Have it Your Way:
Using the ODS EXCEL Destination with the DATA step Report Writing Interface

Pete Lund, Looking Glass Analytics, Olympia, WA

ABSTRACT

SAS© provides some powerful, flexible tools for creating tabular reports, like PROC REPORT and PROC TABULATE. With the advent of the Output Delivery System (ODS) you have almost total control over how the output from those procedures looks. But, there are still times where you need (or want) just a little more control and that's where the DATA step Report Writing Interface can help.

The Report Writing Interface (RWI) is just a fancy way of saying you're using the ODSOUT object in a DATA step. Method calls on this object allow you to create tables, embed images, add titles and footnotes and more – all from within a DATA step, using whatever DATA step logic you need. Also, all the style capabilities of ODS are available to you so that your DATA step created output can have fonts, sizes, colors, backgrounds and borders to make your report look just like you want. And, great news: you can use the RWI with the ODS EXCEL destination!

This presentation will quickly cover some of the basics of using the ODSOUT object and then walk through some of the techniques to create output in Excel from a DATA step using the RWI with ODS EXCEL destination.

THE BASICS OF RWI

In the past, PUT statements were used in DATA _NULL_ reporting to create reports in a DATA step and, with the introduction of ODS styles, those reports could look very nice. But, the new world of DATA _NULL_ reporting is the Report Writing Interface (RWI) and together with the new ODS EXCEL destination, tables can be defined right in the DATA step code and even produce many different tables in the same spreadsheet. Before our discussion of the Report Writing Interface, please understand that this paper is just to get your interest piqued.

The Report Writing Interface is always going to produce results that are sent to some ODS destination. That being the case, the DATA step containing the RWI code will be inside an "ODS sandwich" – the statements that start and end the file being created. The focus of this paper is creating Excel spreadsheets with RWI, so the ODS sandwich would look something like this:

```
ods excel file="<file location>.xlsx" <style=ODS style> <options()>;

data _null_;
  <data step code>
run;

ods excel close;
```

In the past, we could use the ODS HTML destination, the ExcelXP tagset or the MSOffice2k tagset to get output that would open in Excel. But none of these options actually created Excel files. The ODS EXCEL destination does create native Excel files!

In all the example code that follows a couple assumptions are made (unless noted otherwise to expound on the example):

1. They are all within a DATA step, so no DATA…; or RUN; statements will usually not be shown
2. Often code that has already been shown and discussed will not be repeated

3. All of these examples would be creating Excel (xlsx) files, so the "ODS sandwich" statements will usually not be shown (ODS EXCEL file=…; and ODS EXCEL CLOSE;)

The RWI uses a DATA step object called ODSOUT. There are "methods" (like functions) of that object that will create tables, rows, cells, text, page breaks, lines, etc. To use an ODSOUT object it is first declared and given a name – this only has to be done once in the DATA step and is routinely placed in a conditional section of code:

```
if _n_ eq 1 then
  do;
    declare odsout t();
    <other statements>
  end;
```

This gives us an ODSOUT object named "t" – we'll use that name to reference methods that build our output.

Again, the DECLARE statement only has to be executed once in the DATA step.

Once the object is declared you can call "methods" that perform different tasks. For instance, with our object "t," just a few of the possible methods are:

t.table_start()	- begins a table (there is a table_end method that closes a table)
t.row_start()	- begins a row in that table – you can have as many rows in the table as you want (there is also a row_end method that closes a row)
t.format_cell()	- inserts a cell (column) into that row – you can have as many cells in a row as you want, but each row must have the same number of cells
t.format_text()	- inserts a line of text (not part of a table)
t.title()	- creates a page title (there is also a footnote method that creates a page footnote)

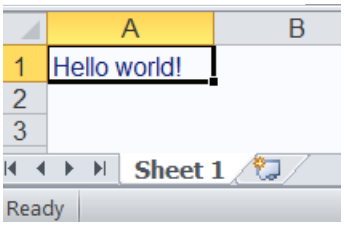
Note that all of these methods calls have parentheses, which are required – even if empty. There are arguments that can be placed in the parentheses. For example, the FORMAT_CELL method has a "DATA" argument that specifies the text to be printed in the cell (note: the "TEXT" argument is equivalent). You can specify style attributes in most method calls as well, specifying cell borders, background colors, font characteristics, etc., with the STYLE_ATTR or STYLE_ELEM arguments. These will be discussed in more detail as we move along.

Here's a very simple example of using the ODSOUT object. No dataset will be read by the DATA step, so wrapping the DECLARE statement in an _N_ = 1 loop is not necessary, because the step will execute only once.

```
data _null_;
  declare odsout t();

  t.format_text(data: 'Hello World);
run;
```

This method simply writes text to the page. By default is will be left justified and use the font attributes associated with the body of the ODS style in effect.

The output from this step would be a single cell table with the words in the row 1, column A. Note that if there are any titles that are still active, these may appear on the page and then text would be in the first row after titles – more about this later.



CREATING SIMPLE TABLES

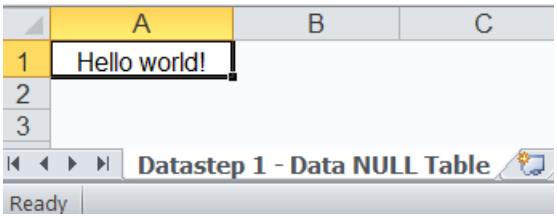
But, reports are usually not comprised of just text. The RWI can define tables of data and there are sets of table, row and cell methods that allow us to do that.

The creation of a table is pretty straightforward:

```
t.table_start();
  t.row_start();
  t.format_cell(data: 'Hello World);
  t.row_end();
t.table_end();
```

- *Start the table*
- *Start the first, and only, row*
- *Insert a single cell with some text*
- *End the row*
- *End the table*

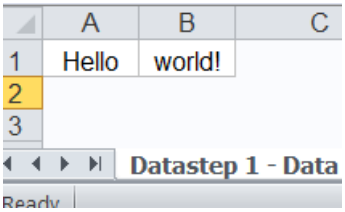
As noted above, the parentheses following the method calls are required, even if no arguments are passed. Also note that the DATA argument in the FORMAT_CELL method call is exactly the same as that in the FORMAT_TEXT method call in the earlier example. But, the appearance of the output is slightly different: the text is now centered in the cell and the tab name is different. We'll look at ways to control both of those things later in the paper.



