Paper A07-2009

Turn-Key Performance Metrics using Base SAS® and Excel VBA

Michael C. Frick, Warren, MI.

ABSTRACT

In today's competitive environment, everyone is being asked to do more with less. Here I describe a turn-key process which automatically produces weekly and monthly performance metrics with virtually no manual intervention. At the push of a button, base SAS is used to assemble the data and then dynamically generate custom VBA source code to produce approximately 200 Excel reports. As there is some initial set-up time, the methodology would be most useful to those who need to repetitively generate lots of tables and charts that have similar structure with a minimum of ongoing manual effort.

INTRODUCTION

A variety of tools and mechanisms exist for transferring data from SAS to Excel to create well-formatted tables and charts. For many years, I have relied on a set of customized SAS macros which pass Excel 4 macro commands to Excel via a DDE connection. These methods have been well documented in many SAS user forums. See Vyverman's SUGI 2001 paper [1] as a good starting reference. Despite the quirkiness of working with Excel 4 macros, these methods have worked well for me. Below are some of the major advantages and disadvantages I have experienced.

Advantages

1. Separation of code (queries stored in SAS) from data (reports in Excel).
2. Eliminates links and formulae from Excel reports.
3. Reduces the need to maintain Excel templates.
4. Allows for a turn-key operation which can be run by non technical personnel.

Disadvantages

1. Lack of access to new Excel capabilities.
2. Conversion issues as new versions of Excel are released
3. Difficult to produce charts (need an Excel template).
4. Limited Excel 4 documentation.
5. Requires detailed knowledge of Excel 4 and SAS macros

Unfortunately, after I migrated from Excel 2003 to 2007, not all of the Excel 4 calls worked properly. I also had issues transferring some of our Excel graph templates; hence my search for a new approach.

Here, I present what I believe is a new twist on the old theme. Instead of using Excel 4 type macros via a DDE connection, I use base SAS to generate a complete VBA program which is stored as a text file. While still in base SAS, I use a system call to launch Excel with a special XLSM file which loads the VBA program and executes it to create the desired reports. This approach maintains the advantages listed above while eliminating most of the disadvantages (still need SAS macro and VBA knowledge).

Although the primary focus of this report is to discuss how base SAS can be used to generate and execute the custom VBA code, I will also provide an overview of our turn-key batch process for completeness.

MINIMUM OF EFFORT

Figure 1 describes our batch submittal process. At the start of either a weekly or monthly reporting cycle, our first step is to use a text editor to manually update key parameters that drive the process. Usually this only involves changing the reporting dates (there is a little more work at the start of the calendar year) – total time about 1 minute. The second step is just submitting a SAS job – another minute. The final step involves making sure the job ran correctly and reports have been compressed and archived properly. Although the elapsed time is certainly longer, we spend approximately 15-30 minutes of manual effort to obtain 33 Excel workbooks that contain approximately 200 individual tabular and graphical reports.

Our batch process depends on one SAS program serving as the job control program for a series of SAS programs, each of which produces results for one primary metric. Appendix A provides more detail on our batch process for those who are interested.

Figure 1: Batch submittal process

DESIRED OUTPUT

Before attacking the SAS/VBA details, I thought it might be useful to explore the format of the desired output. Figure 2 shows the structure of our standard tabular report. The columnar structure is rigid. It will always display a column for each month, a last 3-mo column, and a year-to-date column. The row structure has more freedom. The number of detail and summary lines is dictated by the data. Row heights, font size, and font types are determined as part of the SAS generation process.

Figure 3 shows our standard bar chart. The format and colors are fixed. Although we always show 3 years of history plus current year, the number of series could vary based on the data.

A typical work-book for one of our metrics would contain 5-6 tabular reports and one bar chart. Each would have a tab for volume, one for end-end performance, and one for each sub-process performance. There would be one work-book per metric for each country plus a combined work-book for the region. Each of our SAS report programs is responsible for generating the regional and country specific work-books for a particular metric. We will look at one of these programs in detail in a later section.

