

ODS: The DATA Step Knows

William F. Heffner, SAS Institute Inc., Cary, NC

INTRODUCTION

The Output Delivery System (ODS) in Version 7 implements and controls the formatting of all SAS procedure output. Although primarily created for procedure output, this new functionality is also available for DATA step programs. This paper will first briefly describe the Output Delivery System, and then discuss in more detail the DATA step interface to this new SAS System component.

ODS: OUTPUT DELIVERY SYSTEM

The Output Delivery System in Version 7 is a new sub-system which all SAS procedures use to produce output. In previous versions of the SAS System all procedures wrote exclusively to the SAS listing file and to output data sets. In Version 7, all procedures produce ODS *output objects* – binary objects that are rendered to various output destinations by the ODS sub-system.

Conceptual Model

In Version 6 of the SAS System, procedures produced output by filling a “line buffer” and then flushing it to the SAS listing file. This approach does not allow proportional font support, nor does it allow access to data values in the listing without rereading the listing. More importantly, there was also no way to upgrade the system to support new output forms as they became available. Each procedure was wed to its own formatting code. ODS was conceived as a better way for procedures to produce output. In Version 7, procedures produce a binary representation of output called an *output object*.

An output object consists of two component parts: a *data object* containing the raw data values for the piece of output, and a *template* describing how the piece of output should look. The output object is added to the system and ODS decides, based on the output destinations set by the user, how to render the output.

The most important benefit of this approach is that it removes the formatting responsibility from the procedure. This enables the SAS System to support many flavors of output without modifying the procedures to support them directly.

Formatting of Output

Each output object that is produced by a run of the SAS System is rendered to whatever output destinations the user has selected. For the initial release of ODS, the SAS listing file, output data sets, and HTML will be supported as destinations. For subsequent releases, support for PS (PostScript) and PCL (Printer Control Language) destinations, an RTF (rich text format) destination, and a persistent output document destination are being investigated. Each output destination can be controlled by the **ODS** global statement. Multiple output destinations can be active at the same time, i.e., one step can create one output object which is rendered to multiple destinations.

ODS Data Object

The ODS data object is the vehicle that a procedure uses to move the data for a piece of output to the ODS system. The data object is primarily an internal object over which the SAS user has no control. However, the SAS user has the flexibility to convert all ODS data objects into SAS data sets.

Templates

A template is a description of how you would like a piece of output to look when it is rendered. Templates contain formatting information like data column order, text for data column headings, format specifications for columns, and stylistic references. Every template in the SAS System is fully editable via the **TEMPLATE** procedure in batch, and from the SAS Explorer in DMS mode.

ODS Global Statement

The **ODS** global statement is a new statement in Version 7 that gives users some control over ODS. The ODS statement contains sub-statements which control each of the various output destinations, select or exclude individual pieces of output, and manipulate the SAS template concatenation path.

DATA STEP ODS INTERFACE

The DATA step can be used to generate output through ODS, taking advantage of all the output destinations available, including the standard listing file and HTML-based output. This requires creating an ODS template and data object, ‘binding’ them together, and specifying the target destination. Templates are defined, created and stored with the **TEMPLATE** procedure. The target output destination is specified by the **ODS** global statement.

A data object can be created and bound to a template using the DATA step. The DATA step interfaces with the ODS system component through its external file output interface. Specifically, the **FILE** statement is used to define a data object and bind a template to it, creating an output object. The **PUT** statement is used to write data to the object (in the form of data rows).

This output object, created by the DATA step, is then processed by the ODS component of the SAS System. ODS appropriately formats the data, and sends it to the currently specified target destination(s).

DATA STEP: CREATING ODS DATA OBJECTS

The DATA step **FILE** statement now supports an **ODS=** option to enable the creation of an ODS data object. This option allows you to create a data object by specifying data columns and associated attributes, and ‘mapping’ them to specific DATA step variables. The mapped data step variables will supply values for the associated data columns.

Generally, the FILE statement ODS= option will have the following form:

```
FILE PRINT ODS <= ( < TEMPLATE = 'template.name' >
                   < COLUMNS = ( column_spec ) >
                   < VARIABLES = ( variable_spec ) >
                   < OBJECT = name >
                   < OBJECTLABEL = 'object label' >
                   < GENERIC = ON | OFF >
                   < LABEL = 'default column label' >
                   < DYNAMIC = ( dynamic_spec ) >
                   ) >;
```

...where *column_spec* can be one or more of the following:

```
column_name <= variable_name > < ( column_attrs ) >
```

...where *variable_spec* can be one or more of the following:

```
variable_name <= column_name > < ( column_attrs ) >
```

...where *column_attrs* can be one or more of the following:

```
FORMAT = formatw.d
GENERIC = ON | OFF
LABEL = 'column label'
DYNAMIC = ( dynamic_spec )
```

...where *dynamic_spec* can be one or more of the following:

```
dynamic_name <= variable_name >
dynamic_name <= constant_value >
```

Only one COLUMNS= or one VARIABLES= sub-option may be given per ODS= specification. See [Mapping DATA Step Variables to Data Object Columns](#) below. Note that the ODS= option may only be used when the FILE statement references the fileref PRINT (i.e., FILE PRINT ODS= ...;). Note also that the ODS keyword may be used alone, without any sub-options (i.e., FILE PRINT ODS;). This default behavior is described in [Default ODS Option Behavior](#) below.