It's simple to make the tables more table-like, by adding more columns and rows. Getting more columns is just a simple matter of having more FORMAT_CELL calls between the ROW_START and ROW_END. Each FORMAT_CELL call goes into a separate column on the row.

```
t.row_start();
  t.format_cell(data: 'hello');
  t.format_cell(data: 'world');
t.row_end();
```

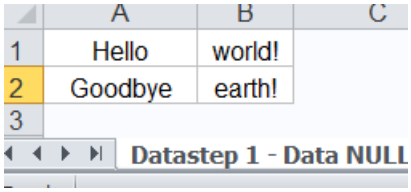
A row can have any number of cells – here we're creating a row with two cells (columns), each with a single word



Getting more rows is just as simple – add as many ROW_START…ROW_END blocks needed between the TABLE_START and TABLE_END. One thing to note – there must be the same number of columns (cells) in each row. We will see later that there are arguments on the FORMAT_CELL method to span rows and/or columns.

```
t.row_start();
  t.format_cell(data: 'hello');
  t.format_cell(data: 'world');
t.row_end();
t.row_start();
  t.format_cell(data: 'goodbye');
  t.format_cell(data: 'earth');
t.row_end();
```

A table can have any number of rows – here we've added a second row to the previous example



Notice that, by default, the widths of the columns are based on the longest text in the column. Later, ways to control lots of attributes of the tables, which could have made these two tables look much the same, will be shown.

DATA-DRIVEN TABLES

It's not too practical to think of hard-coding all the data to be presented in a table. Fortunately, in addition to a quoted string, the DATA attribute of the FORMAT_CELL method shown in the examples above can take a variable or expression as its value. This allows for creation of tables from data in datasets or from variables created in the DATA step. The following examples use the class list from the SASHELP.CLASS dataset.

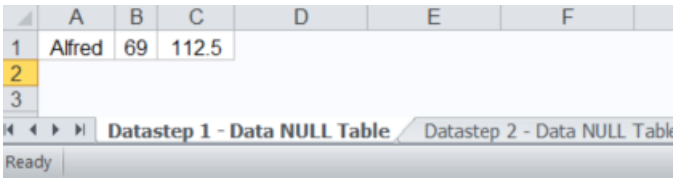
```
set sashelp.class;

if _n_ eq 1 then declare odsout t();

t.table_start();
  t.row_start();
    t.format_cell(data: name);
    t.format_cell(data: height);
    t.format_cell(data: weight);
  t.row_end();
t.table_end();
```

Now, bring in a dataset and we'll use values from that to populate our table

Remember, only need to declare

Now, rather than quoted values in the DATA argument of the FORMAT_CELL method call, put the name of a variable. The contents of that variable will be placed in the cell.

But, there's a slight problem with the above code – the TABLE_START and TABLE_END methods are going to be called for every iteration of the DATA step. By default, in Excel each table is on a separate tab. So, the result is a single row worksheet for every observation, as shown in the snippet above. That might be what is wanted, but probably not.



The solution is simple – place the TABLE_START call in the _N_ eq 1 logic and the TABLE_END call to be triggered conditionally using the variable on an END= option on the SET statement.

```
set sashelp.class end=done;

if _n_ eq 1 then
  do;
    declare odsout t();

    t.table_start();
    t.row_start();
      t.format_cell(data: 'Name');
      t.format_cell(data: 'Height (ins)');
      t.format_cell(data: 'Weight (lbs)');
    t.row_end();
  end;

  t.row_start();
    t.format_cell(data: name);
    t.format_cell(data: height);
    t.format_cell(data: weight);
  t.row_end();

if done then t.table_end();
```

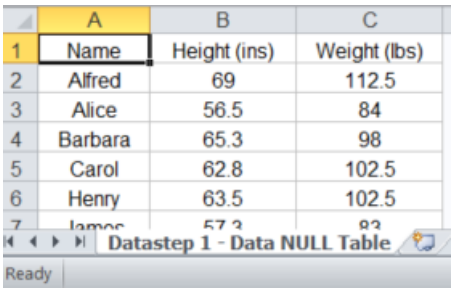
Use the END= option to define a variable that will be set to 1 (true) when the end of the dataset is reached.

In addition to declaring the ODSOUT object, move the TABLE_START call to the _N_ eq 1 block of code. Also, this is a good place to add a single header row to the table. This ROW_START and ROW_END block will only be executed once.

The row and cell code is exactly the same as before – all that needed to be changed was when the table started and stopped. Now each of these rows will be in the same table

When the end of the dataset is reached, end the table.

Those couple simple changes create a single table. There is a header row to tell the reader what's in the table. Now is the time to take a look at to control not just what appears in the table, but how the table appears.



CONTROLLING THE APPEARANCE OF THE TABLES IN CODE

The programmer has control over almost all aspects of the appearance of the table – text attributes like font, color, size and style; cell attributes like borders, size, alignment and background. All can be controlled down to

the tiniest detail. There are four arguments that can be used in most method calls to do this: STYLE_ATTR, STYLE_ELEM, INLINE_ATTR and INLINE_ELEM. (Note:the STYLE_ELEM argument usually references a user-defined style template and the INLINE_ arguments are used when there are multiple DATA arguments and are not discussed in the paper.)

The real difference between the two STYLE arguments is where the list of attributes to apply to the object is maintained. The STYLE_ATTR argument lists the "overrides" of the default attributes in the method call. The STYLE_ELEM argument references a style element that is defined in PROC TEMPLATE for the ODS STYLE that is currently in use.

A quick example will show how easy, yet powerful, this is. First, change the appearance of the header rows to set them off by overriding a few of the attributes of the cells.

	A	B	C
1	**Name**	**Height (ins)**	**Weight (lbs)**
2	Alfred	69	112.5
3	Alice	56.5	84
4	Barbara	65.3	98
5	Carol	62.8	102.5
6	Henry	63.5	102.5

```
t.row_start();
  t.format_cell(data: 'Name',
                style_attr: 'background=yellow fontweight=bold');
  t.format_cell(data: 'Height (ins)',
                style_attr: 'background=yellow fontweight=bold');
  t.format_cell(data: 'Weight (lbs)',
                style_attr: 'background=yellow fontweight=bold');
t.row_end();
```

The STYLE_ATTR argument is used to change default attributes. These are the cells in the header row, which will be made bold, with a yellow background.

Each cell can have its own list of attributes. In the code below, some attributes will also be added to the data rows to left-align the name. Also, the height and weight values are right-aligned and the weight column is in italics.

	A	B	C
1	**Name**	**Height (ins)**	**Weight (lbs)**
2	Alfred	69	*112.5*
3	Alice	56.5	*84*
4	Barbara	65.3	*98*
5	Carol	62.8	*102.5*
6	Henry	63.5	*102.5*

```
t.row_start();
  t.format_cell(data: name, style_attr: 'just=left');
  t.format_cell(data: height, style_attr: 'just=right');
  t.format_cell(data: weight,
                style_attr: 'just=right fontstyle=italic');
t.row_end();
```

As was noted for the DATA attribute earlier, the STYLE_ATTR values can be either a quoted string, as above, or a character variable (or expression). In the code above, we would get identical results as above for the header row.

```
Header_over = 'background=yellow fontweight=bold';
t.format_cell(data: 'Name', style_attr: HW_over);
t.format_cell(data: 'Height (ins)', style_attr: HW_over);
t.format_cell(data: 'Height (lbs)', style_attr: HW_over);
```

In this simple example, a hard-coded variable is used to set come common attributes. But, using variables instead of hard-coded attribute values might also allow a dataset to contain not only the data, but information about how the data should be displayed.