Major Title 1

Major Title 2

		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	3Mo	CY
Variable	**Group 1**														
	Detail 1	40	69	76	76	80	82	-	-	-	-	-	-	78	72
	Detail 2	49	104	85	62	39	41	-	-	-	-	-	-	50	53
	Detail 3	43	100	52	44	52	66	-	-	-	-	-	-	54	52
	Group 1 Total	52	92	84	61	54	54	-	-	-	-	-	-	56	65
	Group 2														
Variable	Detail 4	63	44	42	48	43	68	-	-	-	-	-	-	47	48
	Detail 5	63	83	71	66	69	72	-	-	-	-	-	-	68	69
	Detail 6	38	24	29	35	35	92	-	-	-	-	-	-	35	32
	Detail 7	47	35	32	42	51	65	-	-	-	-	-	-	47	39
	Detail 8	52	40	34	37	52	67	-	-	-	-	-	-	43	39
	Group 2 Total	49	38	42	39	43	57	-	-	-	-	-	-	43	43
	Overall Total	49	39	56	45	46	55	-	-	-	-	-	-	48	48

Footnote 1 Footnote 2

Figure 2: Sample tabular report

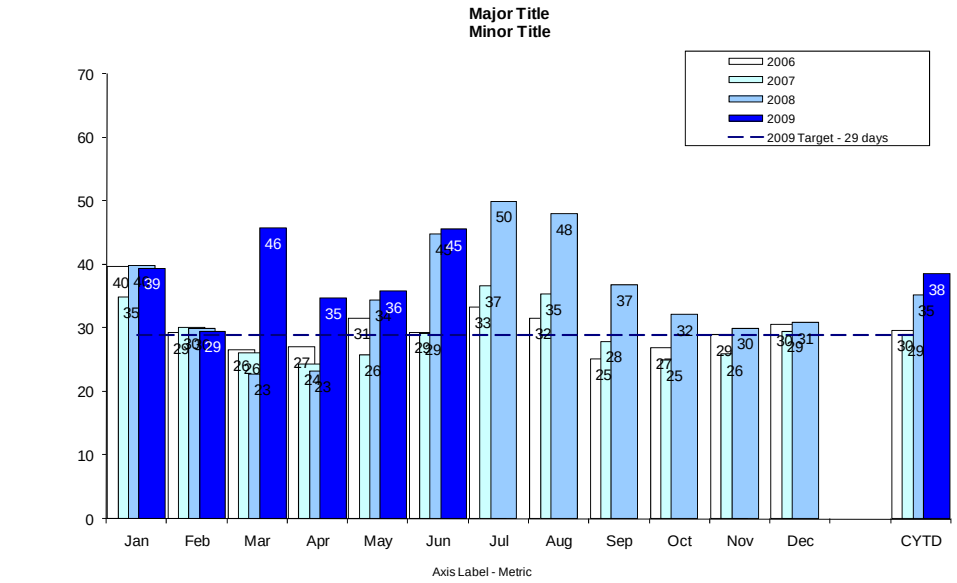

Figure 3: Sample bar chart

Static, pre-written subroutines

```
Attribute VB_Name = "OTDReports"
Function mcf_CreateWorkBook(sheet_name As String) As Workbook
 Dim wkb1 As Workbook
 Set wkb1 = Workbooks.Add
 ' delete all sheets except sheet 1 and rename as N1
 Application.DisplayAlerts = False
 Do While wkb1.Sheets.Count > 1
   wkb1.Sheets(wkb1.Sheets.Count).Delete
 Loop
 Application.DisplayAlerts = True
 Worksheets(1).name = sheet_name
 Set mcf_CreateWorkBook = wkb1
End Function
Sub mcf_SaveWorkbook(wkb As Workbook, workbook_name As String, sheet_name As String)
 Sheets(sheet_name).Select
 Application.DisplayAlerts = False
 wkb.SaveAs Filename:=workbook_name
 Application.Quit
End Sub
Sub mcf_CreateWorkSheet(wkb As Workbook, sheet_name As String)
 Call mcf_AddSheet(wkb, sheet_name)
 Call mcf_PageSetup
End Sub
```

