PharmaSUG 2023 - Paper SD-185

A Light-Weight Framework to Manage Programs and Run All the TLFs in R

Chi-Hua Huang, Astellas Pharma US, Inc.

ABSTRACT

In most cases, a lot of Table/Listing/Figure (TLF) programs are developed in a study and many standard and study-specific considerations take place in its analyses. For each study deliverable, a set of programs are selected and run to produce outputs accordingly. Some programs have dependencies in general. Both program selections and dependencies determine the order of running the programs, i.e., processing time. Furthermore, R pre-loads data into memory, time to load physical data impacts more on processing, especially when dealing with large data.

While R has been widely used to generate analysis results, there is not yet a comprehensive solution available to batch run multiple R scripts concurrently. To address this, this paper presents a light-weight framework that utilizes R base, tidyverse, sassy, and rstudioapi packages in conjunction with MS excel to efficiently produce/manage the TLFs for a deliverable. In addition, it will demonstrate the use of delegate function to connect the individual programs with a run-all program, by embedding the specific logics into a function while creating all the TLFs.

INTRODUCTION

It is common to have more than one deliverable for a study, for example, format review, dry run, IA, and DBL. In addition, there could be DSUR, DSMB and safety annual report for long-term studies. Time to time, the lead programmer has to prepare multiple deliverables in a short time. The best practice is to have the TLF list for each deliverable in a file. Then the run-all program can select the TLF programs for a deliverable from the list. The sequence of execution for the programs can also be assigned in the list.

A nice feature of RStudio, 'Background Job', allows us to create multiple TLFs concurrently and import the analysis data from the global environment of run-all program to each summitted job. In addition, we can assign the maximum number of jobs running at a time to refrain the system from being out of memory.

Another important thing is to have a universal program structure and use delegate functions across all the TLF programs. The main purpose is to couple with the run-all program well, i.e., add or modify the logic behind the scenes, minimize the changes needed due to different deliverables, and reduce the frequency of retrieving the physical file when creating all the TLFs.

In the next sections, it starts the journey from the packages used in the framework, and then folder structure, deliverable file, individual program structure, delegate function, and run-all program.

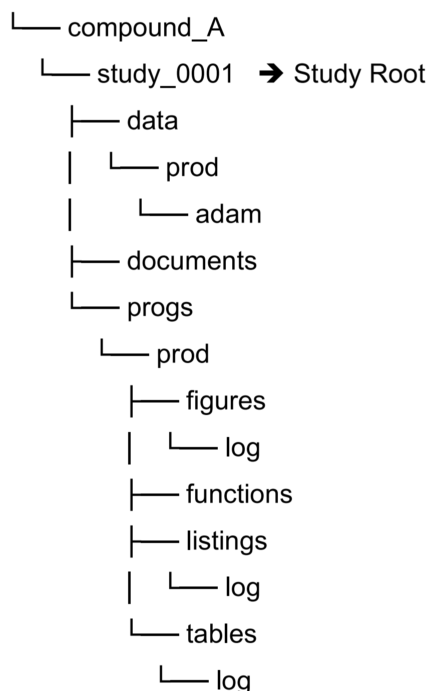
PACKAGES USED

The major packages used in the demo programs presented below are tidyverse, sassy, and rstudioapi. Tidyverse is widely used. It has a bunch of sub-packages that are grouped in five categories, i.e., import, wrangle, visualize, model and program. The usages in the demo are for data import, data wrangling, and programming mainly – not for visualize or model. Four sub-packages of sassy (logr, libr, fmtr, and reporter) are used in the sample programs. Logr provides simple steps to create a program execution log, including program name, R version and packages used, operating system information, and working directory at the log header. Furthermore, logs from dplyr and tidyr packages are consolidated in the main log as well. Libr loads related datasets in a folder as a library. It supports several file types, including 'sas7bdat', 'xpt', 'xls', 'xlsx', 'csv', and r data. Fmtr formats basic summary statistics, like range, quantiles, median, event number (percentage), and mean (standard deviation). Reporter package is for writing out the rtf and PDF files. It defines the structure of a TFL in five parts, including page header, page footer, title, footnote and content. All these parts can be added to the output separately. Lastly, Rstudioapi is utilized to interact with the document open in order to get program name and path while creating a page

header. Another crucial function of rstudioapi is submitting background jobs for each TLF – shown in Display 1.

PROJECT FOLDER STRUCTURE

The following folder structure is in the demo project. 'Study_0001' is the study root, and the working directory. Refer to Table 1 for the purpose of each folder.



Folder	Purpose
/data/prod/adam	To put analysis data sets
/documents	To put deliverable file
/progs/prod/tables /progs/prod/listings /progs/prod/figures	▪ To put TLF programs in the respective folder names ▪ To put the rtf and PDF files created from the TFL programs
/progs/prod/tables/log /progs/prod/listings/log /progs/prod/figures/log	To put TLF programs in the respective folder names
/functions	To put the global functions

Table 1 Folders Description

DELIVERABLE FILE

We record the TLF information for each deliverable in an excel file, named 'deliverable.xlsx'. Table 2 is an example consisting of two major deliverables, 'IA' and 'DBL'. Columns 'Program Name' and 'Output Name' form a unique key for each TLF output. It allows us to have multiple tables having different contents/layout but sharing the same table number in different deliverables. Columns 'Table Number', 'Title (N)', 'Footnote (N)', and 'Source' are used for automation when creating TLF. Column 'Groups' is

used to select a set of TLFs for a deliverable. We use '|' as a delimiter. Column 'Batch Number' is used to group a couple of TLFs when submitting a run-all program. The run-all program will execute TLF programs by from batch number and execution sequence number in an ascending order.

A common use case is to assign a larger batch number to the TLFs that read the large analysis data in or do the complicated analysis which takes long time. In addition, we can also assign the different batch number and execution sequence number considering the dependencies of data or results from other programs. For example, ADSL must be created/updated before other analysis datasets. In addition, some figures may be created from tables results.

Program Name	Output Name	Output Type	Output Number	Title 3	Title 4	Footnote 2	Footnote 3	Source	Grouped Select	Batch Number	Execution Sequence #
t_ae_socpt	_all_saf	Table	Table 9.1.1	Treatment-Emergent Adverse Events	Safety Analysis Set	Treatment emergent events are defined as events which start on or after the start of treatment.	Percentages are based on the number of subjects in the safety population within each treatment group.	ADAE, ADSL	IA|DBL	1	2
t_ae_socpt	_all_saf_cq01	Table	Table 9.1.2	Treatment-Emergent Adverse Events of Special Interest: Dermatologic Events	Safety Analysis Set	Treatment emergent events are defined as events which start on or after the start of treatment.	Percentages are based on the number of subjects in the safety population within each treatment group.	ADAE, ADSL	DBL	1	3
t_ae_overall	_all_saf	Table	Table 9.1	Analysis of Adverse Event Summary	Safety Analysis Set	Treatment emergent events are defined as events which start on or after the start of treatment.	Percentages are based on the number of subjects in the safety population within each treatment group.	ADAE, ADSL	IA|DBL	1	1
t_vs_summary	_all_saf	Table	Table 9.2	Vital Signs	Safety Analysis Set	End of treatment is the last on-treatment assessment of the specified vital sign (on or before the Week 24 visit).		ADVS, ADSL	IA|DBL	2	2
t_lb_summary	_all_saf	Table	Table 9.3	Quantitative Laboratory Test Results in SI Units, Chemistry	Safety Analysis Set	At each post-baseline analysis visit n is the number of subjects with non-missing values at the analysis visit and at baseline.	End of Treatment' is the last observed value while on treatment (prior to or at Week 24).	ADLB, ADSL	IA|DBL	2	1

Table 2 Deliverable File - Output

The deliverable file can also record the important study information. It can be a central repository to store the global variables used across all or most TLFs. In the following Table 3, the sponsor's name, study name, and deliverable name displayed in the page header of each page are recorded in the deliverable file.