Below is a list of some common style attributes, though there are many more, and an example value. See the SAS documentation for a complete list.

Common style attributes (with sample value)	
background	#ff0000 (or red)
color	#00ff00 (or green)
flyover	“Pay attention to this!”
fontsize	14ot
fontstyle	Italic
fontweight	bold
Just	right
tagattr	“format:##,##0”
textdecoration	underline
vjust	bottom
…numerous border attributes (color, width, style)	

In the list above is the “just” attribute for justifying the text in the cell, which was used in an earlier example. There is also a JUST argument for the FORMAT_CELL method which does the same thing. In the example above, the following are equivalent:

```
t.format_cell(data: name, style_attr: 'just=left');

t.format_cell(data: name, just: 'left');
```

The just= attribute and the JUST argument do the same thing

Another, and probably preferable way, to deal with groups of common attributes is to create an ODS style element that contains those attributes. PROC TEMPLATE is used to create the style, which will then be used in the ODS PDF statement that defines the output file.

```
proc template;
  define style MyExcel;
  parent=styles.excel;

  style MyHeaders from body /
    background=yellow
    fontweight=bold;
  end;
run;
```

Create a new ODS style, called MyExcel, that uses the EXCEL style as its base (that’s the default style for the Excel destination).

The attributes that were in the STYLE_ATTR are now put in a STYLE element called MyHeaders. The BODY element in the EXCEL style is what usually defines the attributes of the text in the table. This overrides two of those values.

The following FORMAT_CELL calls will again produce the same headers as those above with the STYLE_ATTR argument.

```
ods excel file=<file reference> style=MyExcel;

<... previous DATA step code ...>

t.format_cell(data: ‘Name’, style: 'MyHeaders');
t.format_cell(data: ‘Height (ins)’, style: 'MyHeaders');
t.format_cell(data: ‘Weight (lbs)’, style: 'MyHeaders');
```

Use the newly defined style in the ODS EXCEL statement.

Instead of the STYLE_ATTR argument, use the STYLE_ELEM argument. The style element name is in quotes, but could also be a variable that contains the style element name.

As might expect be expected, the STYLE_ATTR and STYLE_ELEM arguments can also be used together. If both are used, the attribute list is additive, but common attributes use the values in the STYLE_ATTR. If an attribute is both in the style element and in the attributes list, the value in STYLE_ATTR takes precedence.

```
t.format_cell(data: ‘Height (ins), style: 'MyHeaders', style_attr: ‘background=#ffc7ce);
t.format_cell(data: ‘Weight (lbs), style: 'MyHeaders', style_attr: 'fontstyle=italic');
```

In the example above, attributes set in both places are used. The height header is still bold, but now has a different background color. The weight header has had another attribute added – italic text. The other attributes, set in the style, yellow background and bold, still remain.

	A	B	C
1	**Name**	**Height (ins)**	***Weight (lbs)***
2	Alfred	69	112.5
3	Alice	56.5	84
4	Barbara	65.3	98
5	Carol	62.8	102.5
6	Henry	63.5	102.5

Being able to set attributes in both places gives a lot of control over how the output will look. Also, there's no real "right" or "wrong" way to do it. Sometimes, it is very handy to be able to see all the attribute values in the DATA step code, without having to look at the PROC TEMPLATE code. It is often advantageous to see the values where they're being used. If this is so, only use the *STYLE_ELEM* argument when there are a lot of attributes being set or there are a lot of cells with a common set of attributes.

"REAL" COLUMN AND ROW HEADERS

The discussion of style attributes segued a bit into looking at column headers. There are a couple ways to create real column headers, as defined in the current ODS style.

For a row that is to be a header, setting the TYPE argument to 'H' on ROW_START call will cause the cells in that row to be styled with the attributes for headers in the current style definition. For example, in the header row we've been using, we can get rid of all the style attributes we've set, either through STYLE_ATTR or STYLE_ELEM (note: the yellow background with bold black text is not the default style). Now, with the simple code below, we get the header attributes associated with the Excel style (assuming we haven't changed the style on the ODS EXCEL statement):

```
t.row_start(TYPE: 'H');
  t.format_cell(data: 'Name');
  t.format_cell(data: 'Height (ins)');
  t.format_cell(data: 'Weight (lbs)');
t.row_end();
```

	A	B	C
1	**Name**	**Height (ins)**	**Weight (lbs)**
2	Alfred	69	112.5
3	Alice	56.5	84
4	Barbara	65.3	98
5	Carol	62.8	102.5
6	Henry	63.5	102.5

Now we have bold dark blue text with a light blue background. This would be the same appearance for procedure output headers in the Excel destination.

Another way to get the defined header appearance is to wrap the header rows between HEAD_START and HEAD_END method calls. This causes all the rows between those two methods to be treated as headers, with no reference on the rows or cells themselves needed. The code below would produce the same results as seen above.

```
t.head_start();
  t.row_start();
    t.format_cell(data: 'Name');
    t.format_cell(data: 'Height (ins)');
    t.format_cell(data: 'Weight (lbs)');
  t.row_end();
t.head_end();
```

Notice that there is no style-related information on any of the method calls here.

NOTE: Using the Excel destination there is no real difference in using either method above. For the PDF destination, beginning in v9.4M6, there is. Using the HEAD_START/HEAD_END method will treat these rows as true headers – styling them correctly and repeating them on the top of each page. The TYPE: 'H' on ROW_START used to do the same, but the behavior has changed. Now, this method only places the header row on the first page and it is not repeated on subsequent pages.

Beginning in v9.4M6, the TYPE argument can be used on a FORMAT_CELL call as well, treating it as a header column. Assume that we had the header rows as defined above, and used this for the data rows:

```
t.row_start();
  t.format_cell(data: name, TYPE: 'H');
  t.format_cell(data: height);
  t.format_cell(data: weight);
t.row_end();
```

Now use the TYPE argument on the FORMAT_CELL call to make the name column look like a header

We would now have a worksheet with both row and column headers.

	A	B	C
1	**Name**	**Height (ins)**	**Weight (lbs)**
2	**Alfred**	69	112.5
3	**Alice**	56.5	84
4	**Barbara**	65.3	98
5	**Carol**	62.8	102.5

SPANNING COLUMNS AND ROWS

There are header cells in the table above, but information in a table can often be more understandable with cells that span multiple columns or rows that contain related information. There are two arguments in the FORMAT_CELL method that control the spanning: COLSPAN, for specifying the number of columns the cell should span, and ROWSPAN, for specifying the number of rows that a cell should span.