Dynamically built subroutine calls

```
' Start of Report Specific code'
Sub Write_Reports()
 dim wkb as workbook

' Start of code to generate new workbook for metric Stock, Country NA
 set wkb=mcf_CreateWorkBook("Volume")
 Call mcf_CreateWorkSheet(wkb,"End_End")
 Call mcf_WriteHeading(9, 1, True, xlCenter, 18.75, 16, "NA Dealer Stock Replenishment Time")
 call mcf_SaveWorkbook(wkb,"c:\SAScode_ReportHistory\NA_Stock_2009_05_31","End_End")
End Sub
```

Figure 4: Sample VBA code file

ANATOMY OF OUR GENERATED VBA PROGRAMS

Since our focus is on using SAS macros to generate VBA code that will produce the charts shown in Figures 2 and 3, I thought it would also be useful to describe the structure of the VBA code before we begin to explore the SAS macros themselves. Figure 4 above provides a condensed example of a generated VBA code file.

The upper half of the code is a collection of VBA subroutines that are written ahead of time and do not change from run to run. If called properly, they will generate Excel reports in the desired format.

Our SAS macros are concerned with generating the "Write_Reports" subroutine shown in the bottom portion of Figure 4. It contains a collection of calls to the pre-written subroutines which are tailored to produce a specific set of reports based on a specific set of data. For compactness, I have only listed a few each of the VBA functions and their associated calls.

Appendix B provides a more complete list of our subroutine declarations to illustrate the types of pre-written routines that were necessary for our application. But, as the necessary routines will change from application to application, I have chosen to save space and not provide detailed VBA source code for all of our functions. Note, however, we will walk through a few of the more important routines in later sections.

ANATOMY OF OUR SAS REPORT PROGRAMS

Figure 5 shows the logic flow for one of our report programs. Step 1 is basic SAS – collect data and compute statistics. The output of this step is a SAS data set with a specific format. It must contain the following information:

- Country Code (NA, US, CN, MX)
- Calendar Year
- Group Class Variable (e.g. Vehicle Type - Car or Truck)
- Detail Class Variable (e.g. Plant Name)
- Line Type Variable (e.g. Detail, Subtotal, and Overall Total)
- Metric1-Metric14.
- Vo11-Vol14

The Metric1-Metric14 stands for 14 separate variables (1-12 for months, 13 for 3-mo value, and 14 for CYTD value). Metric is just the prefix for the metric being calculated. Hence, if we were calculating Delivery Date Reliability, we might have 14 variables named DDR1-DDR14. Similarly, the Vol1-Vol14 variables contain respective volumes. As Step 1 is basic SAS processing, I will not provide detail.

Step 2 is a collection of SAS macros responsible for generating VBA source code and is the primary focus of this report. Input to Step 2 is the SAS dataset written by Step 1 and the pre-written VBA subroutines described in the previous section. The task for step 2 is to dynamically create the proper subroutine calls to the VBA routines based on our current data and situation. Output from Step 2 is a text file that contains both the pre-written VBA subroutines and the dynamically crafted subroutine calls. This process is described in detail in the next section.

Step 3 uses a system call to launch Excel with a specially designed work-book which will load and execute the VBA file generated by Step 2. Output from Step 3 is our desired Excel reports. More detail is provided on this step in the VBA Execution section.

