
License: MIT                                     :/*required, not null*/
Encoding: UTF8                                   :/*required, not null*/

Required: "Base SAS Software"                    :/*optional*/

/* **DESCRIPTION** */
/* All the text below will be used in help */
DESCRIPTION START:

The SQLinDS package is an implementation of
the macro-function-sandwich concept introduced in:
"Use the Full Power of SAS in Your Function-Style Macros"
the article by Mike Rhoads, Westat, Rockville, MD

Copy of the article can be found at:
https://support.sas.com/resources/papers/proceedings12/004-2012.pdf

SQLinDS package provides following components:
 1) %dsSQL_inner() macro
 2) dsSQL() function
 3) %SQL() macro

Library DSSQL is created in a subdirectory of the WORK library.

DESCRIPTION END:
```

❷

```
/*** HELP START ***/

/* >>> dsSQL library: <<<
 *
 * The dsSQL library stores temporary views
 * generated during %SQL() macro's execution.
 * If possible, created as a subdirectory of WORK:

   LIBNAME dsSQL BASE "%sysfunc(pathname(WORK))/dsSQLtmp";

 * if not then redirected to WORK

   LIBNAME dsSQL BASE "%sysfunc(pathname(WORK))";

**/

/*** HELP END ***/

data _null_;
  length rc0 $ 32767 rc1 rc2 8;
  rc0 = DCREATE("dsSQLtmp", "%sysfunc(pathname(work))/"                );
  rc1 = LIBNAME("dsSQL",    "%sysfunc(pathname(work))/dsSQLtmp", "BASE");
  rc2 = LIBREF ("dsSQL"                                                );
  if rc2 NE 0 then
    rc1 = LIBNAME("dsSQL", "%sysfunc(pathname(work))", "BASE");
run;

libname dsSQL LIST;
```

❸

```
/*** HELP START ***/

/* >>> %dsSQL_Inner() macro: <<<
 *
 * Internal macro called by dsSQL() function.
 *
 * Recommended for SAS 9.3 and higher.
 * Based on paper:
 * "Use the Full Power of SAS in Your Function-Style Macros"
 * by Mike Rhoads, Westat, Rockville, MD
 * https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
 *
**/

/*** HELP END ***/

/* inner macro */
%MACRO dsSQL_Inner() / SECURE;
  %local query;
  %let query = %superq(query_arg);
  %let query = %sysfunc(dequote(&query));

  %let viewname = dsSQL.dsSQLtmpview&UNIQUE_INDEX_2.;
  proc sql;
    create view &viewname as
      &query
    ;
  quit;
%MEND dsSQL_Inner;
```

❹

```
/*** HELP START ***/

/* >>> dsSQL() function: <<<
 *
 * Internal function called by %SQL() macro.
 *
 * Recommended for SAS 9.3 and higher.
 * Based on paper:
 * "Use the Full Power of SAS in Your Function-Style Macros"
 * by Mike Rhoads, Westat, Rockville, MD
 * https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
 *
**/

/*** HELP END ***/

proc fcmp
  /*inlib  = work.&packageName.fcmp*/
  outlib = work.&packageName.fcmp.package
;
  function dsSQL(unique_index_2, query $) $ 41;
    length
      query query_arg $ 32000 /* max query length */
      viewname $ 41
    ;
    query_arg = dequote(query);
    rc = run_macro('dsSQL_Inner' /* <-- inner macro */
                  ,unique_index_2
                  ,query_arg
                  ,viewname
                  );
    if rc = 0 then return(trim(viewname));
    else
      do;
         put 'ERROR:[function dsSQL] A problem with the dsSQL() function';
         return(" ");
      end;
  endsub;
run;
quit;
```

❺

```
/*** HELP START ***/

/* >>> %SQL() macro: <<<
 *
 * Main macro which allows to use
 * SQL's queries in the data step.
 * Recommended for SAS 9.3 and higher.
 * Based on paper:
 * "Use the Full Power of SAS in Your Function-Style Macros"
 * by Mike Rhoads, Westat, Rockville, MD
 * https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
 *
 * EXAMPLE 1: simple sql query

   data class_subset;
     set %SQL(select name, sex, height from sashelp.class where age > 12);
   run;

 * EXAMPLE 2: with dataset options

   data renamed;
     set %SQL(select * from sashelp.class where sex = "F")(rename = (age=age2));
   run;

