PharmaSUG 2026 - Paper DV-357

Three Ways to Over-Engineer Your SAS Custom Steps

Mary Dolegowski and S. Robert Collins, SAS

ABSTRACT

Custom Steps in SAS Viya® offer a way to create reusable, user-friendly code for specific tasks. Custom Steps are created as flows which can include SAS and open source code as well as other Custom Steps or macros. In addition, Custom Steps allow you to create a flexible graphical user interface to guide users when setting parameters and options. This paper illustrates three ways to enhance your Custom Steps by: implementing using hidden reference tables; managing dependencies; custom warnings, errors, and debug logic.

INTRODUCTION

The term “over-engineered” in this context is a bit tongue-in-cheek. Some of the features discussed may be overly complicated for simple or lightly used processes. However, for those designing out more complex and layered processes these options these processes can open up the dynamic nature of Custom Steps and increased ease of use for the end user of the Custom Steps.

There are a few points to consider when deciding if adding these over-engineered options. These approaches are justified if the end-user benefits from the results; if the time needed to maintain the Custom Step is reduced; or if the processes shield the user from issues such as out-of-synch linked tables. The answer can vary and the Custom Steps can accommodate both options while keeping the interface the same. If the designer’s goal is a flexible and reusable collection of Custom Steps that can easily be applied to multiple scenarios, all while providing an intuitive user experience, then these are some tools to keep in mind.

The following sections will walk through three different examples for where designers can enhance the basic processes of Custom Steps. We will show how to set up hidden reference tables, then expand on the example by adding control option dependencies, and then end with enabling debugging and working with custom logging.

HIDDEN REFERENCE TABLES

Hidden reference tables combine a collection of features offered in the Custom Steps that together provide a more dynamic set of list options. Custom Steps have the ability to hide objects that are not relevant to the end user but are useful for the overall process; provide automatic dynamic data updates; and the ability to customize the processes to align with simplifying the end user’s experience. These lists can be displayed in a variety of controls. For example, the following section shows the process populating a drop-down list.

In many situations simply having a predefined, static list of options will suffice and be a more simplistic design choice. Hidden reference tables provide a more dynamic alternative that utilizes multiple control objects to improve the user experience when dealing with data that gets updated more often. The important thing to consider when determining which approach is best is how often the list of choices changes and the maintenance required around the Custom Step. Is the Custom Step going to be used often enough that having it be more dynamic will save time or is it something worth updating at a more *ad hoc* cadence.

One of the primary benefits of using a hidden reference table is that it can be tied to an existing dataset. For example, if you have a dataset that contains the display name of a report and the corresponding URI, the display name can be used in the hidden reference table for ease of the users, while the back-end code contained in the Custom Step can easily use the URI for programmatic needs. Since the data is in a dataset, the references between a display name and any connecting identifiers can already be preset and then used with ease within the Custom Step. Together this provides a friendly approach for users that shields them from the more complex back-end process.

HOW TO CREATE A HIDDEN REFERENCE TABLE

Below is an example of how to create a hidden reference table. The process assumes that you are already within the Custom Step design tab.

1. Add an Input Table from the Data folder within the controls domain.
2. Uncheck ‘Required’ under the Input Table Properties as this will be preset.
3. Check the ‘Make control read-only’ to not have the input table control be interactable.
4. (Optional) Check the ‘Hide control at runtime’ check box to not have this section show for end users.
5. Fill in Input Table defaults options for the following two options to preset the reference table:
    a. Default library
    b. Default table