Step 4 is a cleanup step and uses systems calls to compress and copy the newly created reports to the appropriate archives. As this step is outside the main focus of the report, detail is not provided; however, the code segment below provides an example of how the files are compressed using a special version of the WINZIP® program which can be executed from a command line. All of the monthly workbooks are zipped to the same archive which can later be copied to various back-up locations. DSN is a macro variable built from the date parameters passed to the system and is used as a date stamp for the reports.

```
data _null_;
 x "'C:\Program Files\WinZip\Pro\WZZIP'
      &MainDrive.\&RptDir.\MonthRpt_&dsn..zip
      &MainDrive.\&RptDir.\&country._&metric._&dsn..xlsx";
run;
```

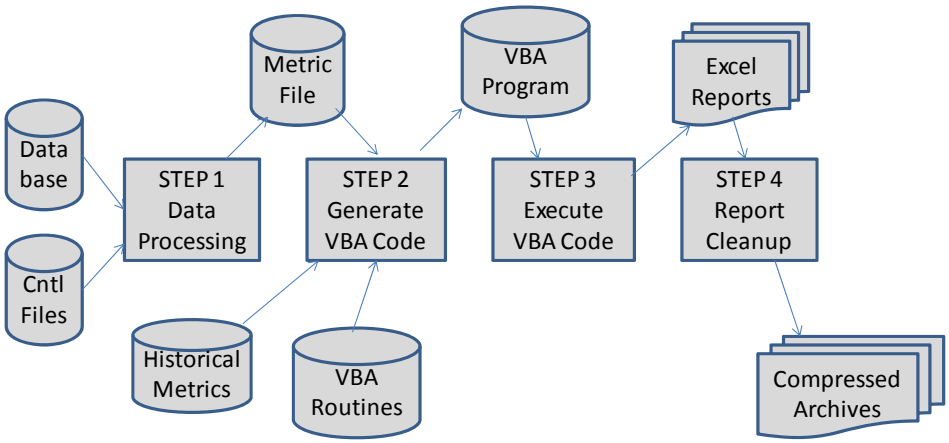

Figure 5: Logic Flow for SAS Report Program

SAS REPORT PROGRAMS – STEP 2 – VBA GENERATION

In this section, I discuss the SAS macros used to generate the VBA source code required to build our Excel-based reports. Besides the obvious flexibility and re-usability SAS macros bring to any project, they are used here to hide the VBA syntax from the SAS programmer. Once the SAS macros and VBA subroutines are written, a SAS programmer with limited to no knowledge of VBA could use them to produce new reports in the defined format. For brevity, I have omitted lines from the sample code segments below that are not explicitly germane to generating VBA code. Places where code has been omitted are marked with a ". . .".

The first block of code below exists in the main-line section of the SAS program and invokes the SAS "write_reports" macro which initiates the report generation process. It has eleven parameters. T3 is the name of the SAS dataset containing the metric data as described in the previous section. WRPT_IN and WRPT_OUT are internal file names pointed at the text file containing the pre-written VBA subroutines and the output file which will contain the completed VBA code. The "Y" indicates whether or not this particular work-book requires a bar chart. BARDATA is the name of the SAS dataset that contains the historical data for the bar chart (has the same format as T3). STOCK is the prefix for the metric variables. The "4" specifies how many work-books are to be produced followed by the country code for each.

```
%write_reports(t3,wrpt_in,wrpt_out,Y,BarData,Stock,4,NA,US,CN,MX);
```

The "write_reports" SAS macro shown below has 3 primary functions: (1) move the pre-written subroutines to the new VBA file and add the subroutine declaration for the new "Write_reports" VBA subroutine which will contain the generated subroutine calls, (2) iteratively call the "write_report" SAS macro to add the VBA subroutine calls that will generate a workbook for each country (only the first call is shown below), and (3) add the VBA code to close out the VBA routine "Write_reports".

```
%macro
write_reports(fn,f_in,f_out,bar_flag,fn_bar,metric,nCountries,c1,c2,c3,c4);
* open bas file that contains generic code and write to the new file;
data _null_;
 length txt $ 256;
 infile &f_in.;
 file &f_out.;
 input @1 txt $char256.;
 put @1 txt $256.;
data _null_;
 file &f_out. mod;
 put @1 "Sub Write_Reports()";
 .  .  .
 %if nCountries>0 %then
     %write_report(&fn.,&f_out.,&bar_flag.,&fn_bar.,&metric.,&c1.);
.  .  .
data _null_;
 file &f_out. mod;
 put @1 'End Sub';
%mend;
```