Global Variable Description	Global Variable Name	Global Variable Value
Study number (printed on the output)	study_name	Study_0001
Deliverable Name (printed in the top right of the page, line 1)	output_status	Final DBL
Sponsor name (printed on the output)	sponsor_name	Astellas

Table 3 Deliverable File - Global

INDIVIDUAL PROGRAM STRUCTURE

The following pseudo code demonstrates the main structure for an individual program. There are two major functions used in the sample codes. First, the error handler function 'tryCatch' is used. The statements surrounded by curly brackets after 'finally' statement will be executed whether the program ends successfully or not. Second, an idea of delegate function is implemented. By using it, there is a placeholder to add some specific logic when a TLF program is submitted by a run-all program. For example, a TLF program may create multiple outputs, and some of them may not be needed in all the deliverables. We use 'wrap_mainfunc' function to delegate 'mainfunc' function. When a program is submitted by a run-all program, it checks if an output is needed. If not needed, then 'mainfunc' will not be executed.

```
tryCatch({
  # Step 1: Initialize the environment
  # (1.1) Load the default packages, for example, tidyverse, sassy, and rlang
  # (1.2) Load the global user-defined functions under
  #       "study root/progs/prod/functions"

  # Step 2: Turn on the log function in Sassy logr package

  # Step 3: Get the program information
  # (3.1) Study root path
  # (3.2) Program path with and without the program name
  # (3.3) Program name with and without file extension name

  # Step 4: Load the analysis data by the delegate function, 'wrap_libname'

  # Step 5: Create a main function that contains the logics for creating TLFs
  mainfunc <<- function(Output Name in deliverable file,
                        …[List of variables used to filter analysis data]){
    # (5.1) Open the log file and error message file
    # (5.2) Read the deliverable file to get the headers and footers for a
    #       TLF
    # (5.3) Perform the analysis
    # (5.4) Format the analysis results to output
    # (5.5) Close the log file
  }

  # Step 6: Call the main function through a delegator function for each TLF
  wrap_mainfunc(Output Name A in deliverable file, …)
  wrap_mainfunc(Output Name B in deliverable file, …)

}, finally = {
  # Step 7: Actions taken after the program is completed (no matter it ends
  #         Successfully or not).
  # (7.1) output a rds file when the program is executed by a run all program

})
```

A TEAE program, 't_ae_socpt.r', that implements this structure can be found in Appendix I. In addition, the user-defined functions that implements above steps can be found from Appendix III.

DELEGATE FUNCTION

Time to time, we want to plugin additional logics under certain situations when the program is submitted by a run-all program or add additional check-ups to a function before calling it. Creating a delegate function to wrap function can make it easier and more manageable. The delegate function has the same

arguments as the original function, and these arguments can be either forwarded to the original function when certain criteria are met or to another function as needed.

Besides the delegate function, ‘wrap_mainfunc’, mentioned in the last paragraph, another use case is to retrieve data sets efficiently. To minimize the frequency of retrieving the physical file from the server, the analysis data sets are retrieved once and then copied to the background jobs as needed when creating all TLFs in the run-all program. This improves the efficiency significantly when a data set size is large.

Below are the parts of demo codes for ‘wrap_libname’ function that deserved to mention. First, ‘match.call’ function returns all the arguments, and they are forwarded to ‘args_to_list’ function to convert the arguments to a list (lines 2 to 4). After that, the arguments can be fetched by variable name and used to proceed to the additional process.

```
2  in_arg_var_list <- as.list(match.call())
3  in_arg_var_list <- in_arg_var_list[c(-1)]
4  arg_var_values <- args_to_list("libname", in_arg_var_list)
5  in_name <- arg_var_values["name"]
6  in_path <- arg_var_values["directory_path"]
7  in_engine <- arg_var_values["engine"]
```

Second, there is a check for library name. Only ‘adam’, ‘sdtm’, ‘raw’, or ‘derived’ are valid ones. By applying this convention, the TLF programs are clearer and more readable among the programs whthin a study.

```
61 if (!(in_name %in% c('adam','sdtm','raw','derived'))){
62   message(paste0("Libname is not in allowed name: ",
63                  "'adam','sdtm','raw','derived'."))
64   stop()
65 }
```

Third, the ‘do.call’ function is used to call function’. It executes the function named in the first argument and pass the second argument, a list, to the function to be its arguments.

```
70 do.call("libname", arg_var_values, envir = .GlobalEnv)
```

In addition, this delegate function looks up the data sets copied from the run-all program environment first. If a dataset is not found, it uses ‘haven’ package to retrieve the data set from the server.

```
37 vec_text <- c(as.vector(as.character(arg_var_values["filter"])))
38 vec_text_r <- str_replace_all(str_replace_all(str_remove_all(
39   str_replace_all(vec_text, " ", ""), '\\\"'),
40   "\\w{0}[c][(]",""), "[)]",""))
41 vec_1 <- c(str_split_1(vec_text_r, "[,]"))
42 for (em_vec1 in vec_1) {
43   list_globals <- ls(envir = .GlobalEnv)
44   if (!any(list_globals == paste0(in_name, ".",
45                                  as.character(em_vec1)))){
46     if (in_engine == "sas7bdat"){
47       assign(paste0('adam.', em_vec1),
48              haven::read_sas(
49                file.path(adam_path,paste0(em_vec1,'.',in_engine))),
50              envir = .GlobalEnv)
51     }else if (in_engine == "xpt"){
52       assign(paste0('adam.', em_vec1),
53              haven::read_xpt(
54                file.path(adam_path,paste0(em_vec1,'.',in_engine))),
55              envir = .GlobalEnv)
56     }
```

```
57   }
58 }
```

RUN-ALL PROGRAM

Figure 1 is the flowchart of creating TLFs without a run-all program. Each program retrieves the deliverable file and analysis data sets from the database in local or server and loads data into its global environment. When the server or the network is busy, it can take a while to retrieve the data.