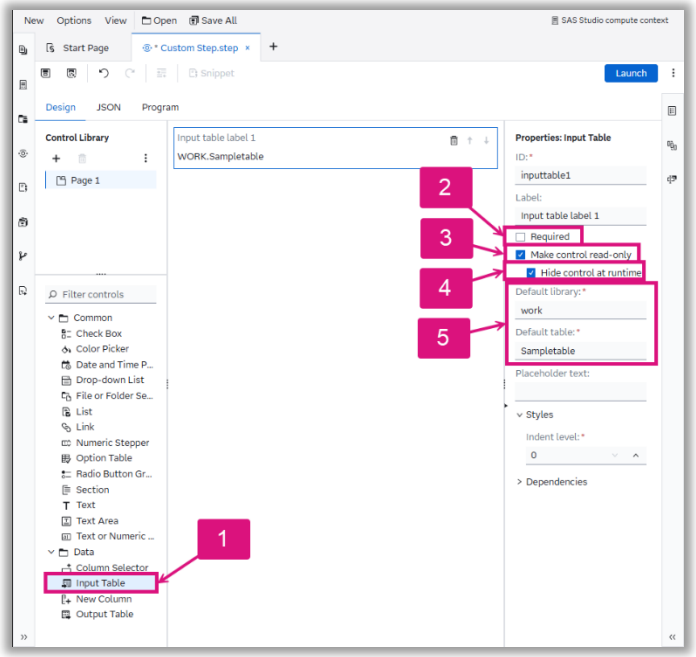

Figure 1: Hidden Reference Table Steps 1 - 5

6. Add Column Selector from the Data folder within the controls domain.
7. Check the ‘Make control read-only’ to not have the input table control be interactable.
8. (Optional) Check the ‘Hide control at runtime’ check box to not have this section show for end users.
9. Under ‘Determine where values come from:’ select the radio button ‘Table’.
10. Select the option that corresponds to the Input Table from step 1 under the ‘Link to input table’ drop-down.
11. Change Minimum and Maximum columns both to 1 so that only one variable is selected.
12. Add the variable name to be used in the in the Item field and click the + to add it to the value list. This sets the variable to be shown in the drop-down list in step 13.

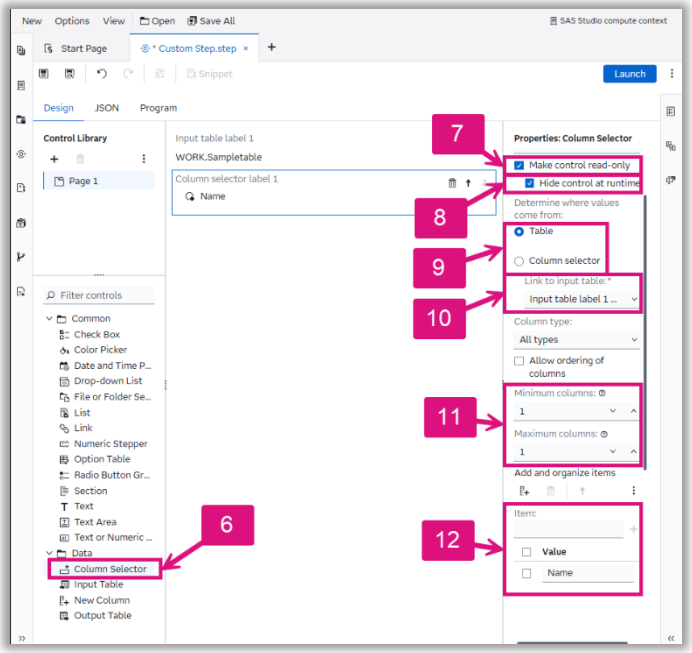

Figure 2: Hidden Reference Table Steps 6 - 12

13. Add Drop-down List from the Common folder within the controls domain.
14. (Optional) Check the ‘Required’ check box to make the drop-down list a required field.
15. Under ‘Determine where values come from:’ select the radio button ‘Dynamic list’.
16. Select the Column Selector set up in step 6 in the ‘Reference a column selector’ drop down.