Part 1

Part 2

Part 2

The “write_report” SAS macro, shown below and called by the “write_reports” SAS macro, has the task to generate all of the VBA function calls necessary to create, format, populate, and save one new workbook.

```
%macro write_report(fn,f_out,bar_flag,fn_bar,metric,country);
data tt;
 set &fn.;
 where country_cd="&country.";
 %create_workbook(&f_out.,&country.,&metric.,Volume);
 %create_worksheet(&f_out.,tt,0,1,End_End,&metric._lt,Titl1,Titl2,Titl3,Titl4
.  .  .
%if "&bar_flag."="Y" %then
 %do;
 %create_bar_chart(&f_out.,&fn_bar.,&metric.,&c1.,Titl1,Titl2
 %end;
%save_workbook
     (&f_out.,""&MainDrive.\&RptDir.\&country._&metric._&dsn."",""End_End"");
%mend;
```

After sub-setting the data, it invokes another SAS macro called “create_workbook”, shown below, which in turn will generate the desired VBA call to the VBA function named “mcf_CreateWorkBook”. To see what is really happening, refer back to the VBA source code on page 3 (first VBA routine listed.) It has one input parameter, “sheet_name”, and is designed to create a new Excel workbook containing only one sheet with that name. Its associated VBA call is given towards the bottom of the VBA source code on page 3. It is the job of the SAS macro “create_workbook” to generate this VBA call. The SAS version of “create_workbook” has 4 parameters. The first gives it the file reference for the output text (the VBA call). The 2nd and 3rd are used to place a reference comment in the VBA code. The 4th is the worksheet name. The macro is just 3 simple put statements which create white space, a VBA comment, and the desired VBA function call.

```
%macro create_workbook(f_out,country,metric,first_sheet);
data _null_;
 file &f_out. mod;
 put @1 "    ";
 put @1 "' Start of code to generate new workbook for metric &metric.,
     Country &country.";
 put @2 "set wkb=mcf_CreateWorkBook(""&first_sheet."")";
%mend;
```

Next, the “write_report” SAS macro calls the “create_worksheet” SAS macro once for each required tabular report (only one call shown). “Create_worksheet” is described in detail below. If this particular report requires a bar chart, “write_report” calls the “create_bar_chart” SAS macro. As it is very similar to the “create_worksheet” macro except the code calls different VBA routines, I will not describe it here. The final call is to the “save_workbook” SAS macro, shown below, which creates the call to the VBA “mcf_SaveWorkbook” subroutine (VBA source on page 3.) Besides saving the workbook, this macro establishes which worksheet should be active when the workbook is opened and kills the Excel application.

```
%macro save_workbook(f_out,workbook,worksheet);
data _null_;
 file &f_out. mod;
 put @2 "call mcf_SaveWorkbook(wkb,&workbook.,&worksheet.)";
run;
%mend;
```

The job of the “create_worksheet” macro is to generate the necessary VBA subroutine calls to create a new worksheet in the workbook, name it, format it, and populate it with data. The partial code list below gives a flavor for how this is done. Note that I have eliminated much of the code that is either redundant or basic SAS programming not explicitly germane to generating VBA code (e.g. only one title call is shown.)

The data step is processing the metric SAS data set created in Step 1 and subsetted in the “create_workbook” SAS macro. The VBA subroutines that only need to be executed once per worksheet are processed under the “_n_=1” do block. Of these, the “mcf_CreateWorkSheet” does the obvious and the others provide the bulk of the titles, margin setting, columns widths, column headings, etc.