A single row added to the table defined above can add set the height and weight columns off a little bit from the name column. This code would immediately precede the row with the height and weight header text.

```
t.row_start(type: 'H');
  t.format_cell(data: '');
  t.format_cell(data: 'Vital Stats', colspan: 2);
t.row_end();
```

A blank cell to go over the Name column and then "Vital Stats" will be in a cell that spans the Height and Weight columns.

	A	B	C
1		**Vital Stats**	
2	**Name**	**Height (ins)**	**Weight (lbs)**
3	**Alfred**	69	112.5
4	**Alice**	56.5	84
5	**Barbara**	65.3	98
6	**Carol**	62.8	102.5

There have to be the same number of columns in each row of the three table. There are three columns in the body of the table and in the row above, taking into account the vital stats column counts as two.

Rows can also be spanned with the ROWSPAN argument. It might look nicer if the "Name" header and the blank cell above it were really one cell. We can do this with ROWSPAN. This will need a tweak to both header roows.

```
t.row_start(type: 'H');
  t.format_cell(data: 'Name', style_attr: 'vjust=bottom', rowspan: 2);
  t.format_cell(data: 'Vital Stats', colspan: 2);
t.row_end();
t.row_start(type: 'H');
  t.format_cell(data: 'Height (ins)');
  t.format_cell(data: 'Weight (lbs)');
t.row_end();
```

Name is now moved to the top header row and will span two rows – the text will be at the bottom of the two cells

The second row still really have three columns – the first is in the spanned rows in the row above it

As noted earlier, all the rows should have the same number of columns. In this case we have to do a little mental math to see that there really are three columns in each row, when the row and column spanning are taken into account. The only difference in the appearance of the output here and in the example above is there is no cell border between rows 1 and 2 in column A.

	A	B	C
1		**Vital Stats**	
2	**Name**	**Height (ins)**	**Weight (lbs)**
3	**Alfred**	69	112.5
4	**Alice**	56.5	84
5	**Barbara**	65.3	98
6	**Carol**	62.8	102.5

THE TAGATTR STYLE ATTRIBUTE – LETTING EXCEL DO THE WORK

The usual SAS methods of formatting data often do not work when using the Excel destination. In the previous example, neither of the following would affect the results – the numbers would still appear as they do above, without a forced decimal place:

```
Format Height 5.1;

t.format_cell(data: put(Weight,5.1));
```

There is even a FORMAT argument on the FORMAT_CELL call that does not affect the results:

```
t.format_cell(data: Weight, format: '5.1');
```

Fortunately, there is a way to get formatted results. The TAGATTR style attribute can pass Excel formatting information that is applied to the results:

```
t.format_cell(data: Height, style_attr: 'tagattr="format:##0.0"');
t.format_cell(data: Weight, style_attr: 'tagattr="format:##0.0"');
```

The formatting references used in TAGATTR are those by Excel. If you right-click a cell in Excel and select Format Cells… and select Custom, you will see the value that would use. With the ##0.0 reference here, we get the forced decimal place and see the 0's that were not in the previous example.

	A	B	C
1		Vital Stats	
2	Name	Height (ins)	Weight (lbs)
3	Alfred	69.0	112.5
4	Alice	56.5	84.0
5	Barbara	65.3	98.0
6	Carol	62.8	102.5

TAGATTR can also pass formulas to Excel. Let's suppose we want to add a column for BMI after the Weight column. We could calculate the value in SAS or we could pass the formula to Excel and let it do the calculation. The BMI is the height (in inches) divided by the square of the weight (in pounds) times 703. So, in light of our worksheet layout, for each row (R), the weight is one column back (C[-1]) divided by the square of height, which is two columns back (C[-2]^2) all times 703. The following formula would produce the correct resuls.

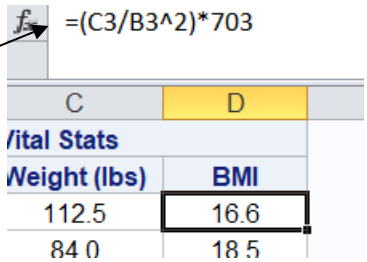
```
t.format_cell(data: ., style_attr: 'tagattr="formula:(RC[-1]/RC[-2]^2)*703 format:##0.0"');
```

Since we're passing the formula to Excel it does not matter what's in the DATA argument of the FORMAT_CELL call. It's set to missing in the example above, but could be anything – the value is ignored.

Notice that both the formula and format can be included in the TAGATTR attribute – the snippet on the left below shows the formatted results. Also notice, in the snipped on the right, that the value of the cell is the Excel formula, not just the value.

	A	B	C	D
1		Vital Stats		
2	Name	Height (ins)	Weight (lbs)	BMI
3	Alfred	69.0	112.5	16.6
4	Alice	56.5	84.0	18.5
5	Barbara	65.3	98.0	16.2
6	Carol	62.8	102.5	18.3

The formula, not just the value

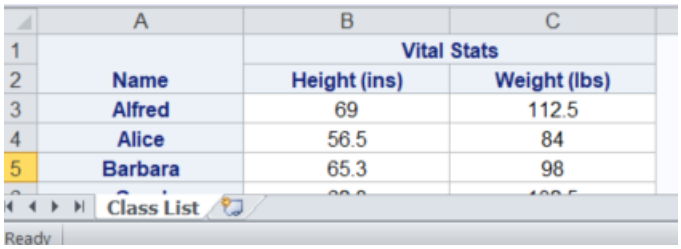


CONTROLLING APPEARANCE WITH EXCEL OPTIONS

In most ODS destinations (PDF, HTML, etc) most all of the formatting of the output is done in the SAS code, with STYLE_ATTR and other methods. However, with the EXCEL destination you can control a lot of the appearance with options that are set in the OPTIONS argument of the ODS EXCEL statement. The options contain <option.="”<value>” pairs, as many as needed, and all the values are quoted. For example, With no change to the DATA step code in the previous example, we could add some options to the ODS EXCEL statement.

```
ods excel file='<file.xlsx>'
  options(sheet_name='Class List' absolute_column_width='125px,125px,125px');
```

Now, all the columns have a fixed width and the worksheet has a more reasonable name. Again, you can use as many options as needed. Most of the options correspond to options you can set in Excel. Below is a list of a few common options (more will be discussed in the next couple sections).



Some common options for the EXCEL destination	
sheet_name	Renames the worksheet
frozen_headers	Sets the number of rows to be frozen at the top of the worksheet
absolute_column_width	List of column widths, column by column
row_heights	List of row heights, by type of row (headers, data, BY lines, titles) NOTE: header rows are defined by the HEAD_START/END methods

All the values of the options are quoted and multiple options are separated by spaces.

TITLES AND FOOTNOTES

By default, titles will go into the worksheet header and footnotes into the footer. This may or may not be the desired placements. There are options that can change that default behavior.