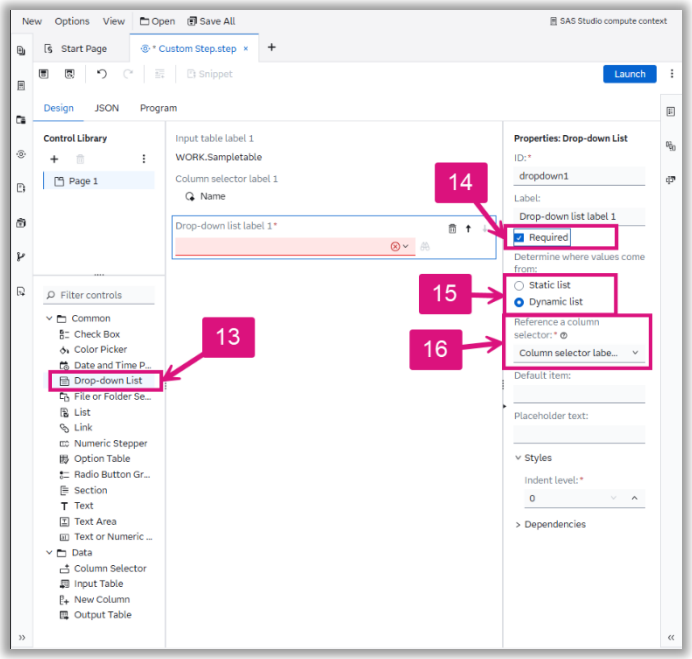

Figure 3: Hidden Reference Table Steps 13 – 16

DEPENDENCIES

Dependencies within Custom Steps let designers determine what control objects can be visible or enabled based on prior selections within the Custom Step. This allows designers to restrict what end users can either see or interact with, simplifying the user experience while reducing the need for code on the back end to prevent conflicts or errors. The visibility settings for dependencies hide entire objects based on the logic set with display rules. The enablement settings for dependencies enable or disable objects by greying them out. Both provide familiar direction for end users but do introduce additional testing for the designer.

While dependencies create a clear guide for end users, there are design requirements and testing that cause them to need more validation than just splitting the processes into independent Custom Steps. Designers have to decide if it is more effective to provide multiple options to end users in one contained Custom Step or if it is more effective to provide multiple Custom Steps and then expect the end user to select the right one. The other consideration is how having a field available which the end user may or may not need/interact with, might influence the back-end code. What happens if the end user enters something in an unneeded field either purposely or accidentally? Will that cause issues? If it might, that could be a good use case for creating a dependency.

Using the example in the next section, if the designer has a set list but wants the end user to have the flexibility to add on to it or use something more custom, a dependency can be a useful tool. By adding in the check box to allow for options outside of the provided list, the code simply needs to determine if the check box is checked or not. At the same time, the end user will see the options dynamically change based on if the box is checked. If it is, then the original options are removed and a new option to manually add a field appears. This ensures a clearer end-user experience.

HOW TO CREATE A DEPENDENCY

This example will expand on the Hidden Reference Table to add an option to manually add on to the reference list but the process can be applied to numerous control options and sub domains

1. Add Check Box above hidden reference table.
2. Change the Label to something descriptive.

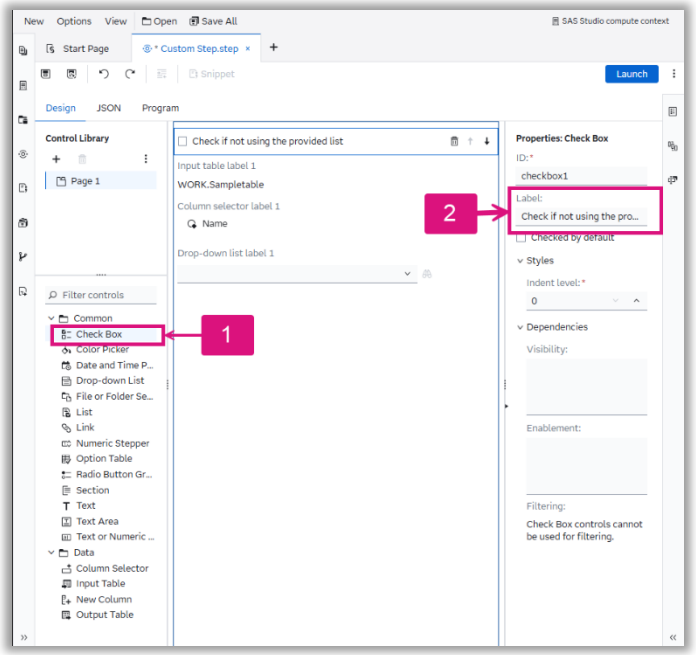

Figure 4: Dependencies Steps 1 - 2

3. In Drop-down list properties uncheck ‘Required’.
4. Expand Dependencies option in Drop-down list properties and add "!$checkbox1" to either ‘Visibility’ or ‘Enablement’ based on requirements. This will ensure that when the checkbox is unchecked the drop-down list will be either visible or enabled.