Then, for each observation in the dataset, we write a row of data into the new worksheet. As the row format is different for total lines versus detail lines, I call a different subroutine depending on the line-type. The first two parameters for all three line types are the starting Excel column # and row #. For the total lines, I also pass the row and font heights (detail line uses the defaults established in page setup). The “Str2” is a text

variable that contains the text that is written into the first column of the report. “Str” contains the monthly, 3-mo, and CYTD numbers accumulated as a comma delimited string. If you refer back to the tabular report example in Figure 2, you will notice that values for future months are shown as a “-“. As this will obviously change from month to month, I found it more convenient to bundle up the data in SAS as a comma delimited string and pass a string variable to the VBA routine. Appendix C gives this SAS code.

Unpacking and assigning the values to cells on the VBA side is straightforward. The source code for the “mcf_SubTotalLine” is given below. The first few lines format the row (set row height, etc.). The first string, captured in parameter TXTA, is assigned to the first report column. Code to parse the second string looks very similar to how one might perform the task in SAS (at least us old SAS dinosaurs).

```
%macro
create_worksheet(f_out,fn,int,addsheet,sheet,metric,titl1,titl2,titl3,titl4);
.  .  .
data _null_;
 file &f_out. mod;
 set &fn. end=endf;
.  .  .
 if _n_=1 then
    do;
     if &addsheet.=1 then put @2 "Call mcf_CreateWorkSheet(wkb,""&sheet."")";
     put @2 "Call mcf_WriteHeading(9, 1, True, xlCenter, 18.75, 16,
        ""&titl1."")";
.  .  .
      put @2 " Call mcf_ColumnHeadings(5, 2, 42.75, 16, _";
      put @2 "
""Plant,CPL,Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec,3Mo,CY"")";
      put @2 "Call mcf_SectionHeading(2, 6, """ veh_type """, 18.75, 14)";
      n=7;
    end;
.  .  .
if line_type = 'SubTotal' then
      put @2 "Call mcf_SubTotalLine(2," n ", 15.75, 12," str2  str ")";
else
if line_type ='Total' then
      put @2 "Call mcf_OverallTotalLine(2," n ", 57, 16," str2 str ")";
else
    put @2 "Call mcf_WriteDetailLine(2," n "," str2  str ")";
.  .  .
%mend;


Sub mcf_SubTotalLine(col_offset As Integer, row As Integer, _
                     row_ht As Integer, font_ht As Integer, _
                     txtA As String, txtB As String)
 Dim col As Integer
 Dim k As Integer, L As Integer
 Dim txt As String
 Rows(mcf_RowHeight_Str(row)).rowheight = row_ht
 Cells(row, col_offset).Font.Bold = True
 Cells(row, col_offset).Font.Size = font_ht
 Cells(row, col_offset).Value = Trim(txtA)
 L = 1
 col = col_offset + 2
 k = InStr(L, txtB, ",")
 Do While k > 0
    Cells(row, col).Value = Trim(Mid(txtB, L, k - L))
    L = k + 1
    col = col + 1
    k = InStr(L, txtB, ",")
 Loop
 Cells(row, col).Value = Trim(Mid(txtB, L))
 Call mcf_RightJustify(row, col_offset + 2, col_offset + 14)
End Sub
```

SAS REPORT PROGRAMS – STEP 3 – VBA EXECUTION

The output from Step 2 is a text file containing VBA source code. Our task here is to load Excel, have it import the text file as a VBA module, and then execute the VBA module.

To accomplish this task, I make use of two different Excel functions, one old and one new. In older versions of Excel, one could define a module called "Auto Open". When a workbook with an "Auto Open" module was opened, the code inside the module was executed. In newer versions of Excel, one can define modules that are executed based on different events (like mouse clicks). Luckily, one of the events is Workbook_Open. As both the new and old methods are available, I utilize the Workbook_Open method to import the text file (it executes first) and then the Auto Open functionality to execute our "Write-Reports" subroutine to create the desired reports.