Defining the Template

An ODS template is specified via the TEMPLATE= sub-option of the ODS= option. A template contains column and column-heading information for the output object, as well as physical formatting specifications. The template name is a 1, 2, or 3 level name (e.g., 'aaa', 'aaa.bbb', or 'aaa.bbb.ccc'), and must be a quoted string constant value. This name is passed to ODS, which will search for the template in the locations specified in the **ODS** global statement.

If no template name is given in the ODS= option, a default template ('base.datastep.table') is used.

Defining the Data Object

An ODS data object consists of rows and columns of raw data, much like a SAS data set. The data object is created by the DATA step, using the column names specified by the COLUMNS= sub-option of the ODS= option. The data object column name should match the name of a defined column in the template. See [Mapping DATA Step Variables to Data Object Columns](#) below for more information. The order of the column specifications in the COLUMNS= sub-option defines the order of the columns in the data object.

Column names must conform to the same rules as Version 7 variable names. The first character must be a letter (A, B, C, ..., Z) or an underscore (_), and subsequent characters can be letters, digits (0, 1, ..., 9), or underscores. Variable list notation, of the form 'col1-col5', may be used in the COLUMNS= sub-option to indicate a range of columns.

A name can be specified for a data object with the OBJECT= sub-option. The name must conform to the rules for SAS variable names. A descriptive label can also be added to a data object by specifying a quoted string constant value with the OBJECTLABEL= sub-option. These identifiers (name and label) are visible using the SAS Explorer. If not specified, the data object name defaults to 'FilePrintn', where 'n' increments for each DATA step run during the current SAS session. The object label will default to the currently defined title, unless it is the default SAS System title. In that event, the object name will also be used for the object label.

Defining Attributes for the Data Object

Other attributes (aside from the name and label) may be specified for ODS data objects. These attributes are either default settings for all columns, or global settings for the data object. A default column label may be specified with the LABEL= sub-option. This label would be applied to any column which had no specific label specified.

By default, only one data object column may be mapped to each template column. This one-to-one mapping is accomplished by matching names. However, this one-to-one mapping can be bypassed using the **generic** attribute. If a template column is marked as generic, then more than one data object column can be mapped into it. These data object columns must also be marked as generic, which is done via the GENERIC= sub-option.

The GENERIC= sub-option specifies whether or not one or more columns in the data object are considered to be generic. By default, columns are considered to be **not generic** (i.e., GENERIC=OFF). Specifying GENERIC=ON as an ODS= sub-option causes all columns in the data object to be generic. This default setting can be overridden on a per-column basis, using GENERIC= as a column attribute in the COLUMNS= sub-option. See [Defining Attributes for Data Object Columns](#) below for more discussion.

A powerful feature of ODS templates is the use of dynamic scalar and value pairs. Templates can specify substitution names, called dynamic scalars. These substitution names are used in the template, instead of specific values for various attributes. At runtime, values can be specified for these dynamic scalars. This allows the dynamic specification of various user defined information, instead of 'hard-coding' the information in the template.

One or more of these dynamic scalar / value pairs can be specified using the DYNAMIC= sub-option. The 'scalar' names should match the template-defined dynamic substitution name. The 'value' should be a constant value or a DATA step variable name. (See [Using Custom Templates](#) in the [EXAMPLES](#) section below for complete examples.)

If a variable name is specified for a dynamic scalar value, then the value of the DATA step variable is passed to the data object at runtime. ODS will substitute this value in place of the scalar (or substitution) name in the template definition when formatting the output. The data type required for the value is governed by the dynamic template attribute being set. Attributes expect either

numeric, character, or Boolean data types. ODS makes an attempt to convert any scalar value whose type does not match the required type. Boolean values in the template have values of ON and OFF. DATA step values of 1 and 0, respectively, correspond to these settings.

Mapping DATA Step Variables to Data Object Columns

As previously discussed, a data object and its columns are defined by the COLUMNS= sub-option. In order to populate this data object, values of DATA step variables (from the program data vector) will need to be moved into the data object columns. These DATA step variables need to be mapped to specific data object columns. This specification is done via the COLUMNS= sub-option, as well.