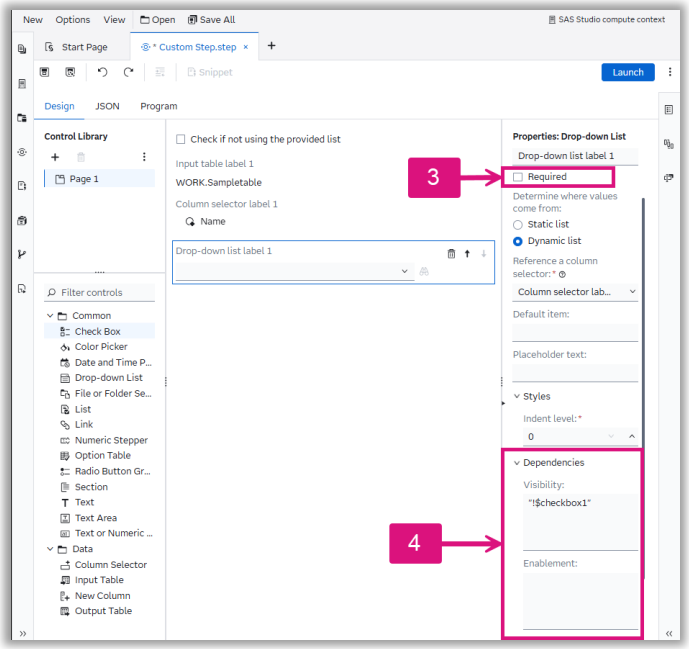

Figure 5: Dependencies Steps 3 to 4

5. Add Text or Numeric Field
6. Expand Dependencies option in Text or Numeric Field properties and add "$checkbox1" to either ‘Visibility’ or ‘Enablement’ based on requirements

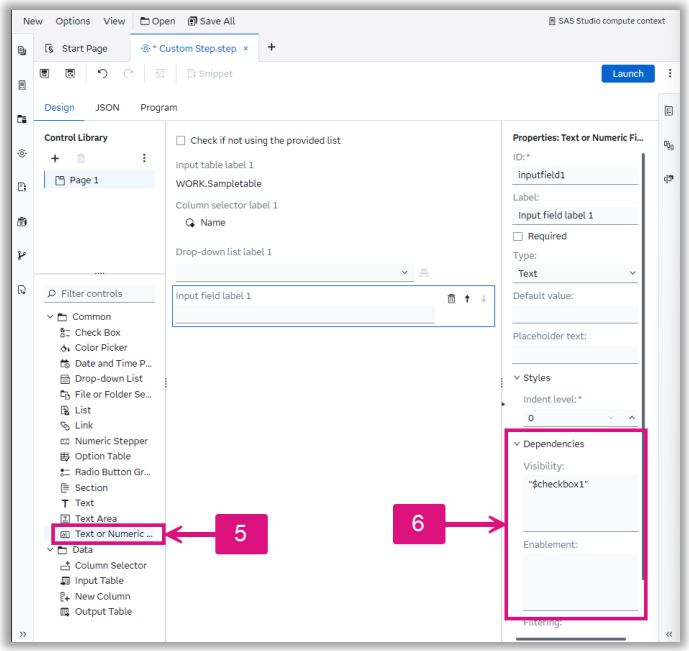

Figure 6: Dependencies Steps 5 to 6

The below figures show the two options for interaction now provided in the same Custom Step.