The code segments below show an example of both Workbook Open and Auto Open modules that could be saved in an Excel macro workbook and later executed by the SAS code also shown below. Note, although in the example below, the VBA file always has to be saved with the same name, it would be a simple matter to have Excel read the name of the file to load from another text file whose name does not need to change. For more information on these as well as other Excel VBA capabilities see "Excel 2007 VBA Programmer's Reference" by Green et al [2].

```
Private Sub Workbook_Open()
ThisWorkbook.VBProject.VBComponents.Import Filename:="c:\mcf_Reports_.bas"
End Sub
```

```
Sub Auto_Open()
Call Write_Reports
End Sub
```

```
data _null_;
 x "'c:\program files\microsoft office\office12\excel.exe'
     /r c:\TestBAS_Import.xlsm";
run;
```

CONCLUSION

It has long been possible to dynamically create perfectly formatted Excel sheets from base SAS that require no manual cleanup using DDE connections and Excel 4 macro calls. The downside has been the inability to access modern Excel functionality added after VBA replaced the Excel 4 macro language. The VBA code generation techniques presented in this paper allow the base SAS programmer to have the best of both worlds: strict control of the Excel output from within SAS and access to all Excel functionality.

APPENDIX A – BATCH PROCESSING

Given that focus changes over time, we have added/dropped/changed performance metrics often over the years that our process has been in place. Early on I decided to isolate reports for each metric in different SAS programs and wrote one master SAS program to serve as the job control mechanism. Since the calculations for metrics are essentially independent of each other, our job controller is straightforward.

The first code segment below shows the top part of the program. Its job is to isolate the individual SAS programs from their own code storage locations and the destination for their Logs, Lists, and Reports. Key settings are stored in a text file whose name is assigned to the "SYSP" macro variable which is passed to each program through the DOS "SYSPARM" parameter (see next code segment); hence, only the master control program has the file name and path hard-coded. This feature was invaluable two years ago when we changed hardware, is a critical part of our disaster recovery plan, and enables one to easily establish a test environment separate from the production code with minimal changes to the production programs.

```
options  mprint symbolgen mlogic source2 noxwait;
%let BatchSAS=C:\Program Files\SAS\SAS 9.1\sas.exe;
%let BatchWINZIP=C:\Program Files\WinZip\Pro\WZZIP;
%let sysp=c:\SasCode\Production\ControlFiles\dd_SYSPARM.txt;
data _null_;
 x=datetime();
 put 'starting time:' x datetime20.;
proc delete data=sysp;
data sysp;
 infile "&sysp." truncover;
 input @1 drive            $2.;
 input @1 CodeDir          $80.;
 input @1 RptDir           $80.;
 input @1 PD_Drive         $80.;
 input @1 D_Archive_Drive  $80.;
 input @1 H_Archive_Drive  $80.;
 input @1 S_Archive_Drive  $80.;
 input @1 Archive_Flag  $1.;
 call symput("MainDrive",trim(drive));
 call symput("CodeDir",trim(CodeDir));
 call symput("RptDir",trim(RptDir));
 call symput("PD_Drive",trim(PD_Drive));
 call symput("D_Archive_Drive",trim(D_Archive_Drive));
 call symput("H_Archive_Drive",trim(H_Archive_Drive));
 call symput("S_Archive_Drive",trim(S_Archive_Drive));
 call symput("Archive_Flag",trim(Archive_Flag));
run;
```

The next code segment shows the second half of the master control program which consists of a SAS macro which executes our programs one at a time and the first couple of calls to the macro. Although simple, I have found this to be a robust, adaptable process that only requires base SAS to implement.

```
%macro program_run;
x """&BatchSAS."" -sysin &&MainDrive.\&CodeDir.\Reports\&Prog_Dir.\&Prog..sas
-sysparm &sysp.";
x "copy  &Prog..log   &LstDir.\&Out_Dir.\&Prog._&dsn..log";
x "erase &Prog..log";
x "copy  &Prog..lst   &LstDir.\&Out_Dir.\&Prog._&dsn..lst";
x "erase &Prog..lst";
x """&BatchWINZIP."" &LstDir.\&Out_Dir.\&Out_Dir.LogAndList.zip
     &LstDir.\&Out_Dir.\&Prog._&dsn..log";
x """&BatchWINZIP."" &LstDir.\&Out_Dir.\&Out_Dir.LogAndList.zip
    &LstDir.\&Out_Dir.\&Prog._&dsn..lst";
run;
%mend;
run;
%let Prog_Dir=LeadTime;
%let Out_Dir=LeadTime;
%let Prog=LeadTime_Sold_V3;
%program_run;
run;
%let Prog=LeadTime_Stock_V3;
%program_run;
run;
```