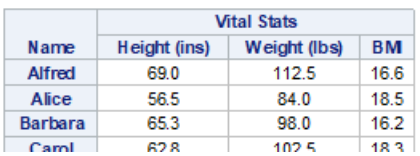
Title/footnote-related options for the EXCEL destination	
embedded_titles	"yes" puts the titles in the first row(s) of the worksheet itself, centered and spanning all the columns in the worksheet. There is one blank row after the last title.
embedded_footnotes	"yes" puts the footnotes in the last row(s) of the worksheet itself, centered and spanning all the columns in the worksheet. There is one blank row before the first footnote.

Titles and footnotes set in TITLE and FOOTNOTE statements will appear in the worksheet. There are also equivalent methods – the two examples below would generate the same titles.

```
title1 h=12pt f='Helvetica\bold' 'Class List';
title2 h=10pt f='Helvetica\bold' 'Boys and Girls';
```

TITLE method calls will usually be in the _N_ eq 1 block of code, before the start of a table

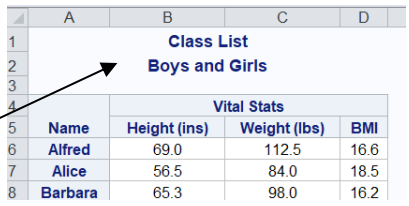
```
t.title(data: 'Class List', style_attr: 'fontsize=12pt fontface=Helvetica fontweight=bold');
t.title(data: 'Boys and Girls',
        style_attr: 'fontsize=12pt fontface=Helvetica fontweight=bold' start: 2);
```



Class List
Boys and Girls

By default, titles are in the worksheet header

The EMBEDDED_TITLES option puts the title in the worksheet itself



The START argument on the TITLE method call is equivalent to the title number. The same is true for FOOTNOTE statements and methods.

An advantage of using TITLE and FOOTNOTE methods is that the text can be dynamic, based on data or logic. Like the FORMAT_CELL DATA argument, the TITLE (and FOOTNOTE) DATA argument can be a text string, a variable, or expression.

One thing to note: the JUST argument, or style attribute, is not valid for the TITLE or FOOTNOTE methods. If you want something other than the default centered values, you must use the TITLE and FOOTNOTE statements.

MULTIPLE TABLES AND TABS

Often you want to have more than one table in your output. In Excel, this can mean one of two things: multiple tables on the same worksheet and/or multiple worksheets. In the EXCEL destination, the default behavior is to put each piece of new output, from procedures or RWI tables, on a new worksheet. This behavior can be overridden with the EXCEL destination option SHEET_INTERVAL. In this case, the value "NONE" will cause all output to remain on the same worksheet.

Option for the controlling	
sheet_interval	When should new worksheets be generated: - TABLE creates a worksheet for each table (default) - NONE creates one worksheet with all the data - BYGROUP creates a table for each BY group - Can also be PAGE, PROC or NOW

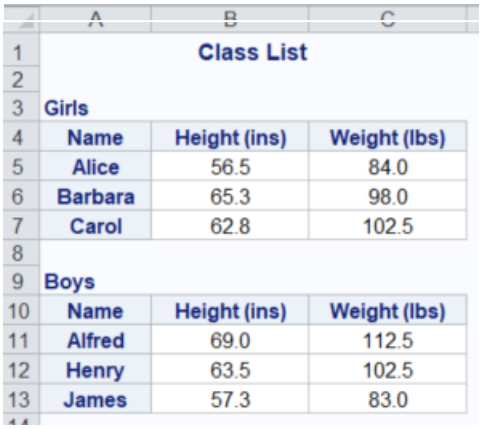
With RWI, we can produce the multiple tables in one of two ways: multiple DATA steps or logic within a DATA step to create more than one table. Appendix A shows the code to produce the class list we've used in the examples broken into separate tables for girls and boys using multiple DATA steps. There is really no change to what we've done before except using the SHEET_INTERVAL='NONE' option and adding a WHERE clause to the DATA step.

There is a lot of code repetition here as both DATA steps are basically the same. We could use a macro, passing the WHERE parameter, or we could use logic in the DATA step to do the same thing. Appendix B shows code to produce the exact same results using BY processing in a single DATA step.

The main thing to notice here is that the table start and end processing has moved. Most of the logic that was in the _N_ eq 1 block is not in a FIRST. block of code. We can have the DATA step create as many tables as needed, so the TABLE_START and header row creation are moved here.

The code in Appendix A (multiple DATA steps) and Appendix B (single DATA step) both produce the same results.

NOTE: to save space, only three obs per gender are shown below



	A	B	C
1		Class List	
2			
3	Girls		
4	Name	Height (ins)	Weight (lbs)
5	Alice	56.5	84.0
6	Barbara	65.3	98.0
7	Carol	62.8	102.5
8			
9	Boys		
10	Name	Height (ins)	Weight (lbs)
11	Alfred	69.0	112.5
12	Henry	63.5	102.5
13	James	57.3	83.0

```
if first.Sex then
  do;
    t.format_text(data: Sex, format: '$Gender.', style_attr: 'fontweight=bold');
    t.table_start();
    t.row_start(type: 'H');
      t.format_cell(data: 'Name');
      t.format_cell(data: 'Height (ins)');
      t.format_cell(data: 'Weight (lbs)');
    t.row_end();
  end;
```

Also, the TABLE_END has been moved to a LAST. block of code. Otherwise, the code is just the same as seen before.

```
if last.Sex then t.table_end();
```

Let's say we want to get a separate worksheet for the lists of girls and boys. It's a simple process, but does require separate DATA steps. In fact, the code in Appendix C is virtually identical to Appendix A (except the DATA step has been placed in a macro). There are really only three differences. First, some changes to the initial ODS EXCEL option: we do not want to suppress the default behavior of a separate worksheet for each table, so the SHEET_INTERVAL option is removed. Also, we'll have a separate worksheet for girls and boys, so the SHEET_NAME option is changed.

```
ods excel file="<file name>.xlsx" options(sheet_name='Girls' embedded_titles='yes');
```

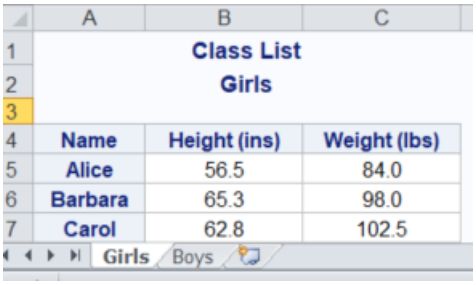
In the DATA step there is just one change. We will use a TITLE method to create a second title line with the gender value. The first title line, from a TITLE statement, will remain in effect.

```
t.title(data: Sex, format: '$Gender.', style_attr: 'fontweight=bold', start: 2);
```

Finally, between the two DATA steps we'll change the SHEET_NAME option to "Boys" (from the initial "Girls").