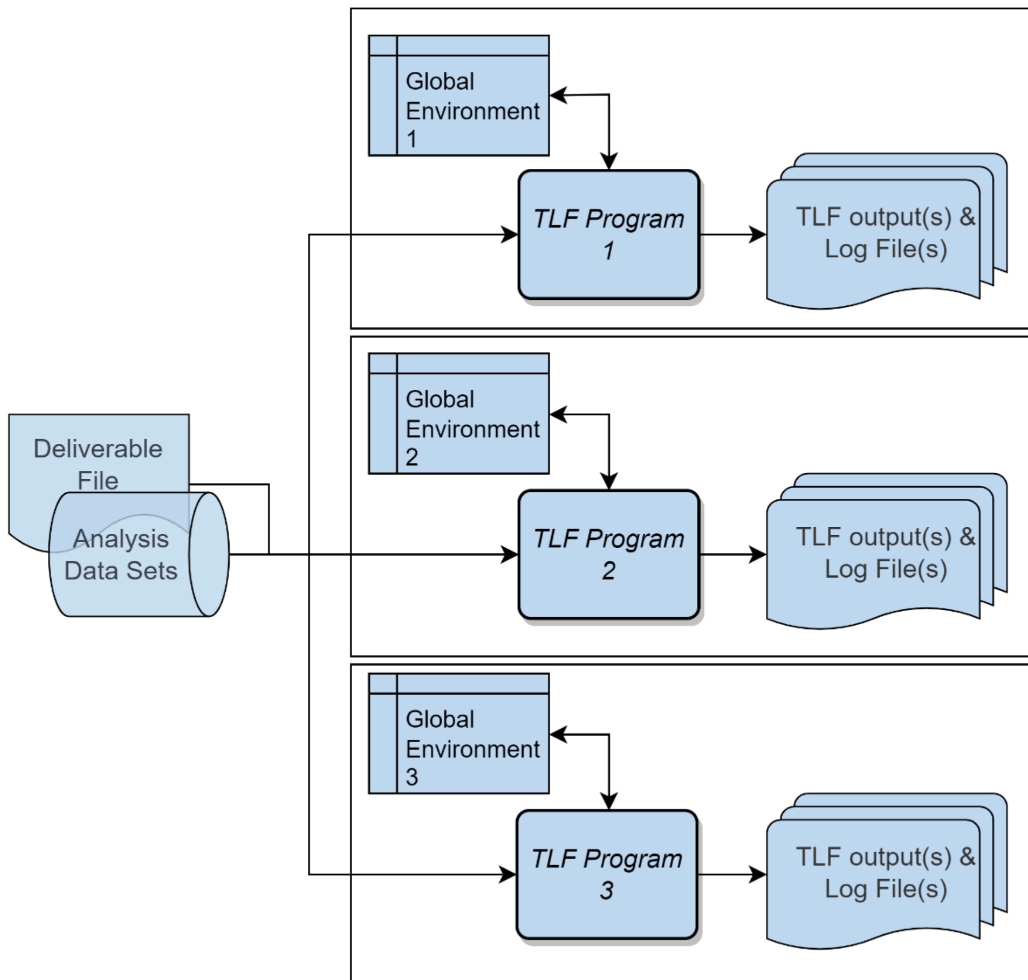


Figure 1 Create TLFs without run-all Program

Figure 2 is the flowchart that demonstrate how a run-all program in this project works. The run-all program gets the list of programs from the deliverable file to create TLFs. It submits each of these programs as an independent background job. The environments for each background job are completely isolated, so the temporary data or results in the background jobs won't contradict each other. In addition, it also imports all the variables in the global environment to each background job, including the analysis data, R objects and so on. Since the data source is displayed in each TFL, we use the 'Source' column in the deliverable file to determine when to load and unload a data set to the global environment of run-all program. Copying data from memory is much faster than from physical files. It saves more time when there are more repeated TLFs (i.e., with same layout as in unique).

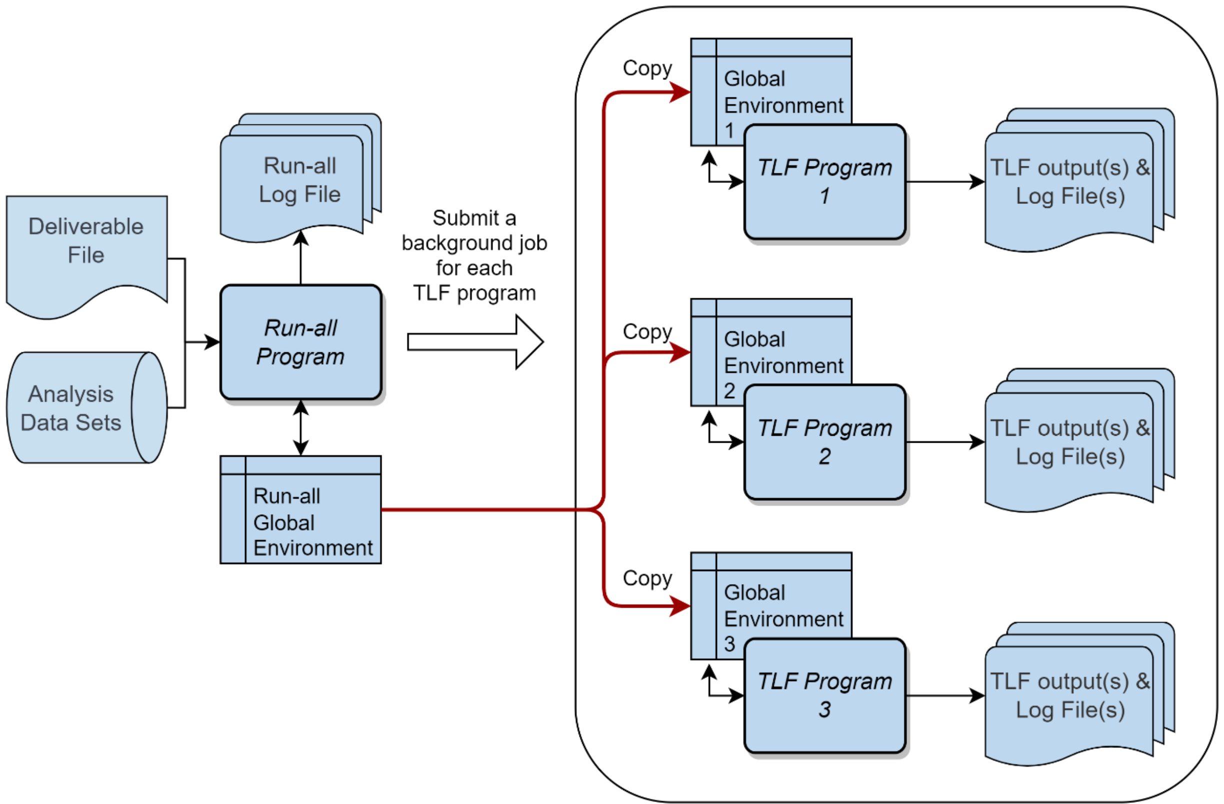


Figure 2 Create TLFs with run-all Program

The demo run-all program can be found in Appendix II. There is a couple of important statements worthy mentioning. First, we specify that the table programs in 'IA' deliverable and are needed in the following codes. The run-all program selects rows where 'Grouped Select' cell contains 'IA' and 'Output Type' is 'Table' from the deliverable file.

```
81    # Assign the group and type
82    grouped_select <- "|IA|" %>% put()
83    output_type <- "Table" %>% put()
84
85    # Select TLF programs by grouped_select and output_type
86    sep(paste0("Select a set of program to run for ", grouped_select,
87               " ", output_type))
88    df_pgm <- dlvOutput %>% mutate(
89      Grouped_Select = if_else(is.na(Grouped_Select), '',
90                          paste0('|', dlvOutput$Grouped_Select, '|'))
91    ) %>% filter(Output_Type == output_type ) %>%
92      select(Program_Name, Output_Name, Output_Type, Output_Number,
93             Grouped_Select, Batch_Number, `Execution_Sequence_#`,
94             Source)
```

Second, we assign 5 to the maximum number of programs that can be submitted concurrently. A 'rds' file is scheduled to output at the end of execution for a TFL program, no matter it ends successfully or not. We can also add more additional information to forward to the run-all program in this 'rds' file if needed. Since there are no ways to communicate between run-all program and individual background jobs, the mechanism we use to monitor a background job is to search and read the 'rds' file in the log folder.

```
210   # Setup how many programs can submit at the same time
```

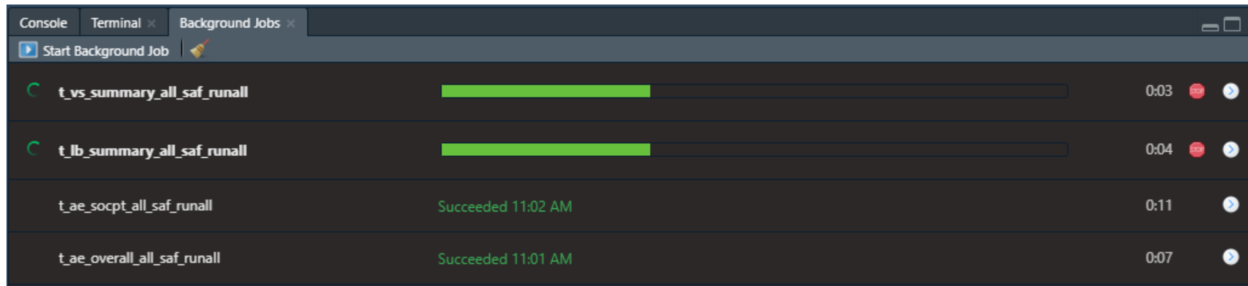
```
211  job_exec_limit <- 5 %>% put()
.....
265    # if number of programs submitted great or equal the limit,
266    # check if there are any of them are completed already.
267    if (exec_cnt >= job_exec_limit) {
268      list_rst <-
269        list.files(path = file.path(runall_pgm_location, 'log'),
270                   pattern = ptn)
271      # Use exec_pgms list and log file to determine how many jobs
272      # are still running
273      while (length(exec_pgms[which(!exec_pgms %in%
274                                       list_rst)]) >= job_exec_limit) {
275        list_rst <-
276          list.files(path = file.path(runall_pgm_location, 'log'),
277                     pattern = ptn)
278        # Wait for 2 seconds
279        Sys.sleep(2)
280      }
281      exec_cnt = length(exec_pgms[which(!exec_pgms %in% list_rst)])
282    }
```