The COLUMNS= sub-option takes a list of column specifications. These column specifications can be as simple as a list of column names, with no other information (e.g., COLUMNS=(c1 c2 c3)). In this case, the specified column names will be mapped to DATA step variables of the same name. If those variable names do not exist in your DATA step program, then an error will be generated.

Variable names can be explicitly mapped to data object columns with the COLUMNS= sub-option. Follow each column name in the COLUMNS= list with an equal sign (=) and the DATA step variable name which should be mapped to that data object column. In the example below, the columns *col1*, *col2*, and *col3* are mapped to the variables *first_name*, *last_name*, and *home_address*, respectively.

```
data _null_;
  file print ods = ( template='user.mine.a'
                    columns = ( col1 = first_name
                               col2 = last_name
                               col3 = home_address ) );
  infile myinfo;
  input first_name $10. last_name $15. home_address $30.;
  ...
run;
```

Alternatively, the VARIABLES= sub-option of the ODS= option may be used to define the data object columns and their DATA step variable mappings. With VARIABLES=, the specified list contains DATA step variable names (with optional equal sign and column name for an explicit mapping), as opposed to the column names in the COLUMNS= sub-option. This is the only difference between these two sub-options. Use of one or the other is according to preference. The following example is functionally equivalent to the previous example.

```
data _null_;
  file print ods = ( template='user.mine.a'
                    variables = ( first_name = col1
                                 last_name = col2
                                 home_address = col3 ) );
  infile myinfo;
  input first_name $10. last_name $15. home_address $30.;
  ...
run;
```

If no explicit column-to-variable mappings are needed, the two sub-options are identical. The specification VARIABLES=(c1 c2 c3) is equivalent to COLUMNS=(c1 c2 c3). Both would map the DATA step variables *c1*, *c2*, and *c3* into data object columns *c1*, *c2*, and *c3*.

Note that variable list notation (*var1-var5*) is supported for both column names and DATA step variable names, in both the COLUMNS= and VARIABLES= sub-options. The example below maps the variables *var1*, *var2*, *var3*, *var4*, and *var5* to the columns *col1*, *col2*, *col3*, *col4*, and *col5*.

```
data _null_;
  file print ods = ( template='user.mine.b'
                    columns = ( col1-col5 = var1-var5 ) );
  infile myinfo;
  input var1 - var5.;
  ...
run;
```

Either the COLUMNS= or the VARIABLES= sub-option may be used in an ODS= specification, but not both. Using both will generate an error. If no COLUMNS= or VARIABLES= sub-option is specified, then all DATA step variables are defined as data object columns, and those names that match template column names will be included in the output.

Defining Attributes for Data Object Columns

Attributes can be defined for individual data object columns. These column attributes are specified inside of the COLUMNS= (or VARIABLES=) sub-option. (See ODS= syntax above.) These specifications will override any defaults for a particular column. Any attributes which may be specified for the data object (LABEL=, GENERIC=, and DYNAMIC=) may also be specified for individual columns. Additionally, a format may be attached to individual columns with the FORMAT= sub-option.

The LABEL= sub-option assigns a descriptive label to the specified column. The primary use of a column label would be as a default heading when formatting the column, if no heading is specified in the template column definition itself. If no label is specified for a column, the mapped variable's label value would be used. If no label is specified, and no label for the mapped variable exists, then the variable name is used as for the column heading.

Specify GENERIC= to change a particular column's generic attribute from the data object default. The data object default is determined by any GENERIC= setting for the data object. (See [Defining Attributes for the Data Object](#) above.) If it has not been set explicitly, the data object default is GENERIC=OFF (i.e., columns are not generic). Use GENERIC= inside the COLUMNS= (or VARIABLES=) sub-option to set this attribute for specific columns.

Use DYNAMIC= inside a COLUMNS= (or VARIABLES=) sub-option to specify dynamic scalar / value pairs for particular columns. A dynamic scalar value set for a specific column will override a dynamic value for the same scalar set for the entire data object. As with data object attributes, the scalar value for column attributes can be a constant value or a DATA step variable name. In addition, a special indicator of `_LABEL_` can be used as a column attribute scalar value. This specifies that the label defined for the column-mapped DATA step variable be used as the scalar value.

The FORMAT= sub-option can be used to attach a format to the data object column. The format would be used by ODS when formatting the column output. Note that this column attribute format does not override a format specification for the column in the template. This format will only be used if there is none specified in the template definition.