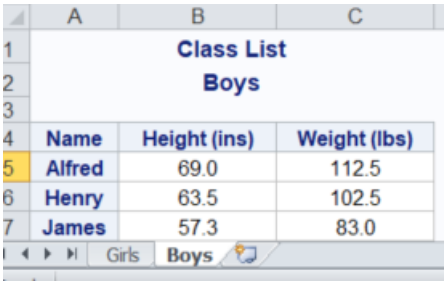
```
ods excel options(sheet_name='Boys');
```

Notice what's happening here. The next DATA step will create a table and, by default, it will go onto a new worksheet. So, before that table start, we can give the worksheet a name. There is no FILE= option on the ODSS EXCEL statement, so the output continues to go to the same spreadsheet. Also, the options are persistent, so the EMBEDDED_TITLE set in the initial ODSS EXCEL statement remains in effect. The result is shown below – two worksheets, one for the girls and one for the boys.



	A	B	C
1		Class List	
2		Girls	
3			
4	Name	Height (ins)	Weight (lbs)
5	Alice	56.5	84.0
6	Barbara	65.3	98.0
7	Carol	62.8	102.5

Girls Boys



	A	B	C
1		Class List	
2		Boys	
3			
4	Name	Height (ins)	Weight (lbs)
5	Alfred	69.0	112.5
6	Henry	63.5	102.5
7	James	57.3	83.0

Girls Boys

CREATING PRINTABLE WORKSHEETS

Viewing data in Excel is one thing, getting it to print the way you want it is another. There are a number of ODS EXCEL statement options that can help. Like the other options we've seen, these mimic options that can be set in Excel directly. .

In Appendix D is a simple DATA step creating a worksheet of baseball stats, containing a total of 11 columns. Without any options, it looks fine on the screen, but when printed has seven pages of eight columns, and then seven more pages with the remaining three columns. Probably not what's wanted and can be remedied with a few simple options.

Some printing-related options for the EXCEL destination	
pages_fitwidth	The number of pages wide the printed output will be – this is often set to 1 to get all the columns on a page
pages_fitheight	The number of pages the output can be – this is most often set to a large number, to make sure all the output fits
row_repeat	The range of rows to repeat on each printed page – note that the FROZEN_HEADERS option only affects the worksheet view, not the printed output

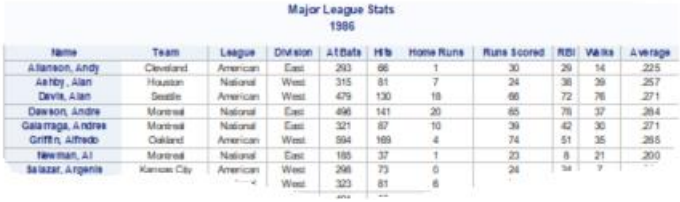
There are numerous printing-related options that can be set and a complete list can be found in the documentation listed in the References below.

In this case, the following ODS EXCEL statement solves the problem:

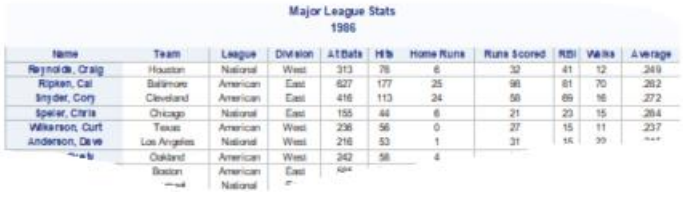
```
ods excel file="c:\test\Excel_RWI ExampleOOf.xlsx"
  options(embedded_titles='yes' pages_fitwidth='1' pages_fitheight='999' row_repeat='1:4');
```

The PAGES_FITWIDTH option is set to one, forcing all the columns onto each page. When that option is set, the PAGES_FITHEIGHT option should also be set – if not, all of the data will be on a single page. In this case, it is set to a much larger number than needed. It will only take as many pages as needed with the font sizes and row heights in effect.

There are two titles on the page, after which there is an automatic blank row, and a single header row. The ROW_REPEAT option specifies that rows 1-4 should be on the top of each page. This will be the titles and headers. Viewing the print preview from Excel, we see the following for the first couple pages.


Major League Stats
1986

Name	Team	League	Division	AtBats	Hits	Home Runs	Runs Scored	RBI	Walks	Average
Allanson, Andy	Cleveland	American	East	293	66	1	30	29	14	.225
Ashby, Alan	Houston	National	West	315	81	7	24	38	39	.257
Davis, Alan	Seattle	American	West	479	130	18	66	72	76	.271
Dawson, Andre	Montreal	National	East	496	141	20	65	78	37	.284
Galarraga, Andres	Montreal	National	East	321	87	10	39	42	30	.271
Griffin, Alfredo	Oakland	American	West	594	169	4	74	51	35	.285
Newman, Al	Montreal	National	East	185	37	1	23	8	21	.200
Salazar, Argenis	Kansas City	American	West	298	73	0	24	24	7	.245
			West	323	81	6				


Major League Stats
1986

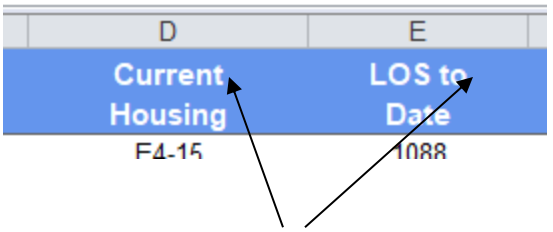
Name	Team	League	Division	AtBats	Hits	Home Runs	Runs Scored	RBI	Walks	Average
Reynolds, Craig	Houston	National	West	313	78	6	32	41	12	.249
Ripken, Cal	Baltimore	American	East	627	177	25	98	81	70	.282
Snyder, Cory	Cleveland	American	East	416	113	24	58	69	16	.272
Speier, Chris	Chicago	National	East	155	44	6	21	23	15	.284
Wilkerson, Curt	Texas	American	West	236	56	0	27	15	11	.237
Anderson, Dave	Los Angeles	National	West	216	53	1	31	15	20	.245
	Oakland	American	West	242	58	4				
	Boston	American	East	494						
		National								

All 11 columns are the page and the titles and headers are repeated, all with no change to the code itself – just adding a few simple options.

A Sometimes Important Excel Destination Option – FLOW

SAS does some behind the scenes estimation of what "nice" column widths would be. There is a potential issue as this happens before the ABSOLUTE_COLUMN_WIDTH option is applied and with the way that the that the "nice" column widths SAS calculates are implemented.

In the snippet to the right, the real values of the column headers are "Current Housing" and "LOS to Date" and the columns widht option has been set to fully display them. It looks like the text has just wrapped. However, looks can be deceiving – SAS has actually inserted a hard line break between the words to get the text to fit in what it calculated as the best width for the column. This will hardly ever be the desired behavior. This does not just happen in header rows either, data values can also have the break inserted.