 * EXAMPLE 3: dictionaries in datastep

   data dictionary;
     set %SQL(select * from dictionary.macros);
   run;

**/

/*** HELP END ***/


/* outer macro */
%MACRO SQL() / PARMBUFF SECURE;
  %let SYSPBUFF = %superq(SYSPBUFF); /* macroquoting */
  %let SYSPBUFF = %substr(&SYSPBUFF, 2, %LENGTH(&SYSPBUFF) - 2); /* remove brackets */
  %let SYSPBUFF = %superq(SYSPBUFF); /* macroquoting */
  %let SYSPBUFF = %sysfunc(quote(&SYSPBUFF)); /* quotes */
  %put NOTE-***the query***; /* print out the query in the log */
  %put NOTE-&SYSPBUFF.;
  %put NOTE-****************;

  %local UNIQUE_INDEX; /* internal variable, a unique index for views */
    %let UNIQUE_INDEX = &SYSINDEX;
  %sysfunc(dsSQL(&UNIQUE_INDEX, &SYSPBUFF)) /* <-- call dsSQL() function,
                                               see the WORK.SQLinDSfcmp dataset */
%MEND SQL;
```

❻ MIT license text (used by default, mind macrocode in the first line):

```
Copyright (c) %sysfunc(today(),year4.) &packageAuthor.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

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
AUTHORS 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 IN THE
SOFTWARE.
```

## Appendix B

Here reader can find headers (with parameters description) of macros `%generatePackage()`, `%loadPackage()`, `%unloadPackage()`, and `%helpPackage()`.

- ---

```
%macro GeneratePackge(
                  /* location of package files */
 filesLocation = %sysfunc(pathname(work))/%lowcase(&packageName.)
,testPackage=Y    /* indicator if tests should be executed,
                     default value Y means "execute tests" */
,packages=        /* location of other packages if there are
                     dependencies in loading */
)/secure;
```

- ---

```
%macro loadPackage(
  packageName                          /* name of a package,
                                          e.g. myPackageFile.zip,
                                          required and not null  */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, options = %str(LOWCASE_MEMNAME)     /* possible options for ZIP filename */
, source2 = /*source2*/               /* option to print out details,
                                          null by default */
, requiredVersion = .                 /* option to test if loaded package
                                          is provided in required version */
, lazyData =                          /* a list of names of a lazy datasets
                                          to be loaded, if not null then
                                          datasets from the list are loaded
                                          instead of a package, asterisk
                                          means "load all datasets" */
, zip = zip                           /* standard package is zip (lowcase),
                                          e.g. %loadPackage(PiPackage)
                                          if the zip is not avaliable use a folder
                                          unpack data to "pipackage.disk" folder
                                          and use loadPackage in the form:
                                          %loadPackage(PiPackage, zip=disk, options=)
                                         */
)/secure;
```

---

```
%macro unloadPackage(
  packageName                          /* name of a package,
                                          e.g. myPackageFile.zip,
                                          required and not null  */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, options = %str(LOWCASE_MEMNAME)     /* possible options for ZIP filename */
, source2 = /*source2*/               /* option to print out details,
                                          null by default */
, zip = zip                           /* standard package is zip (lowcase),
                                          e.g. %unloadPackage(PiPackage)
                                          if the zip is not avaliable use a folder
                                          unpack data to "pipackage.disk" folder
                                          and use unloadPackage in the form:
                                          %unloadPackage(PiPackage, zip=disk, options=)
                                         */
)/secure;
```

```
%macro helpPackage(
  packageName                          /* name of a package,
                                          e.g. myPackageFile.zip,
                                          required and not null  */
, helpKeyword                          /* phrase to search in help,
                                          when empty prints description
                                          "*" means prints all help
                                          "license" prints license */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, options = %str(LOWCASE_MEMNAME)     /* possible options for ZIP filename */
, source2 = /*source2*/               /* option to print out details,
                                          null by default */
, zip = zip                           /* standard package is zip (lowcase),
                                          e.g. %helpPackage(PiPackage,*)
                                          if the zip is not avaliable use a folder
                                          unpack data to "pipackage.disk" folder
                                          and use helpPackage in the form:
                                          %helpPackage(PiPackage, *, zip=disk, options=)
                                         */
)/secure;
```