The following FILE statement demonstrates several uses of column attributes. (See [Using Custom Templates](#) in the [EXAMPLES](#) section below for complete examples, including template definition and generated output.) The following is assumed about the template 'user.mine.c'

- It has been defined, stored, and is accessible through use of the TEMPLATE procedure and ODS global statement.
- At least three columns are defined named *c1*, *cg*, and *c4*, of which *cg* has the generic attribute.
- Two dynamic attributes are defined: *chdr*, which specifies the heading to use for column *c3*, and *mywid*, which specifies the column width.

```
label sales = 'Total Sales';
file print ods =
  ( template = 'user.mine.c'
    dynamic = (mywid = 12)
    columns = ( c1 = name
               (label = 'SalesPerson')
               cg = loc
                 (generic = on
                  dynamic = (chdr = 'Location'))
               cg = sold_contracts
                 (generic = on
                  format = 3.
                  dynamic = (chdr='Contracts'
                           mywid=4))
               c4 = sales
                 (format = dollar11.2)
    )
  );
```

The output will have four columns, consisting of values from the DATA step variables *name*, *loc*, *sold_contracts*, and *sales*. The variables *name* and *sales* are mapped to columns *c1* and *c4*, respectively, in the template. The heading for the column *c1* will be 'SalesPerson' (the label given as a column attribute), and the heading for the column *c4* will be 'Total Sales' (the label associated with the variable *sales*). Both the variables *loc* and *sold_contracts* are mapped to the generic column *cg* in the template. Both specify a dynamic value for the attribute *chdr*. This value will be the heading for these columns. A dynamic value is also specified for *width* (4) for *sold_contracts*, which overrides the data object dynamic specification for *width* (12). The *sold_contracts* column width will be 4, while all the other columns will be width 12. Formats are specified for the *sold_contracts* column (3.) and the *sales* column (dollar11.2).

Default ODS Option Behavior

If no sub-options are specified for the ODS= option, the DATA step will take a set of defaults. A default template for the DATA step ('base.datastep.table') has been created and stored in the SASHELP SAS data library. This template defines two generic columns – one column specification maps to numeric variables, and the other to character variables. The template causes all data object columns to be output, in the same order as they are defined on the data object.

This order, as previously discussed, is controlled by the order of the column specifications in the COLUMNS= (or VARIABLES=) sub-option. If no columns are specified, then all DATA step program variables will be added to the data object, in the same order as the DATA step 'sees' the variables.

The following two FILE statements are equivalent. Note that use of the default template forces a default of GENERIC=ON for all

columns, so the GENERIC specification is superfluous in the second FILE statement.

```
file print ods;

file print ods = ( template = 'base.datastep.table'
                  generic = on
                  );
```

You can use the default template in conjunction with other ODS= sub-options. Since by default, all variables are added to the data object, a very common usage of the default template would include a VARIABLES= specification to subset the output columns. The following FILE statement defines a data object with 5 columns, and will use the default DATA step template.

```
file print ods = (variables = ( x1 - x5 ));
```

Interactions with Other FILE Statement Options

The DATA step ODS interface uses the FILE and PUT statements. Even though this interface mimics the standard external file output interface, many FILE statement options are not always applicable. Some have no use because ODS is not list oriented – it has defined data columns. The DSD and DELIMITER= options are list oriented, and therefore have no effect. Others don't make sense because the physical file being written is controlled by ODS, not the DATA step. This type of option includes FILEVAR=, EXPANDTABS, and PAD.

The HEADER= option is not supported because it assumes a level of control over the physical output page that is not available when using the ODS interface. The _FILE_= option (new for Version 7) allows access to the output data buffer directly. But for ODS, this buffer contains raw unformatted data whose order is determined by alignment and space requirements, and is not known to the user. Therefore, the option is not supported.

There are a few options whose effects are contingent upon the ultimate target destination, which is controlled by the ODS global statement. Some are page oriented, and therefore only apply when the final formatting includes page information. The PAGE-SIZE= and LINESIZE= options all fall into this category. For example, the LISTING target of the ODS statement is the listing file, and it supports the concept of pages. However, the HTML target is a destination which does not have pages, and is unaffected by these options.

DATA STEP: WRITING ODS DATA OBJECTS

An ODS data object consists of data columns and data rows. The FILE statement, with an ODS= option, defines the data columns (and the mapped DATA step variables). The PUT statement creates data rows to be written to the data object. In the simplest form, the PUT statement directs the previously mapped variables be moved into an output buffer and written to the data object, thus creating a data row. More complex forms of the PUT statement may be used, however, to exercise more control over this processing. Particular data object columns can be accessed for a data row, temporarily overriding the defined column-variable mappings. Multiple data rows can also be accessed.

Moving Data into the Data Object

For standard external file output, the PUT statement moves data values into an output data buffer. At some point (usually at the end of PUT statement execution), this formatted buffer is written

to the external file. Processing for ODS data objects is similar. The PUT statement moves data values into an output buffer, which eventually will be written to the data object. Each output buffer written to the data object is considered to be a data row.