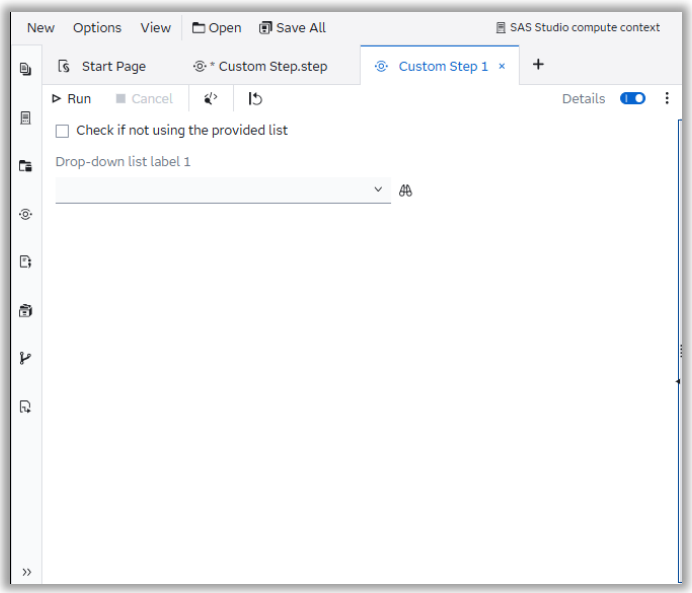

Figure 7: User View - Unchecked Option

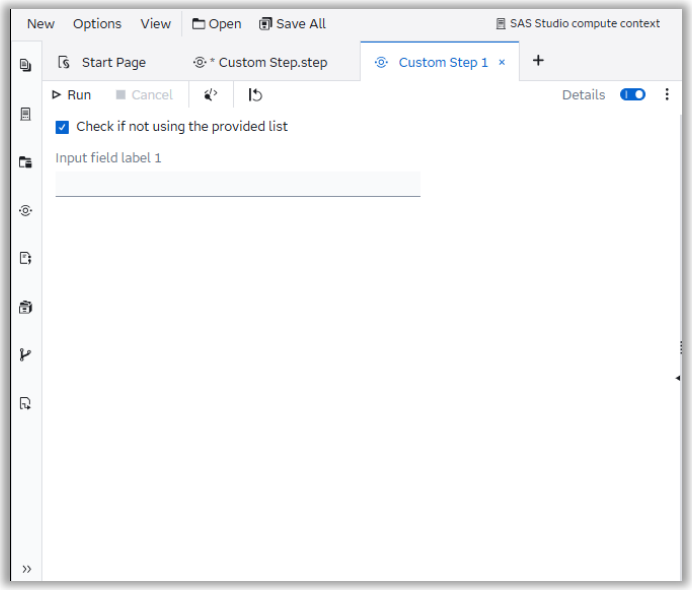

Figure 8: User View - Checked Option

Dependency Syntax

The best reference for understanding the syntax used for dependencies is the “What are Dependencies?” page[1] in the SAS product documentation. The sample in this paper utilizes the ability to reference if a prompt ID, or the variable associated with a control object, is blank or contains a value. This is done with “!$promptid” and “$promptid” respectively.

IMPROVING THE USER EXPERIENCE

ENABELING DEBUGGING

Another nice feature is to include a DEBUG parameter that gives you greater control over information that can be useful when troubleshooting code such as putting additional messages in the log or retaining interim datasets that would otherwise be cleaned up by code. If this is created as a global macro variable, it can be leveraged throughout a program and any macros it may call. You may also choose to give related macros a prefix that can be appended to the macro names and any global macro variables. For instance, to create a macro variable for debugging code for a PharmaSUG example, you might name it psg_Debug.

```
/* Set macro variable to control additional messages or interim dataset retention */
%GLOBAL psg_Debug;
%LET psg_Debug = 1;
```

With this global macro variable in place, you can now conditionally execute or skip code that may be useful for debugging:

```
%IF &psg_Debug  %THEN %PUT Preparing for SQL inner join…;
:
:
%IF NOT &psg_Debug. %THEN %DO;
  PROC DATASETS MEMTYPE=DATA LIBRARY=work;
    DELETE tempdata;
  RUN;
%END;
```

CUSTOM WARNINGS AND ERRORS

SAS' ERROR or ABORT statements can be used to stop processing when issues are encountered with a job but they can be bit heavy-handed. SAS allows you to address exceptions in your code and insert custom errors, warnings and notes into your log. This lets you handle issues much more gracefully.

While these features can be used in open code, one of the most powerful applications of this capability is in code intended for use by others such as SAS Macros or Custom Steps and Flows. These messages can help users diagnose and correct issues without requiring them to be able to interpret the SAS code. The primary method is the use of the PUT statement. If the first word in the PUT statement is "ERROR," "WARNING" or "NOTE" followed immediately by a colon or hyphen, SAS' color highlighting will automatically be applied to these messages in the log. In the examples below, we show how a missing required macro parameter can be reported to the user and how a default value will be used in the instance of another macro parameter not being specified:

```
%PUT ERROR: Required dsName parameter not provided;
%PUT NOTE: Optional count parameter not provided; default count=10 will be used.
```

Some prefer to avoid using the words "error" or "warning" in their code to prevent it from being found by simple searches. If a regular expression isn't a workable solution for you, you can use SAS' %STR macro quoting function to disguise the full words:

```
%PUT %STR(ERR)OR: Required dsName parameter not provided;
```