Hard breaks, equivalent to Alt-Enter, are inserted in these two places

Fortunately, an option was added in v9.4M4 to address this. The FLOW= option disables the newline character insertion. The option has a number of values controlling the scope of the behavior modification.

FLOW= option value, disabling newline character insertion for…	
TABLES	All parts of the table (this is what I always use)
DATA	For data rows
HEADERS	For columns headers
ROW_HEADERS	For row headers
cell-names	Individual cells or ranges

A SPECIAL CASE – FORMATTING DATES FOR EXCEL IN RWI

One last issue to discuss is working with dates. Below are the results of sending the same date (21798 or 09/06/2019) to Excel in five different FORMAT_CELL calls.

	A	B	C	D	E
1	FORMAT assigned	FORMAT statement	FORMAT argument	TAGATTR attribute	Adjusted TAGATTR
2	21798	21798	09/06/2019	09/05/1959	09/06/2019
3					

The code to produce the above is shown in Appendix E.

Column A has a variable where the MMDDYY10 format was assigned to it in the step that created the dataset. Column B has a FORMAT statement in the RWI DATA step. Neither of those has any effect and the actual value is passed to Excel and displayed as the number.

```
format DateValueB mmddyy10.;

  d.format_cell(data: DateValueA);
  d.format_cell(data: DateValueB);
```

Column C uses the FORMAT argument on the FORMAT_CELL call and it looks like the "date" shows up in Excel. However, it is not an Excel date, but a character representation of the date, i.e. "09/06/2019" and if the cell was selected in Excel and Format Cells… was selected and a different date format picked, it would have no effect – the 09/06/2019 would remain.

```
  d.format_cell(data: DateValueC, format: 'mmddyy10.');
```

Column D uses the TAGATTR style attribute to pass along an Excel date format and it "works," at least as far as Excel is concerned. The value is an Excel date, but it has been applied to the actual value, 21798. This is an issue as SAS and Excel each have a different base for dates: in SAS, date 0 is 1/1/1960 and in Excel date 1 is 1/1/1900. So, day 21798 in Excel is 9/5/1959.

```
  d.format_cell(data: DateValueD, style_attr: 'tagattr="format:mm/dd/yyyy"');
```

Column E also uses the TAGATTR attribute, but the date has been "adjusted" by adding 21916 days to the SAS value. Column E is an actual Excel date and could be reformatted in Excel. There is also a macro in Appendix E that makes the adjustment.

```
  d.format_cell(data: DateValueE+21916, style_attr: 'tagattr="format:mm/dd/yyyy"');

  d.format_cell(data: %ExcelDateAdj(DateValueE), style_attr: 'tagattr="format:mm/dd/yyyy"');
```

So, if you're fine with having cells that look like dates, but can't be manipulated in Excel, use the FORMAT argument. If you need to have real dates, "adjust" the dates and use TAGATTR.

This issue is true of time and datetime values. There are SAS macros in Appendix E (ExcelTimeAdj and ExcelDateTimeAdj) that can be used in the same way as above.

This is only an issue with the Report Writing Interface. Output from SAS procedures (REPORT, TABULATE and PRINT) adjust the dates and send them correctly formatted, even with just a FORMAT statement.

CONCLUSION

The Report Writing Interface is a powerful tool in the SAS reporting toolbox and the ODS EXCEL destination now makes it even easier to get output in a format that so many people want. This paper has just scratched the

surface of what the Excel destination and the Report Writing Interface can do. There are many conference papers on the topics – use Lex Jansen's search site to look for papers on both topics: www.lexjansen.com.

REFERENCES

The Online Docs for the EXCEL destination can be found with a web search – something like "SAS 9.4 Output Delivery System: User's Guide" will get you a number of hits. The current ODS EXCEL statement docs are at https://documentation.sas.com/?docsetId=odsug&docsetTarget=p09n5pw9ol0897n1qe04zeur27rv.htm&docsetVersion=9.4&locale=en

The Online Docs for the Report Writing Interface are in the ODS Advanced Topics and can be found with a web search – something like "SAS Output Delivery System: Advanced Topics" will work. The current RWI docs are at https://documentation.sas.com/?docsetId=odsadvug&docsetTarget=n0w8et93ubh1enn1f34empn5948l.htm&docsetVersion=9.4&locale=en

Chevell Parker of SAS Tech Support has a great blog on the Excel Destination: https://blogs.sas.com/content/sgf/2017/02/20/tips-for-using-the-ods-excel-destination/

AUTHOR CONTACT INFORMATION

Pete Lund
Looking Glass Analytics
215 Legion Way SW
Olympia, WA 98501
(360) 528-8970
pete.lund@lgan.com

Please feel free to get in touch with questions or with tips or tricks that you've figured out!

ACKNOWLEDGEMENTS

Appendix A

Code for multiple tables on the same worksheet from multiple DATA steps

```
title 'Class List';

  * The SHEET_INTERVAL value of "none" causes multiple tables to be written to the *;
  * same worksheet.                                                                 *;

ods excel file="<file name>.xlsx"
  options(sheet_name='Class List' embedded_titles='yes' sheet_interval='none');

  * Only process the girls (F)                                                     *;

data _null_;
  set sashelp.class(where=(Sex eq 'F')) end=done;

  if _n_ eq 1 then
    do;
      declare odsout t();

  * Use FORMAT_TEXT to simulate a title                                            *;

      t.format_text(data: Sex, format: '$Gender.', style_attr: 'fontweight=bold');
      t.table_start();
      t.row_start(type: 'H');
        t.format_cell(data: 'Name');
        t.format_cell(data: 'Height (ins)');
        t.format_cell(data: 'Weight (lbs)');
      t.row_end();
    end;

    t.row_start();
      t.format_cell(data: Name, type: 'H');
      t.format_cell(data: Height, style_attr: 'tagattr="format:##0.0"');
      t.format_cell(data: Weight, style_attr: 'tagattr="format:##0.0"');
    t.row_end();

  if done then t.table_end();
run;

  * No need to repeat the title...                                                 *;

title;

  * ...and only process the boys (M)                                              *;

data _null_;
  set sashelp.class(where=(Sex eq 'M')) end=done;

  <exact code as above>

run;

ods _all_ close;
```