One difference between the two forms of processing involves the data in the output buffer. For standard output, the data values are formatted into the output buffer (i.e., any associated format is applied to the data before arriving in the buffer). However, in data object processing, raw unformatted data is written to the data buffer. Formatting information is 'remembered' by the data object, but is not applied to the data until ODS renders output to the final destination.

A new DATA step language element has been added to the PUT statement to facilitate writing to an ODS data object. Instead of requiring specification of the column-mapped variables in the PUT statement, a new keyword, `_ODS_`, can be used. This instructs the PUT statement to move data values for all columns (as defined in the ODS= option) from their program data vector locations into the output buffer. For the following FILE statement, the two PUT statements are equivalent.

```
file print ods = ( variables = ( first_name
                               last_name
                               home_address ) );
put first_name last_name home_address;
put _ods_;
```

A null PUT statement (i.e., a PUT statement with no variables or operators) causes the current output buffer to be written. Since there is no movement of values to the buffer, this would write a data row containing all missing values to the data object.

Accessing Data Object Columns

Generally, a FILE statement with an ODS= specification and a PUT statement with `_ODS_` will suffice. This allows the mapping of DATA step variables to a template specification, and the writing of those variables' values into the data object for processing by ODS. But there are situations where more flexibility is needed; more control over the data object is required from the PUT statement. Conceptually, this flexibility and control is supplied by the column pointer.

For standard external file output, the column pointer indicates the current column location (in terms of characters) in an output buffer. The next item to be formatted into the buffer will begin at this column location. This concept has been carried over into ODS processing, except that the column location is in terms of data object columns instead of characters. The column ordering is defined by the column order of the data object, which is defined by the column order specified in the COLUMNS= or VARIABLES= sub-option. The relative position of any variables specified on the PUT statement will be matched with the columns in the same relative position in the data object.

This matching by relative position allows you to override the specified definition for a column in a PUT statement by writing the value of some arbitrary variable to a particular column. This override is temporary, and lasts only for the duration of that PUT statement. The following 'business phone' example demonstrates this functionality. If no value for `business_phone` is available, then `home_phone` is used for the column in position three.

```
file print ods = ( variables = ( first_name
                               last_name
                               business_phone ) );
if (missing(business_phone)) then
  put first_name last_name home_phone;
else
  put _ods_;
```

The column pointer controls can be used to position to specific data object columns. Relative positioning ('+' operator) supports a numeric expression, which is added to the current column pointer to obtain a new column location. Absolute positioning ('@' operator) supports both numeric and character expressions. A numeric expression changes the current column pointer to the specified numeric value. A character expression result is used to match a defined column name, and the column pointer is set accordingly. For the following FILE statement, all of the PUT statements are equivalent.

```
file print ods = ( variables = ( one two three ) );
put _ods_ ;
put one two three;
put @1 one @2 two @3 three;
put @2 two @1 one +1 three;
put +2 three @2 two +(-2) one;
put @'two' two @1 one +1 three;
```

The `_ODS_` specification can be used in a PUT statement in conjunction with variable specifications and column pointer controls. In this way, one column may be overridden, without needing to specify all columns on the PUT statement. The location of the column pointer after an `_ODS_` specification is one greater than the maximum column number defined. So, if variable specifications follow `_ODS_` in the PUT statement, a column pointer control will be needed to reposition to the desired column. The PUT statement from the previous 'business phone' example can be changed from

```
put first_name last_name home_phone;
```

to the shorter

```
put _ods_ @3 home_phone;
```

The `_ODS_` specification can appear anywhere in a PUT statement. However, note that `_ODS_` causes data to be moved into the output buffer for specific columns **only** if a PUT statement has not already moved data into that column. In other words, if a PUT statement specifically moves a value into the buffer for a column via a variable specification, an `_ODS_` appearing later in the PUT statement will not overwrite that buffer location. For example, the following two PUT statements (from the 'business phone' example) create equivalent data rows.

```
put _ods_ @3 home_phone;
put @3 home_phone _ods_;
```

In the first PUT statement, the `_ODS_` specification moves values for `first_name`, `last_name`, and `business_phone` into the buffer. It then positions to column 3, and explicitly overwrites the value of `business_phone` with the value for `home_phone`. In the second PUT statement, we first position to column 3 and move the value for `home_phone` into the buffer. Then, the `_ODS_` specification will cause the values for `first_name` and `last_name` to be written to the buffer. But the `home_phone` value in the buffer will not be overwritten, because the PUT statement has already explicitly moved a data value into that column position.

The second PUT statement would actually execute slightly faster, because some unneeded data movement is avoided.

Accessing Data Object Rows

The flexibility and control to access multiple data object rows is also available. This is done via the FILE statement N= option and line pointer controls. The trailing @ can also be used to hold the buffer for the next PUT statement.