In addition, the run-all program will not submit the background jobs of next batch until the background jobs of current batch are all completed.

```
232    # If the jobs of the previous batch are not all completed,
233    # need to wait until all the jobs in previous batch completed.
234    if(length(exec_pgms_batch)>0){
235      max_exec_batchnum <- max(unlist(exec_pgms_batch), na.rm = T)
236      # Check if all the jobs of the previous are completed
237      # every 1 second
238      if (df_pgm[loop_idx,]$Batch_Number > max_exec_batchnum){
239        while (length(exec_pgms_batch[names(exec_pgms_batch) %in%
240                                         list_rst == FALSE]) > 0){
241          list_rst <-
242            list.files(path = file.path(runall_pgm_location, 'log'),
243                       pattern = ptn)
244          Sys.sleep(1)
245        }
246      }
247    }
```

At last, we use 'jobRunScript' function in rstudioapi package to submit a background job. 'importEnv = TRUE' means the run-all program imports its global environment to the background jobs automatically. Display 1 shows the real-time background jobs status in RStudio console.

```
198    job_id <- rstudioapi::jobRunScript(pgm_path,
199                                       workingDir = study_root,
200                                       name = paste0(Program_Name,
201                                                     Output_Name,
202                                                     "_runall"),
203                                       importEnv = TRUE,
204                                       exportEnv = paste0(Program_Name,
205                                                          Output_Name,
206                                                          "_runall"))
```



Display 1 Background Jobs Console

CONCLUSION

There is not a unified run-all program that fits all the scenarios. If the subjects or datapoints in a study are not so many, then the analysis datasets can be pre-loaded in the run all program environment and copy to the multiple background jobs. It would save the time on loading analysis data. However, when working on the late phase or ISS/ISE studies, it's common that the file sizes of finding analysis datasets are over 500 MB. Then more facts need to be considered to adjust the run all program. For example, the size of memory of the server, the upper limit of number of background jobs running concurrently, and the number of TLF programs that retrieve large analysis datasets. The lead programmer may have to adjust the run-all program to fit the specific needs. Also, the run-all program can't communicate with the background jobs directly, so they need to communicate through a physical file.

Despite of above things to be concerned, by applying this framework, we can manage not only the elements in each TLF, but also the scope in each deliverable easily. In addition, a seven-step program structure and module functions let the programs be in a similar style, making the programs more readable and easier to maintain. Delegate functions do things behind the seen, which connect the run-all program and individual TLF programs smoothly.

REFERENCES

Wickham, Hadley and Grolemund, Garrett (2017). "R for Data Science". Retrieved January 4, 2023 (https://r4ds.had.co.nz/index.html).

Wickham, Hadley (2014, 2019). "Advanced R". Retrieved January 4, 2023 (https://adv-r.hadley.nz/index.html).

Bosak, David. "sassy system". Retrieved January 4, 2023 (https://r-sassy.org/).

Bosak, David. "reporter: Creates Statistical Reports". Retrieved January 4, 2023 (https://cran.r-project.org/web/packages/reporter/index.html).

Bosak, David. "libr: Libraries, Data Dictionaries, and a Data Step for R". Retrieved January 4, 2023 (https://cran.r-project.org/web/packages/libr/index.html).

Bosak, David. "logr: Creates Log Files". Retrieved January 4, 2023 (https://cran.r-project.org/web/packages/logr/index.html).

Bosak, David. "fmtr: Easily Apply Formats to Data". Retrieved January 4, 2023 (https://cran.r-project.org/web/packages/fmtr/index.html).

Ushey, Kevin, JJ Allaire, Hadley Wickham, and Gary Ritchie. "rstudioapi". Retrieved January 4, 2023 (https://cran.rstudio.com/web/packages/rstudioapi/index.html.)

PHUSE. Github Repository - CDISC Pilot Study01 ADaM data. Retrieved February 27, 2023

(https://github.com/phuse-org/phuse-scripts/blob/master/data/adam/TDF_ADaM_v1.0.zip)

CDISC. Github Repository – sdtm-adam-pilot-project. Retrieved February 27, 2023

(https://github.com/cdisc-org/sdtm-adam-pilot-project)

ACKNOWLEDGMENTS

The author would like to acknowledge Astellas Pharma, Inc. for providing the opportunity to work on this paper and thanks Reza, Sharmeen (Director Statistical Programming at Astellas Pharma) and Clark, Steve (Associate Statistical Programming Director at Astellas Pharma) for their thoughtful comments.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Chi-Hua Huang
Astellas Pharma Global Development, Inc.
chi-hua.huang@astellas.com
https://www.astellas.com/en/