Appendix B

Code for multiple tables on the same worksheet from a single DATA step

```
  * Sort the sashelp.class dataset by gender and name                    *;

proc sort data=sashelp.class out=class;
  by sex name;
run;

title 'Class List';

  * The SHEET_INTERVAL value of "none" causes multiple tables to be written to the *;
  * same worksheet.                                                      *;

ods excel file="<file name>.xlsx"
  options(sheet_name='Class List' embedded_titles='yes' sheet_interval='none');

data _null_;
  set class;
  by Sex;

  * The only thing we need to do on _N_ eq 1 is declare the odsout object *;

  if _n_ eq 1 then declare odsout t();

  * The rest of the code that was in the _N_ eq 1 block is now in a first.sex *;
  * block.  Start a new table at the first observation of the BY variable and use *;
  * FORMAT_TEXT to give the table a "title"                               *;

  if first.Sex then
    do;
      t.format_text(data: Sex, format: '$Gender.', style_attr: 'fontweight=bold');
      t.table_start();
      t.row_start(type: 'H');
        t.format_cell(data: 'Name');
        t.format_cell(data: 'Height (ins)');
        t.format_cell(data: 'Weight (lbs)');
      t.row_end();
    end;

    t.row_start();
      t.format_cell(data: Name, type: 'H');
      t.format_cell(data: Height, style_attr: 'tagattr="format:##0.0"');
      t.format_cell(data: Weight, style_attr: 'tagattr="format:##0.0"');
    t.row_end();

  * Now, end the table on the last observation of the BY variable        *;

  if last.Sex then t.table_end();
run;

ods _all_ close;
```

Appendix C

Code for multiple worksheets in the same spreadsheet

```
  * Create a macro of the DATA step - the only parameter needed is the sex value   *;

%macro ClassList(gender);
  data _null_;
    set sashelp.class(where=(Sex eq "&gender") obs=3) end=done;

    if _n_ eq 1 then
      do;
        declare odsout t();

  * Create a second title using the TITLE method with a START value of 2           *;

       t.title(data: Sex,format: '$Gender.', style_attr: 'fontweight=bold', start: 2);
        t.table_start();
        t.row_start(type: 'H');
          t.format_cell(data: 'Name');
          t.format_cell(data: 'Height (ins)');
          t.format_cell(data: 'Weight (lbs)');
        t.row_end();
      end;

      t.row_start();
        t.format_cell(data: Name, type: 'H');
        t.format_cell(data: Height, style_attr: 'tagattr="format:##0.0"');
        t.format_cell(data: Weight, style_attr: 'tagattr="format:##0.0"');
      t.row_end();

    if done then t.table_end();
  run;
%mend;

title 'Class List';

  * We will have a separate worksheet for girls and boys, so name it accordingly   *;

ods excel file="<file name>.xlsx"
  options(sheet_name='Girls' embedded_titles='yes');

  * Only process the girls (F)                                                     *;

%ClassList(F);

  * Having an ODS EXCEL statement without a FILE= will continue to write to the    *;
  * same spreadsheet. Here we're naming the new worksheet.  The options are per-   *;
  * sistent, so the EMBEDDED_TITLES remains in effect.                             *;

ods excel options(sheet_name='Boys');

  * ...and only process the boys (M)                                               *;

%ClassList(M);

ods _all_ close;
```

Appendix D

Code for creating a printable worksheet

```
title1 'Major League Stats';
title2 '1986';
footnote;

  * To make the worksheet more printable, set a few new options:                  *;
  *   - PAGES_FITWIDTH='1' will put all the columns on one page                   *;
  *   - PAGES_FITHEIGHT='999' sets the max number of pages to use - if this option *;
  *     is not set and PAGES_FITWIDTH is, all the output will be on one page      *;
  *   - ROW_REPEAT='1:4' will print those rows on the top of each page.  The four  *;
  *     rows are the two title rows, the blank row between the titles and the     *;
  *     table and the header row.                                                 *;

ods excel file="<file name>.xlsx"
  options(embedded_titles='yes' pages_fitwidth='1' pages_fitheight='999' row_repeat='1:4');

data _null_;
  set sashelp.baseball end=done;

  if _n_ eq 1 then
    do;
      declare odsout b();
      b.table_start();
      b.row_start(type: 'H');
        b.format_cell(data: 'Name');
        b.format_cell(data: 'Team');
        b.format_cell(data: 'League');
        b.format_cell(data: 'Division');
        b.format_cell(data: 'At Bats');
        b.format_cell(data: 'Hits');
        b.format_cell(data: 'Home Runs');
        b.format_cell(data: 'Runs Scored');
        b.format_cell(data: 'RBI');
        b.format_cell(data: 'Walks');
        b.format_cell(data: 'Average');
      b.row_end();
    end;

  b.row_start();
    b.format_cell(data: Name, type: 'H');
    b.format_cell(data: Team);
    b.format_cell(data: League);
    b.format_cell(data: Division);
    b.format_cell(data: nAtBat);
    b.format_cell(data: nHits);
    b.format_cell(data: nHome);
    b.format_cell(data: nRuns);
    b.format_cell(data: nRBI);
    b.format_cell(data: nBB);
    b.format_cell(data: nHits/nAtBat, style_attr: 'tagattr="format:.000"');
  b.row_end();

  if done then b.table_end();
run;

ods _all_ close;
```

Appendix E

Sending dates from RWI to Excel

```
data DateTest;
  format DateValue1 mmddyy10.;
  DateValueA = mdy(9,6,2019);  DateValueB = mdy(9,6,2019);
  DateValueC = mdy(9,6,2019);  DateValueD = mdy(9,6,2019);
  DateValueE = mdy(9,6,2019);
  output;
run;

title 'Dates in RWI';

ods excel file="c:\test\Excel_RWI Example00f.xlsx";

data _null_;
  set DateTest end=done;

  format DateValueA mmddyy10.;

  if _n_ eq 1 then
    do;
      declare odsout d();
      d.table_start();
      d.row_start(type: 'H');
        d.format_cell(data: 'FORMAT assigned');
        d.format_cell(data: 'FORMAT statement');
        d.format_cell(data: 'FORMAT argument');
        d.format_cell(data: 'TAGATTR attribute');
        d.format_cell(data: 'Adjusted TAGATTR');
      d.row_end();
    end;

  format DateValueB mmddyy10.;

  d.row_start();
    d.format_cell(data: DateValueA);
    d.format_cell(data: DateValueB);
    d.format_cell(data: DateValueC, format: 'mmddyy10.');
    d.format_cell(data: DateValueD, style_attr: 'tagattr="format:mm/dd/yyyy"');
    d.format_cell(data: DateValueE+21916, style_attr: 'tagattr="format:mm/dd/yyyy"');
   * d.format_cell(data: %ExcelDateAdj(DateValueE),
                  style_attr: 'tagattr="format:mm/dd/yyyy"');
  d.row_end();

  if done then d.table_end();
run;

ods _all_ close;



%macro ExcelDateAdj(VN);
  &VN + 21916
%mend;

%macro ExcelTimeAdj(VN);
  round(&VN,60)/86400
%mend;

%macro ExcelDateTimeAdj(VN);
  datepart(&VN)+21916+(timepart(round(&VN,60))/86400)
%mend;
```