The relative positioning operator (/) will move the line pointer to the next data row. If N=1, then the contents of the output buffer is written to the data object. If N>1, then the next buffer in the N= group of buffers is considered 'current'. No buffers are written to the data object until the line pointer exceeds the N= setting. When this happens, all buffers in the N= group are written to the data object. As with standard external file output, once a buffer is released to be written, it is no longer accessible from the DATA step program. Therefore, data rows cannot be modified once written to the data object.

The trailing @ is used to hold an output data buffer, so that the next PUT statement can act on that buffer. This could be useful when a small subset of columns need to be overridden in a few data rows. The 'business phone' example could be re-written again.

```
file print ods = (variables = ( first_name
                              last_name
                              business_phone ) );
if ( missing(business_phone) ) then
  put @3 home_phone @;
put _ods_;
```

As with standard external file output, it is possible in ODS processing to specify too many items in a PUT statement. More variables could be specified on the PUT statement than columns were defined in the ODS= option. When this happens, the behavior depends on the current end-of-record setting (FLOWOVER, STOPOVER, or DROPOVER). The default is FLOWOVER – the current output buffer is written to the data object, and the 'extra' variable specifications are written to a new buffer. STOPOVER causes the DATA step to stop processing with an error. DROPOVER allows processing to continue, just ignoring the 'extra' variables.

EXAMPLES

The following set of examples begins with a simple DATA step using default settings, and progresses through more complicated scenarios with user-written, customized templates. These examples assume that the output destination is the SAS listing file (which is the default). But, the examples would also work for HTML destinations. The data used for all examples is identical, and is not repeated after the first example.

Using the Default Template

The first example uses all defaults for the ODS= option. Recall that this will use the default template (base.datastep.table), and write all variables in the program data vector to the output destination. The associated listing output is Output 1. Note that all columns use variable names for headings, except for the column associated with the variable *sales*. Since a label has been specified for that variable, the label is used as the column heading.

```
title 'XYZ Corporate Sales Information';
data _null_;
  input name $10. dept $4. loc $10. sold_contracts sales;
  label sales = 'Total Sales';
  file print ods;
  put _ods_;
datalines;
ZRay      CDD Atlanta   15  45000
CRabb     CDS Charlotte 25  101500
AJonesboro SGA St Louis 12  42700
CHauser   SBC Nashville 22  130150
EJohnson SBC Nashville 10  35500
;
run;
```

Output 1

XYZ Corporate Sales Information				
name	dept	loc	sold_contracts	Total Sales
ZRay	CDD	Atlanta	15	45000
CRabb	CDS	Charlotte	25	101500
AJonesboro	SGA	St Louis	12	42700
CHauser	SBC	Nashville	22	130150
EJohnson	SBC	Nashville	10	35500

If not all variables are needed in the output, you can specify exactly what variables are needed. The next example subsets the variables written to the data object, using the VARIABLES= sub-option. Output 2 does not contain the column associated with the variable *loc*.

```
title 'XYZ Corporate Sales Information';
data _null_;
  input name $10. dept $4. loc $10. sold_contracts sales;
  label sales = 'Total Sales';
  file print ods = ( variables = ( name
                                dept
                                sold_contracts
                                sales
                              ) );
  put _ods_;
datalines;
...
;
run;
```

Output 2

XYZ Corporate Sales Information			
name	dept	sold_contracts	Total Sales
ZRay	CDD	15	45000
CRabb	CDS	25	101500
AJonesboro	SGA	12	42700
CHauser	SBC	22	130150
EJohnson	SBC	10	35500

Since these output listings are roughly equivalent to that available from the PRINT procedure, there is really not much utility in using the default template (except to avoid the procedure step). The power and flexibility of the DATA step's interface to ODS is in the use of custom templates.

Using Custom Templates

User defined customized templates can be built with the TEMPLATE procedure. These templates are stored in SAS data libraries, and can be easily accessed by many users. Template definitions may contain extensive formatting and output information, or they may contain very little. The amount of information contained in a template depends on its intended usage and audience. If a template is to be used for a specific application only, it may be desirable to include very specific formatting information. If a template is to be used more as a general form for many reports for many different users, then it should be less specific, and use some of the generic and dynamic facilities of templates.

The complete syntax for PROC TEMPLATE has not been included in this paper – it is extensive. But some examples of its usage must be included to demonstrate the DATA step interface. The constructs used in these examples should be fairly self-explanatory. The template name is specified by the DEFINE TABLE statement. Template columns are specified by the COLUMN statement. Column attributes are specified by the DEFINE statement. The WIDTH=, HEADER=, JUST=, and FORMAT= options (in the DEFINE statement) specify the width, heading, justification, and format, respectively, for a column. Scalar names used for dynamic substitutions at runtime are defined by the DYNAMIC statement.