APPENDIX I

```
1  tryCatch({
2
3    # Step 1: Initialize the environment
4    init_pgm <- file.path("progs", "prod", "functions", "init_pgm.r")
5    source(init_pgm)
6    init_pgm()
7
8    # Step 2: Turn on the log function in Sassy logr package
9    options("logr.autolog" = TRUE)
10
11   # Step3: Get the program information
12   current_pgm_information()
13
14   # Get temp location for log and report output
15   logdir <- paste0(pgm_location,"/log")
16
17   # Attach the current environment
18   e <<- current_env()
19
20   # Step 4: Load the analysis data by the delegate function,
21   #         'wrap_libname'
22   assign('adam_path', paste(study_root,"/data/prod/adam",sep = ""),
23          envir = .GlobalEnv)
24   wrap_libname(adam, adam_path, "xpt", quiet = TRUE,
25                filter = c('adsl','adae'))
26
27   # Step 5: Create a main function to contain the logic for creating TLFs
28   mainfunc <<- function(output_fl_name, ae_filter){
29     # output_fl_name <- "_all_sad_saf"
30     # ae_filter <- expression(1 == 1)
31
32     # Remove msg file before execution
33     if (file.exists(file.path(logdir,
34                               paste0(pgm_name, output_fl_name, ".msg")))){
35       file.remove(file.path(logdir,
36                             paste0(pgm_name, output_fl_name, ".msg")))
37     }
38
39     # Open the log and message files in a temporary folder
40     log_file <- paste0(pgm_name, output_fl_name, ".log")
41     msg_file <- paste0(pgm_name, output_fl_name, ".msg")
42
43     lf <- log_open(file.path(logdir, log_file))
44
45     # output adam path to the log
46     paste0("adam_path = ", adam_path) %>% put()
47
48     # Output study root to log file
49     paste0("study_root = ", study_root) %>% put()
50
51     # Send code to the log
52     log_code()
53
54     # Read deliverable file
55     sep("Read deliverable file")
```

```
56      header_footer_from_dlv <- file.path("progs", "prod", "functions",
57                                          "header_footer_from_dlv.r")
58      source(header_footer_from_dlv)
59
60      # Get headers and footnotes from deliverable file
61      headers_list <- headers_from_dlv(output_name = output_fl_name)
62      footers_vector <- footers_from_dlv(output_name = output_fl_name)
63
64      # Create population dataset
65      sep("Prepare table data")
66      pop <- adam.adsl %>% filter(!(is.na(TRT01AN))) %>%
67        rename(TRTAN=TRT01AN, TRTA=TRT01A) %>%
68        group_by(TRTAN, TRTA) %>% count() %>% put()
69
70      adae0 <- adam.adae %>%
71        filter(SAFFL == 'Y' & TRTEMFL == 'Y' & eval(ae_filter))
72
73      # Combine data for all level - Overall, SOC and PT
74      adae <- rbind(adae0 %>% mutate(SRCN = 1, MORD = 1, SORD = 1,
75                                     AESOC = "Overall",
76                                     AEDECOD = "Overall"),
77                    adae0 %>% mutate(SRCN = 2, MORD = 2, SORD = 1,
78                                     AEDECOD = AESOC),
79                    adae0 %>% mutate(SRCN = 3, MORD = 2, SORD = 2))
80
81      # A subject only counts once per TRTAN and PT
82      adae1 <- adae %>%
83        distinct(MORD, SORD, TRTAN, TRTA, USUBJID, AESOC, AEDECOD, SRCN) %>%
84        group_by(MORD, SORD, TRTAN, TRTA, AESOC, AEDECOD, SRCN) %>%
85        count() %>% put()
86
87      # Merge population dataset to count percentage and format the result
88      # for displaying
89      adae2 <-left_join(adae1, pop %>% rename(BIGN=n), by="TRTAN") %>%
90        mutate(CNTPCT = fmt_cnt_pct(n, BIGN)) %>%
91        select(MORD, SORD, TRTAN, TRTA.x, CNTPCT, n, AESOC, AEDECOD, SRCN)
92
93      dummyae <- data.frame(SRC='Dummy', CNTPCT_0 = NA, CNTPCT_54 = NA,
94                            CNTPCT_81 = NA, stringsAsFactors=FALSE)
95
96      # Transpose result
97      adae3 <- adae2 %>%
98        pivot_wider(id_cols = !TRTA.x, names_from = TRTAN,
99                    values_from = c("CNTPCT", "n"), ) %>%
100       arrange(MORD, AESOC, SORD, desc(n_81), AEDECOD)
101
102     adae4 <- full_join(adae3, dummyae) %>%
103       mutate(
104         CNTPCT_0 =
105           ifelse(is.na(CNTPCT_0), fmt_cnt_pct(0, 1), CNTPCT_0),
106         CNTPCT_54 =
107           ifelse(is.na(CNTPCT_54), fmt_cnt_pct(0, 1), CNTPCT_54),
108         CNTPCT_81 =
109           ifelse(is.na(CNTPCT_81), fmt_cnt_pct(0, 1), CNTPCT_81),
110         AEDECOD = if_else(SORD == 2, paste0("  ", AEDECOD), AEDECOD)
111       ) %>% filter(is_na(SRC))
112
```

```
113    # Create a vector for population data for table usage
114    arm_pop <- pop %>% ungroup() %>% select(TRTAN, n) %>%
115      deframe() %>% put()
116
117    # Select table columns
118    adae5 <- adae4 %>% ungroup() %>%
119      select(AESOC, AEDECOD, CNTPCT_0, CNTPCT_54, CNTPCT_81) %>% put()
120
121    # Create table
122    tbl <- create_table(adae5, first_row_blank = TRUE, width = 9) %>%
123      # Set default attributes for columns on table
124      column_defaults(from = CNTPCT_0, to = CNTPCT_81, width = 1) %>%
125      # Define table stub
126      stub(vars = c("AESOC", "AEDECOD"),
127           label = "System Organ Class\n   Preferred Term", width = 4) %>%
128      # Define columns
129      define(AESOC, blank_after = TRUE) %>%
130      define(AEDECOD) %>%
131      define(`CNTPCT_0`, align = "center",
132             label = "Placebo", n = arm_pop["0"]) %>%
133      define(`CNTPCT_54`, align = "center",
134             label = "Xanomeline \n Low Dose", n = arm_pop["54"]) %>%
135      define(`CNTPCT_81`, align = "center",
136             label = "Xanomeline\n 0.High Dose", n = arm_pop["81"])
137
138
139    # Output the result in rtf file
140    outfile("RTF", headers_list, footers_vector, tbl)
141    # Output the result in pdf file
142    outfile("PDF", headers_list, footers_vector, tbl)
143
144    # Clean Up ------------------------------------------------------------
145    sep("Clean Up")
146
147    # Close the log
148    log_close()
149
150    # Only display the log in the console when in development stage.
151    log_to_console(lf)
152
153  }
154
155  # Step 6: Call the main function through a delegate function for
156  # each TLF
157  wrap_mainfunc("_all_saf", expression(1 == 1))
158  wrap_mainfunc("_all_saf_cq01", expression(AOCC01FL == 'Y'))
159
160  # Sys.sleep(120)
161
162}, finally = {
163
164  # Step 7: Actions taken after the program is completed (no matter
165  # it ends with success or failure).
166  # End of program
167  end_pgm()
168
169})
```