If you simply abort execution as soon as you encounter any issue, you miss an opportunity to be more helpful for your users. For instance, if you have several macro parameters, you could check each of the parameters before ending the processing due to one missing required parameter. To do this, you can create local macro variables that can be set if any issues are encountered and then use those to control how the remainder of the code is addressed. In the example below, we create a variable named error and set it to 0. Then, if we encounter any issues with any of the macro parameters, we can set the error to 1 and write a custom message to the log. We then continue to test each of the parameters even if we have a concern and perform any necessary clean up and summaries before exiting the code.

```
%MACRO psg_Demo(dsIn=, dsOut=, keyVar=ID);
  /* Set up our local macro variables for error handling */
  %LOCAL error;
  %LOCAL note;
  %LET error = 0;
  %LET note = 0;

  /* Test each of our macro parameters */
  %IF %LENGTH(&dsIn) EQ 0 %THEN %DO;
    %LET error = 1;
    %PUT ERROR: [psg_Demo] Required dsIn parameter for input dataset name not provided;
  %END;
  %IF %LENGTH(&dsOut) EQ 0 %THEN %DO;
    %LET error = 1;
    %PUT ERROR: [psg_Demo] Required dsOut parameter for output dataset name not provided;
  %END;
  %IF %LENGTH(&keyVar) EQ 0 %THEN %DO;
    %LET note = 1;
    %PUT NOTE: [psg_Demo] keyVar parameter not provided; default variable ID will be
used;
  %END;

  /* If errors identified in parameters, skip body of macro */
  %IF &error. EQ 0 %THEN %DO;
    <Remaining macro code>
  %END;
  %ELSE %DO;
    DATA _NULL_;
      ERROR "ERROR" [psg_Demo] Errors noted above during psg_Demo macro execution.";
    RUN;
  END;
```

For production macros, you would want to include additional error handling to further guide your users. For example, if a parameter specifies an input dataset, you would want to check to make sure that dataset exists. If an output dataset is specified as a parameter, you would want to make sure the location exists. If a parameter is the name of a variable in a dataset, you would want to check to make sure that variables exists in the referenced dataset and, perhaps, even check the data type or if the value is within an expected range.

CONCLUSION

While at the initial pass Custom Steps appear to be a simple UI for macros, upon deeper inspection they can provide a more complex and dynamic user experience that not only helps guide users more effectively but provide a more dynamic foundation for designers to decrease overall maintenance efforts. The joy of Custom Steps is that the decision to stay simplistic or go more dynamic is a choice. The designer can pick and choose and mix when to keep things simple and when to “over-engineer.” The main things to keep in mind is maintenance and user option overload. It’s a balance that hopefully the tools provided in this paper will make easier.

REFERENCES

1. SAS Institute (Retrieved 2026, March 15). *What are Dependencies?* SAS Product Documentation. https://go.documentation.sas.com/doc/en/sasstudiocdc/default/webeditorcdc/webeditorsteps/n15hygtd9nd7kun0z3isz8spd93w.htm

RECOMMENDED READING

Queen, MK (2023, December 06). *SAS Viya: Control Dependencies in a Custom Step*. SAS Communities Library. https://communities.sas.com/t5/SAS-Communities-Library/SAS-Viya-Control-Dependencies-in-a-Custom-Step/ta-p/906509

Brown, A (2020, October). *Custom Errors, Warnings, and Notes in SAS®, Plus a Somewhat Practical Use of PROC EXPLODE.* https://sesug.org/proceedings/sesug_2020_final_papers/Know_Your_SAS__Foundations/SESUG2020_Paper_110_Final_PDF.pdf

Rosenbloom, M and Lafler, K (2013). *Best Practices: PUT More Errors and Warnings in My Log, Please!* https://support.sas.com/resources/papers/proceedings13/350-2013.pdf

CONTACT INFORMATION

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

Mary Dolegowski
SAS Institute
Mary.Dolegowski@sas.com
https://www.linkedin.com/in/maryliang/

S. Robert Collins
SAS Institute
Robert.Collins@sas.com
https://www.linkedin.com/in/srobertcollins/

AI Disclaimer: Microsoft Copilot was used to proofread the content and suggest changes.