The following example defines and stores a template which contains very specific formatting information. Column names and headings are defined which are specific to this particular usage. The ensuing DATA step creates the data object and associated listing output, detailed in Output 3.

```
/* Define the template 'user.mine.a' */
proc template;
define table user.mine.a;
  column name dept sold_contracts sales;
  define name;
    width=12
    header='SalesPerson';
  end;
  define dept;
    width=12
    just=c
    header='Department';
  end;
  define sold_contracts;
    width=4
    format=3.
    just=c
    header='Contracts';
  end;
  define sales;
    width=12
    format=dollar11.2 ;
  end;
end;
run;
title 'XYZ Corporate Sales Information';
/* Create output object and listing with DATA step */
data _null_;
  input name $10. dept $4. loc $10. sold_contracts sales;
  label sales = 'Total Sales';
  file print ods = ( template='user.mine.a' );
  put _ods_;
datalines;
...
```

```
;
run;
```

Output 3

XYZ Corporate Sales Information

SalesPerson	Department	Contracts	Total Sales
ZRay	CDD	15	\$45,000.00
CRabb	CDS	25	\$101,500.00
AJonesboro	SGA	12	\$42,700.00
CHauser	SBC	22	\$130,150.00
EJohnson	SBC	10	\$35,500.00

The template 'user.mine.a' is very specific to this 'sales' DATA step. It may be difficult to reuse it for other applications. You can make the template more flexible by using **dynamic** substitutions for some of the information, and removing any 'hard-coded' information. Dynamic values for these substitutions can be supplied by the DATA step through its ODS interface.

The template in the next example ('user.mine.b') uses dynamic substitutions for the column width (*mywid*) and the column heading (*chdr*). Values for these dynamic substitutions are specified in the DATA step FILE statement ODS= option. This template could now more easily be used in other applications, because it no longer contains information specific to the 'sales' DATA step.

In the ensuing DATA step, a default value for *mywid* (12) is specified. This value is used as the width for all columns, except for column *c3*, where the value of *mywid* is overridden (4). A value for *chdr* is also specified as a column attribute of columns *c2* and *c3*. The format specifications for columns *c3* and *c4* have also been moved from the template definition into the DATA step.

The output from this example is identical to Output 3.

```
/* Define the template 'user.mine.b' */
proc template;
define table user.mine.b;
  column c1 c2 c3 c4;
  dynamic mywid chdr;
  define c1;
    width=mywid;
  end;
  define c2;
    width=mywid
    just=c
    header=chdr;
  end;
  define c3;
    width=mywid
    just=c
    header=chdr;
  end;
  define c4;
    width=mywid;
  end;
end;
run;
title 'XYZ Corporate Sales Information';
/* Continued ... */
```

```

/* Create output object and listing with DATA step */
data _null_;
  input name $10. dept $4. loc $10. sold_contracts sales;
  label sales='Total Sales';
  file print ods = ( template='user.mine.b'
    dynamic = ( mywid = 12 )
    columns = ( c1 = name
      (label = 'SalesPerson')
      c2 = dept
      (dynamic =
        (chdr = 'Department'))
      c3 = sold_contracts
      (format = 3.
        dynamic = (chdr = 'Contracts'
          mywid = 4 ))
      c4 = sales
      (format = dollar11.2)
    )
  );
  put _ods_;
  datalines;
  ...
  ;
run;

```

At this point, it would be easy to generate slightly different output using the same 'user.mine.b' template by mapping a different DATA step variable to one of the columns. In the following example, the variable *loc* is mapped into column *c2*, instead of *dept*. The new output is Output 4.

```

title 'XYZ Corporate Sales Information';
/* Create output object and listing with DATA step */
data _null_;
  input name $10. dept $4. loc $10. sold_contracts sales;
  label sales='Total Sales';
  file print ods = ( template='user.mine.b'
    dynamic = ( mywid = 12 )
    columns = ( c1 = name
      (label = 'SalesPerson')
      c2 = loc
      (dynamic = (chdr = 'Location'))
      c3 = sold_contracts
      (format = 3.
        dynamic = (chdr = 'Contracts'
          mywid = 4 ))
      c4 = sales
      (format = dollar11.2)
    )
  );
  put _ods_;
  datalines;
  ...
  ;
run;

```

Output 4

XYZ Corporate Sales Information			
SalesPerson	Location	Contracts	Total Sales
ZRay	Atlanta	15	\$45,000.00
CRabb	Charlotte	25	\$101,500.00
AJonesboro	St Louis	12	\$42,700.00
CHauser	Nashville	22	\$130,150.00
EJohnson	Nashville	10	\$35,500.00

Note that the definitions of the columns *c2* and *c3* in the template 'user.mine.b' are identical. The only reason to define these columns separately is to allow for four columns in the generated output. But since these column definitions are identical, one **generic** column definition could replace them. This generic column could map to more than one data object column, thus allowing any number of columns in the generated output. This would, once again, make the template more flexible while reducing the amount of information in the template definition.