Table Program 1 - t_ae_socpt.r

APPENDIX II

```
1   tryCatch({
2
3     # Clean up the packages loaded, global variables, and release memory
4     clear_environment <- file.path("progs", "prod", "functions",
5                                    "clear_environment.r")
6     source(clear_environment)
7     clear_environment()
8
9     # Include the delegate function for sassy libname
10    wrap_libname <- file.path("progs", "prod", "functions",
11                              "wrap_libname.r")
12    source(wrap_libname)
13
14    # Include packages needed
15    library(dplyr)
16    library(tidyr)
17    library(readr)
18    library(stringr)
19    library(purrr)
20    library(readxl)
21    library(libr)
22    library(haven)
23    library(rlang)
24    library(logr)
25
26    # Turn on the log function in Sassy logr package
27    options("logr.autolog" = TRUE)
28
29    # Flag to indicate that the runall program is executing
30    runall_mode <- 'Y'
31
32    # Log run all execution datetime
33    runall_dttm <- str_replace_all(Sys.time(),"[-: ]", '_')
34
35    # Assign study root
36    study_root <-
37      "C:/Users/A4033752/Documents/project/compound_A/study_0001"
38
39    # Set working directory
40    setwd(study_root)
41
42    # Assign the library used in all the TLF programs
43    assign('adam_path', paste(study_root,"/data/prod/adam",sep = ""),
44           envir = .GlobalEnv)
45
46    # Assign TFL folder path
47    runall_pgm_path <- rstudioapi::documentPath()
48    runall_pgm_path_split <- (runall_pgm_path %>% str_split("/"))[[1]]
49    runall_pgm_location <-
50      paste(runall_pgm_path_split[-length(runall_pgm_path_split)],
51            collapse = "/")
52
```

```
53   # Get temp location for log and report output
54   logdir <- paste0(runall_pgm_location,"/log")
55
56   # Remove msg file before execution
57   if (file.exists(file.path(logdir, "run-all.msg"))){
58     file.remove(file.path(logdir, "run-all.msg"))
59   }
60
61   # Open the log and message files in a temporary folder
62   log_file <- "run-all.log"
63   msg_file <- "run-all.msg"
64   lf <- log_open(file.path(logdir, log_file))
65
66   # Send code to the log
67   log_code()
68
69   # Read the deliverable file to select TLF programs for a deliverable
70   sep("Read deliverable file")
71   read_dlv <- file.path("progs", "prod", "functions", "read_dlv.r")
72   source(read_dlv)
73   read_dlv()
74   deliverable_name <- dlvGlobal %>%
75     filter(`_macro_name` == "output_status") %>% select(`_macro_value`)
76   # Format the variable names in dlvoutput
77   upd_varnm <- lapply(colnames(dlvOutput),
78                       function (x)  str_replace_all(x, ' ', '_'))
79   colnames(dlvOutput) <- upd_varnm
80
81   # Assign the group and type
82   grouped_select <- "|DBL|" %>% put()
83   output_type <- "Table" %>% put()
84
85   # Select TLF programs by grouped_select and output_type
86   sep(paste0("Select a set of program to run for ", grouped_select,
87              " ", output_type))
88   df_pgm <- dlvOutput %>% mutate(
89     Grouped_Select = if_else(is.na(Grouped_Select), '',
90                              paste0('|', dlvOutput$Grouped_Select, '|'))
91   ) %>% filter(Output_Type == output_type ) %>%
92     select(Program_Name, Output_Name, Output_Type, Output_Number,
93            Grouped_Select, Batch_Number, `Execution_Sequence_#`,
94            Source) %>%
95     mutate(
96       pgm_path = paste0(runall_pgm_location, "/", Program_Name, ".r"),
97       # If the batch number is not assigned, then assign 9999
98       Batch_Number = if_else(is.na(Batch_Number) , 9999,
99                              as.double(Batch_Number)),
100      # If the execution sequence number is not assigned,
101      # then assign 9999
102      `Execution_Sequence_#` =
103        if_else(is.na(`Execution_Sequence_#`) , 9999,
104                as.double(`Execution_Sequence_#`))
105    )
106  df_pgm$outflag <- str_detect(df_pgm$Grouped_Select,
107                               fixed(grouped_select))
108  df_pgm <- df_pgm %>% filter(outflag == TRUE) %>%
109    arrange(Batch_Number, `Execution_Sequence_#`, Source) %>% mutate(
```

```
110      behind = lag(Source),
111      ahead = lead(Source)
112    )
113
114  df_pgm$ordnum <- 1:NROW(df_pgm)
115
116  # Get list of analysis data sets in the folder
117  ext_type='xpt'
118  patt = paste0('\\.', ext_type, '$')
119  list_files <- list.files(path=as.character(adam_path), pattern=patt,
120                           all.files=TRUE, full.names=FALSE)
121  list_files_fmt <-
122    gsub("\\s", ", ", as.character(str_remove(list_files,
123                                              paste0('.', ext_type))))
124
125  read_dlv()
126
127  # Function to submit background job for a TLF
128  exec_jobs <-  function(x){
129    Program_Name <- x[1]
130    Output_Name <- x[2]
131    pgm_path <- x[9]
132
133    # Assign the program name, output name, and program location
134    # to the global environment
135    assign("runall_exec_pgm", Program_Name, envir = .GlobalEnv)
136    assign("runall_exec_output_name", Output_Name, envir = .GlobalEnv)
137    assign("runall_exec_path",
138           paste0(runall_pgm_location, "/", Program_Name, ".r"),
139           envir = .GlobalEnv)
140
141    # Use the Source indicated in the deliverable file to load/unload
142    # the data sets
143    Source <- x[8]
144    Source_1 <- as.vector(str_trim(str_split_1(Source, pattern = '[,]')))
145
146    # If a data set is not loaded in the previous background job,
147    # then load it to the global environment
148    prev_Source <- x[11]
149    if (is.na(prev_Source)){
150      prev_Source_1 <- as.vector(NULL)
151    }else{
152      prev_Source_1 <-
153        as.vector(str_trim(str_split_1(prev_Source, pattern = '[,]')))
154    }
155
156    add_list <- setdiff(Source_1, prev_Source_1)
157    remove_list <- setdiff(prev_Source_1, Source_1)
158    add_idx = 1
159    for (em_add in add_list) {
160      add_idx2 = 1
161      for(em_fl in list_files_fmt){
162        if (toupper(em_add) == toupper(em_fl)){
163          if (ext_type == 'sas7bdat'){
164            assign(paste0('adam.', em_fl),
165                   read_sas(file.path(adam_path,
166                                      paste0(list_files_fmt[add_idx2],
```

```
167                                             '.', ext_type))),
168                     envir = .GlobalEnv)
169           }else if (ext_type == 'xpt'){
170             assign(paste0('adam.', em_fl),
171                     read_xpt(file.path(adam_path,
172                                         paste0(list_files_fmt[add_idx2],
173                                                 '.', ext_type))),
174                     envir = .GlobalEnv)
175           }
176         }
177         add_idx2 = add_idx2 + 1
178       }
179       add_idx = add_idx + 1
180     }
181 
182     # If an existed data set is used in the current background job,
183     # then unload it from the global environment
184     rm_idx = 1
185     for (em_del in remove_list) {
186       for(em_fl2 in list_files_fmt){
187         if (toupper(em_del) == toupper(em_fl2)){
188           list_globals <- ls(envir = .GlobalEnv)
189           list_globals <-
190             list_globals[list_globals ==
191                             paste0('adam.', as.character(em_fl2))]
192           rm(list=list_globals, envir = .GlobalEnv)
193         }
194       }
195       rm_idx = rm_idx + 1
196     }
197 
198     job_id <- rstudioapi::jobRunScript(pgm_path,
199                                         workingDir = study_root,
200                                         name = paste0(Program_Name,
201                                                       Output_Name,
202                                                       "_runall"),
203                                         importEnv = TRUE,
204                                         exportEnv = paste0(Program_Name,
205                                                            Output_Name,
206                                                            "_runall"))
207     job_id
208   }
209 
210   # Setup how many programs can submit at the same time
211   job_exec_limit <- 5 %>% put()
212   # Get how many programs scheduled
213   job_counts <- NROW(df_pgm)
214   # loop index
215   loop_idx <- 1
216   # How many programs are submitted
217   exec_cnt <- 0
218   # Max batch number executing
219   max_exec_batchnum <- 0
220   # Number of programs in execution
221   exec_pgms <- NULL
222   exec_pgms_batch <- list()
223 
```

```
224  # list for program results in rds file
225  list_rst <- NULL
226  # pattern for searching rds files
227  ptn <- paste0(str_replace_all(deliverable_name, "[-: ]", '_') , '_',
228                runall_dttm, '_', "[[:alnum:][:alpha:]_]*\\.rds")
229
230  while (loop_idx <= job_counts)
231  {
232    # If the jobs of the previous batch are not all completed,
233    # need to wait until all the jobs in previous batch completed.
234    if(length(exec_pgms_batch)>0){
235      max_exec_batchnum <- max(unlist(exec_pgms_batch), na.rm = T)
236      # Check if all the jobs of the previous are completed
237      # every 1 second
238      if (df_pgm[loop_idx,]$Batch_Number > max_exec_batchnum){
239        while (length(exec_pgms_batch[names(exec_pgms_batch) %in%
240                                            list_rst == FALSE]) > 0){
241          list_rst <-
242            list.files(path = file.path(runall_pgm_location, 'log'),
243                       pattern = ptn)
244          Sys.sleep(1)
245        }
246      }
247    }
248
249    # Submit a batch job
250    tmpid <- df_pgm[loop_idx,] %>% apply(1, exec_jobs)
251
252    # Log submitted jobs
253    exec_pgms[length(exec_pgms)+1] <-
254      paste0(str_replace_all(deliverable_name, "[-: ]", '_') , '_',
255             runall_dttm, '_',
256             runall_exec_pgm,
257             runall_exec_output_name,
258             ".rds")
259
260    exec_pgms_batch[as.character(exec_pgms[length(exec_pgms)])] <-
261      df_pgm[loop_idx,]$Batch_Number
262    exec_cnt <- exec_cnt + 1
263    loop_idx <- loop_idx + 1
264
265    # if number of programs submitted great or equal the limit,
266    # check if there are any of them are completed already.
267    if (exec_cnt >= job_exec_limit) {
268      list_rst <-
269        list.files(path = file.path(runall_pgm_location, 'log'),
270                   pattern = ptn)
271      # Use exec_pgms list and log file to determine how many jobs
272      # are still running
273      while (length(exec_pgms[which(!exec_pgms %in%
274                                          list_rst)]) >= job_exec_limit) {
275        list_rst <-
276          list.files(path = file.path(runall_pgm_location, 'log'),
277                     pattern = ptn)
278        # Wait for 2 seconds
279        Sys.sleep(2)
280      }
```

```
281       exec_cnt = length(exec_pgms[which(!exec_pgms %in% list_rst)])
282     }
283   }
284
285   # Wait until all the job completed
286   while (length(list.files(path = file.path(runall_pgm_location, 'log'),
287                             pattern = ptn)) < job_counts)
288   {
289     Sys.sleep(2)
290   }
291
292}, finally = {
293
294   # Collect all the log files for each TLF into one, and then delete
295   # all the individual ones.
296   all_logs <- NULL
297   list_rst <- list.files(path = file.path(runall_pgm_location, 'log'),
298                          pattern = ptn)
299   for (x in 1:job_counts) {
300     tmp_rds <-
301       readRDS(file.path(runall_pgm_location, 'log', list_rst[x]))
302     all_logs <- bind_rows(all_logs, tmp_rds)
303   }
304
305   # List the executed programs in the run-all program log
306   all_logs %>% select(Program_Name, Output_Name, Output_Number, ordnum,
307                       enddttm) %>% put(n = 10000)
308
309   saveRDS(all_logs,
310           file = file.path(runall_pgm_location, 'log',
311                            paste0("all_logs_",
312                                   str_replace_all(
313                                     deliverable_name, "[-: ]", '_'),
314                                   '_', runall_dttm, ".rds")))
315   for (z in 1:job_counts) {
316     file.remove(file.path(runall_pgm_location, 'log', list_rst[z]))
317   }
318
319   # Assign 'N' to runall_mode when finish
320   runall_mode <- "N"
321
322   # Clean Up ---------------------------------------------------------
323   sep("Clean Up")
324
325   # Close the log
326   log_close()
327
328   # Only display the log in the console when in development stage.
329   writeLines(readLines(lf, encoding = "UTF-8"))
330
331})
```