The following example creates the template 'user.mine.c', replacing the *c2* and *c3* column definitions with the **generic** column *cg*. The ensuing DATA step maps the variables *loc* and *sold_contracts* to this generic column.

The output from this example is identical to Output 4.

```

proc template;
  define table user.mine.c;
    column c1 cg c4;
    dynamic mywid chdr;
    define c1;
      width=mywid;
    end;
    define cg;
      generic=on
      width=mywid
      just=c
      header=chdr;
    end;
    define c4;
      width=mywid;
    end;
  end;
run;
title 'XYZ Corporate Sales Information';
/* Create output object and listing with DATA step */
data _null_;
  input name $10. dept $4. loc $10. sold_contracts sales;
  label sales='Total Sales';
  file print ods = ( template='user.mine.c'
    dynamic = ( mywid = 12 )
    columns = ( c1 = name
      (label = 'SalesPerson')
      cg = loc
      (generic = on
        dynamic = (chdr = 'Location'))
      cg = sold_contracts
      (generic = on
        format = 3.
        dynamic =
          (chdr = 'Contracts'
            mywid = 4 ))
      c4 = sales
      (format = dollar11.2)
    )
  );
  put _ods_;
  datalines;
  ...
  ;
run;

```

With the generic column now in place, more columns can be added to the output without modifying the template 'user.mine.c'. Additional columns are mapped to the generic column, *cg*. They appear in the output between columns *c1* and *c4*, in the order they are specified in the COLUMNS= sub-option. The following

DATA step uses the 'user.mine.c' template, and adds the 'Department' column back into the output. The resulting output is Output 5.

```

title 'XYZ Corporate Sales Information';
/* Create output object and listing with DATA step */
data _null_;
  input name $10. dept $4. loc $10. sold_contracts sales;
  label sales="Total Sales";
  file print ods = ( template='user.mine.c'
                    dynamic = ( mywid = 12 )
                    columns = ( c1 = name
                                (label = 'SalesPerson')
                                cg = dept
                                (generic = on
                                 dynamic =
                                   (chdr = 'Department'))
                                cg = loc
                                (generic = on
                                 dynamic = (chdr = 'Location'))
                                cg = sold_contracts
                                (generic = on
                                 format = 3.
                                 dynamic = (chdr = 'Contracts'
                                           mywid = 4
                                           ))
                                c4 = sales
                                (format = dollar11.2)
                              )
                    );
  put _ods_;
datalines;
...
;
run;

```

Output 5

XYZ Corporate Sales Information

SalesPerson	Department	Location	Contracts	Total Sales
ZRay	CDD	Atlanta	15	\$45,000.00
CRabb	CDS	Charlotte	25	\$101,500.00
AJonesboro	SGA	St Louis	12	\$42,700.00
CHauser	SBC	Nashville	22	\$130,150.00
EJohnson	SBC	Nashville	10	\$35,500.00

Note also that not all column definitions in a template need to be used. ODS will attempt to match all columns in the template definition with data columns defined for the data object. Any non-matched column definition in either the template or the data object will be ignored.

In the following example, only the template column definitions for *c1* and *cg* are matched up with columns in the data object. The column *cd* is created in the data object, but since no match is found in the template 'user.mine.c', no data is output for that data column. Likewise, no data object column is defined for *c4*, so that column in the template 'user.mine.c' is ignored. The resulting output is Output 6.

```

title 'XYZ Corporate Sales Information';
/* Create output object and listing with DATA step */
data _null_;
  input name $10. dept $4. loc $10. sold_contracts sales;
  label sales="Total Sales";
  file print ods = ( template='user.mine.c'
                    dynamic = ( mywid = 12 )
                    columns = ( c1 = name
                                (label = 'SalesPerson')
                                cd = dept
                                cg = loc
                                (generic = on
                                 dynamic = (chdr = 'Location'))
                              )
                    );
  put _ods_;
datalines;
...
;
run;

```

Output 6

XYZ Corporate Sales Information

SalesPerson	Location
ZRay	Atlanta
CRabb	Charlotte
AJonesboro	St Louis
CHauser	Nashville
EJohnson	Nashville

ACKNOWLEDGMENTS

The author greatly appreciates the contributions by Chris Olinger for providing the general ODS discussion, by Nancy Agnew and Pauline Leveille for new feature testing, and by Jason Secosky and Nancy Agnew for assistance in reviewing this paper.

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

The author can be contacted at

William Heffner
 SAS Institute, Inc.
 SAS Campus Drive
 Cary, NC 27513
 Phone (919) 677-8000
 FAX (919) 677-4444
 Email saswfh@sas.com