Run all Program - run-all.r

APPENDIX III

```
1  wrap_libname <- function(...){
2    in_arg_var_list <- as.list(match.call())
3    in_arg_var_list <- in_arg_var_list[c(-1)]
4    arg_var_values <- args_to_list("libname", in_arg_var_list)
5    in_name <- arg_var_values["name"]
6    in_path <- arg_var_values["directory_path"]
7    in_engine <- arg_var_values["engine"]
8
9    if (exists("runall_mode") && runall_mode == 'Y'){
10     if (!("filter" %in% names(in_arg_var_list))){
11       patt = paste0('\\.', as.character(in_engine), '$')
12       list_files <- list.files(path=get(as.character(in_path)),
13                                pattern=patt, all.files=TRUE,
14                                full.names=FALSE)
15
16       list_files_fmt <- gsub("\\s", ", ", as.character(
17         str_remove(list_files, paste0('.', as.character(in_engine)))))
18       list_globals <- ls(envir = .GlobalEnv)
19       for (em in list_files_fmt) {
20         if (!any(list_globals == paste0(in_name, ".", as.character(em)))){
21           if (in_engine == "sas7bdat"){
22             assign(paste0('adam.', em), haven::read_sas(
23               file.path(adam_path,paste0(em,'.',in_engine))),
24               envir = .GlobalEnv)
25           }else if (in_engine == "xpt"){
26             assign(paste0('adam.', em), haven::read_xpt(
27               file.path(adam_path,paste0(em,'.',in_engine))),
28               envir = .GlobalEnv)
29           }
30         }
31       }
32
33       message('No dataset name(s) is/are specified! This could cuase the
34               performance issue when creating all the TLFs.')
35
36     }else {
37       vec_text <- c(as.vector(as.character(arg_var_values["filter"])))
38       vec_text_r <- str_replace_all(str_replace_all(str_remove_all(
39         str_replace_all(vec_text, " ", ""), '\\\"'),
40         "\\w{0}[c][(]",""), "[)]","")
41       vec_1 <- c(str_split_1(vec_text_r, "[,]"))
42       for (em_vec1 in vec_1) {
43         list_globals <- ls(envir = .GlobalEnv)
44         if (!any(list_globals == paste0(in_name, ".",
45                                         as.character(em_vec1)))){
46           if (in_engine == "sas7bdat"){
47             assign(paste0('adam.', em_vec1),
48                    haven::read_sas(
49                      file.path(adam_path,paste0(em_vec1,'.',in_engine))),
50                    envir = .GlobalEnv)
51           }else if (in_engine == "xpt"){
52             assign(paste0('adam.', em_vec1),
53                    haven::read_xpt(
54                      file.path(adam_path,paste0(em_vec1,'.',in_engine))),
55                    envir = .GlobalEnv)
56           }
57         }
```

```
58          }
59        }
60    }else {
61      if (!(in_name %in% c('adam','sdtm','raw','derived'))){
62        message(paste0("Libname is not in allowed name: ",
63                       "'adam','sdtm','raw','derived'."))
64        stop()
65      }
66      if (!("filter" %in% names(in_arg_var_list))){
67        message('No dataset name(s) is/are specified! This could cuase the
68                performance issue when creating all the TLFs.' )
69      }
70      do.call("libname", arg_var_values, envir = .GlobalEnv)
71
72      lib_load(as.character(in_name))
73      list_globals <- ls(envir = .GlobalEnv)
74      list_globals <- list_globals[list_globals == as.character(in_name)]
75      rm(list=list_globals, envir = .GlobalEnv)
76    }
77}
```

Function 1 - wrap_libname.r

```
1 init_pgm <- function(x){
2
3   library(tidyverse)
4   library(sassy)
5   library(rlang)
6
7   assign("study_root",
8          "C:/Users/[User Name]/Documents/project/compound_A/study_0001",
9          envir = .GlobalEnv)
10  setwd(study_root)
11
12  # Load Functions
13  ptn <- "[[:alnum:][:alpha:]_]\\.[r|R]"
14  list_functions <-
15    as.list(list.files(path = file.path(study_root,
16                                        'progs', 'prod', 'functions'),
17                       pattern = ptn))
18  names(list_functions) <- as.vector(list_functions)
19  list_functions <- list_functions[list_functions != "init_pgm.r"]
20
21  for(i in 1:length(list_functions)) {
22    assign(paste0('aa', i), file.path('progs', 'prod', 'functions',
23                                      list_functions[i]))
24    source(get(paste0('aa', i)))
25  }
```

Function 2 - init_pgm.r

```
1 args_to_list <- function(wrap_func, in_arg_var_list){
2
3   arg_var_list <- formals(wrap_func)
4   arg_var_names <- names(arg_var_list)
5   arg_var_values <- list()
6
```

```
7   for(i in 1:length(arg_var_list)){
8     arg_var_values <- append(arg_var_values,
9                                  arg_var_list[arg_var_names[i]])
10  }
11
12  in_arg_var_names <- names(in_arg_var_list)
13  for(k in 1:length(in_arg_var_list)){
14    if ((names(in_arg_var_list[k])) == ''){
15      arg_var_values[k] <- in_arg_var_list[k]
16    }else {
17      for(j in 1:length(arg_var_names)){
18        if (names(in_arg_var_list[k]) == names(arg_var_list[j])){
19          arg_var_values[j] <- in_arg_var_list[k]
20          break
21        }
22      }
23    }
24  }
25
26  return(arg_var_values)
27}
```

Function 3 - args_to_list.r

```
1 read_dlv <- function(studyRoot = NULL, pgm_path = NULL,
2                      output_name = NULL){
3   current_pgm_information <- file.path("progs", "prod", "functions",
4                                         "current_pgm_information.r")
5   source(current_pgm_information)
6
7   if (is.null(pgm_path)){
8     current_pgm_information()
9   }else {
10    current_pgm_information(pgm_path)
11  }
12
13  if (is.null(studyRoot)){
14    studyRoot <- getwd()
15  }
16
17  dlvFilePath <- paste(studyRoot,"/documents/dlv_read_only.xlsx",
18                       sep = "")
19
20  if (!is.null(output_name)){
21    if (!(exists("runall_mode") && runall_mode == 'Y')){
22      dlvGlobal<- read_excel(dlvFilePath, sheet = "global")
23      dlvOutput<- read_excel(dlvFilePath, sheet = "outputs", skip = 1)
24      dlvLib<- read_excel(dlvFilePath, sheet = "lib")
25    }
26    dlvOutput <- dlvOutput  %>%
27      filter (`Program Name` == pgm_name &
28                `Output Name` == output_name)
29    source_datasets <- dlvOutput %>% select(Source) %>%
30      str_replace_all(" ", "") %>% tolower()
31  }else {
32    dlvGlobal<- read_excel(dlvFilePath, sheet = "global")
33    dlvOutput<- read_excel(dlvFilePath, sheet = "outputs", skip = 1)
34    dlvLib<- read_excel(dlvFilePath, sheet = "lib")
```

```
35   }
36
37   assign("dlvFilePath", dlvFilePath, envir = .GlobalEnv)
38   assign("dlvGlobal", dlvGlobal, envir = .GlobalEnv)
39   assign("dlvOutput", dlvOutput, envir = .GlobalEnv)
40   assign("dlvLib", dlvLib, envir = .GlobalEnv)
41}
```

Function 4 - read_dlv.r

```
1 library(readxl)
2
3 headers_from_dlv <- function (pgm_path = NULL, output_name){
4   current_pgm_information <- file.path("progs", "prod", "functions",
5                                        "current_pgm_information.r")
6   source(current_pgm_information)
7
8   if (is.null(pgm_path)){
9     current_pgm_information()
10  }else {
11    current_pgm_information(pgm_path)
12  }
13
14  pgm_path <- get('pgm_path', envir=.GlobalEnv)
15
16  if (is.null(study_root)){
17    study_root <- getwd()
18  }
19
20  read_dlv(output_name = output_name)
21
22  # Generate text for 'output name'
23  output_name_text <- paste0("[Output: ", pgm_name, output_name,
24                             ".rtf", "]")
25  # Generate text for program path/name and output name
26  # print(paste0("pgm_path = ", pgm_path))
27  pgm_path_output_name_text <- paste0("Program: ", pgm_path, " ",
28                                      output_name_text)
29  # Assign deliverable name
30  deliverable_name <- dlvGlobal %>%
31    filter(`_macro_name` == "output_status") %>% select(`_macro_value`)
32
33  # Study Name
34  study_name <- dlvGlobal %>%
35    filter( `_macro_name` == "study_name") %>% select(`_macro_value`)
36  study_name_text <- paste0("Study: ", study_name)
37  # Generate text for 'source'
38  data_source <- dlvOutput %>% select(Source)
39  data_source_text <- paste0("Source: ", data_source)
40
41  # Generate text for table title
42  tbl_number <- dlvOutput %>% select(`Output Number`)
43
44  # Generate text for table title
45  tbl_title <- dlvOutput %>% select(`Title 3`)
46
```

```
47  # Generate text for population
48  population <- dlvOutput %>% select(`Title 4`)
49  text_list = list(list(pgm_path_output_name_text, deliverable_name),
50                   list(study_name_text, tbl_number ,data_source_text),
51                   tbl_title, population,
52                   paste0(pgm_name, output_name))
53  return(text_list)
54}
55
56footers_from_dlv <- function (pgm_path = NULL, output_name){
57  current_pgm_information <- file.path("progs", "prod", "functions",
58                                       "current_pgm_information.r")
59  source(current_pgm_information)
60
61  if (is.null(pgm_path)){
62    current_pgm_information()
63  }else {
64    current_pgm_information(pgm_path)
65  }
66
67  if (is.null(study_root)){
68    study_root <- getwd()
69  }
70
71  read_dlv(output_name = output_name)
72
73  # Generate text for 'source'
74  footnotes <- dlvOutput %>% select(starts_with("Footnote"))
75  footnotes_t <- t(footnotes)
76  footnotes_t <- footnotes_t[!is.na(footnotes_t)]
77  footer_vector <- c(footnotes_t)
78
79  return(footer_vector)
80}
```

Function 5 - header_footer_from_dlv.r

```
1 current_pgm_information <- function(pgm_path = NULL){
2   if (exists("runall_exec_path")){
3     pgm_path <- runall_exec_path
4   }else if (is.null(pgm_path)){
5     pgm_path <- rstudioapi::getSourceEditorContext()$path
6   }else {
7     pgm_path <- pgm_path
8   }
9
10  # Get current program name
11  pgm_path_split <- (pgm_path %>% str_split("/"))[[1]]
12  pgm_location <- paste(pgm_path_split[-length(pgm_path_split)],
13                        collapse = "/")
14  pgm_full_name <- pgm_path_split[length(pgm_path_split)]
15  pgm_name <- substring(pgm_full_name, 1, nchar(pgm_full_name)-2)
16
17  assign("pgm_path", pgm_path, envir = .GlobalEnv)
18  assign("pgm_location", pgm_location, envir = .GlobalEnv)
19  assign("pgm_full_name", pgm_full_name, envir = .GlobalEnv)
```

```
20  assign("pgm_name", pgm_name, envir = .GlobalEnv)
21  assign("study_root", getwd(), envir = .GlobalEnv)
22}
```

Function 6 - current_pgm_information.r

```
1 wrap_mainfunc <- function (outfl, ...){
2   if (exists("runall_mode") && runall_mode == "Y"){
3     if (outfl == runall_exec_output_name){
4       get("mainfunc")(outfl, ...)
5     }
6   }else {
7     get("mainfunc")(outfl, ...)
8   }
9 }
```

Function 7 - wrap_mainfunc.r

```
1 # This is a function to output files through sassy reporter package
2 outfile <- function(out_type, headers_list, footers_vector, tbl){
3   # Create report spec. Assign output file name, location, type
4   # and font.
5   rpt <- create_report(file.path(pgm_location, headers_list[[5]]),
6                        output_type = out_type,
7                        font = "Arial") %>%
8     # Assign font size
9     options_fixed(font_size = 9) %>%
10    # Assign page header from dlv file
11    page_header(headers_list[[1]][1], headers_list[[1]][2],
12                width = 7) %>%
13    # Assign title from dlv file
14    titles( headers_list[[2]][1], headers_list[[2]][2],
15            headers_list[[2]][3],
16            columns = 3, header = TRUE, bold = FALSE,
17            font_size = 9, blank_row = "none") %>%
18    titles(headers_list[[3]], headers_list[[4]], bold = FALSE,
19           font_size = 9, align = "center", header = TRUE,
20           blank_row = "below") %>%
21    # Add table content
22    add_content(tbl) %>%
23    # Add footnotes from dlv file and a solid line on the top of
24    # the borders
25    footnotes(footers_vector, borders = "top") %>%
26    # Add page footer
27    page_footer(paste0("Date ", Sys.time()), "Astellas",
28                "Page [pg] of [tpg]")
29
30  # Write to file
31  res <- write_report(rpt)
32}
```

Function 8 - outfile.r

```
1 end_pgm <- function(x){
2   # Output a dataframe for individual output
3   if (exists("runall_pgm_path")){
```

```
4     end_dttm <- Sys.time()
5     df_pgm2 <- df_pgm %>%
6       filter(Program_Name == runall_exec_pgm &
7                Output_Name == runall_exec_output_name)
8     df_pgm2$enddttm <- end_dttm
9     saveRDS(df_pgm2,
10            file = file.path(runall_pgm_location, 'log',
11                             paste0(str_replace_all(deliverable_name,
12                                                    "[-: ]",
13                                                    '_') ,
14                                    '_',
15                                    runall_dttm, '_',
16                                    runall_exec_pgm,
17                                    runall_exec_output_name, ".rds")))
18  }
19}
```

Function 9 - end_pgm.r

```
1 clear_environment <- function(keep_vect = NULL){
2   # Clean up the package loaded
3   tryCatch(
4     {invisible(lapply(paste0('package:', names(sessionInfo()$otherPkgs)),
5                       detach, character.only=TRUE, unload=FALSE,
6                       force=TRUE))
7     },
8     error=function(cond) {
9       if (cond[1] == "invalid 'name' argument") {
10        return("Info: No other packages existed!")
11      }else {
12        return(cond)
13      }
14    }
15  )
16
17  library(dplyr)
18  library(tibble)
19  library(tidyr)
20  library(readr)
21  library(stringr)
22  library(purrr)
23
24  # Clean up the global environment
25  list_globals <- ls(envir = .GlobalEnv)
26  names(list_globals) <- as.vector(list_globals)
27  list_globals <- list_globals[list_globals != "clear_environment"]
28  rm(list = list_globals, envir = .GlobalEnv)
29
30  tryCatch(
31    # Shut down the graphical device/driver
32    {dev.off(dev.list()["RStudioGD"])
33    },
34    error=function(cond) {
35      if (cond[1] == "argument is of length zero") {
36        return("Info: RStudioGD is already off!")
37      }else {
```

```
38        return()
39      }
40    }
41  )
42
43  # Release the memory
44  gc()
45
46}
```

Function 10 - clear_environment.r

```
1log_to_console <- function (log_file){
2
3  if (exists("runall_mode") && runall_mode == "Y"){
4    # No needs to write out the log to console
5  }else {
6    writeLines(readLines(log_file, encoding = "UTF-8"))
7  }
8}
```

Function 11 - log_to_console.r