The final code segment shows how the individual metric programs make use the sysparm parameter passed in the background call. The “SYSPP” macro checks to see if the program was launched from batch. If so, it uses the filename passed through the sysin parameter; otherwise it defaults to a hard-coded filename so that the program can be run stand-alone if need be. Once we know the filename, it is an easy matter to read the text file and create the macro variables which will drive the reporting process. Note – having the filename hard-coded in each program is a convenience, not a necessity. In an emergency, I would need to load my entire SASCODE library to a new machine, copy the text file with the key parameters to a known location, update the text file to point to the newly created directory structure on the new machine, and run the master SAS program from batch. The hard-coded value in the individual programs is not a factor.

```
options  mprint symbolgen mlogic source2;
%macro syspp;
data _null_;
 %if &sysparm.= %then
 %do;
call symput("sysp","c:\SasCode\Production\ControlFiles\dd_SYSPARM.txt");
%end;
 %else
 %do;
     call symput("sysp","&sysparm.");
 %end;
%mend;
run;
%syspp;
run;
proc delete data=sysp;
data sysp;
 infile "&sysp." truncover;
 input @1 drive            $2.;
 input @1 CodeDir          $80.;
 input @1 RptDir           $80.;
 input @1 PD_Drive         $80.;
 input @1 D_Archive_Drive  $80.;
 input @1 H_Archive_Drive  $80.;
 input @1 S_Archive_Drive  $80.;
 input @1 Archive_Flag  $1.;
 call symput("MainDrive",trim(drive));
 call symput("CodeDir",trim(CodeDir));
 call symput("RptDir",trim(RptDir));
 call symput("PD_Drive",trim(PD_Drive));
 call symput("D_Archive_Drive",trim(D_Archive_Drive));
 call symput("H_Archive_Drive",trim(H_Archive_Drive));
 call symput("S_Archive_Drive",trim(S_Archive_Drive));
 call symput("Archive_Flag",trim(Archive_Flag));
run;
proc print data=sysp;
run;
```

APPENDIX B – VBA SUBROUTINE DECLARATIONS

```
Function mcf_CreateWorkBook(sheet_name As String) As Workbook
Sub mcf_SaveWorkbook(wkb As Workbook, workbook_name As String, sheet_name As String)
Sub mcf_CreateWorkSheet(wkb As Workbook, sheet_name As String)
Sub mcf_AddSheet(wkb As Workbook, name As String)
Sub mcf_PageSetup()
Sub mcf_SetPrintRange(row1 As Integer, col1 As Integer, row2 As Integer, col2 As Integer)
Function mcf_ColWidth_Str(col1 As Integer, col2 As Integer) As String
Sub mcf_NumberFormat(col1 As Integer, col2 As Integer, str As String)
Sub mcf_ColumnWidth(col1 As Integer, col2 As Integer, col_width As Single)
Function mcf_RowHeight_Str(row As Integer) As String
Sub mcf_ColumnHeadings(row As Integer, col_offset As Integer, row_ht As Integer,  font_ht As Integer,txtB As String)
Sub mcf_RightJustify(row As Integer, c1 As Integer, c2 As Integer)
Sub mcf_Underline(row As Integer, c1 As Integer, c2 As Integer, wt As Integer)
Sub mcf_WriteHeading(col As Integer, row As Integer, boldflag As Boolean,justification As Integer, row_ht As Integer, font_ht As Integer, txt As
String)
Sub mcf_WriteDetailLine(col_offset As Integer, row As Integer,  txtA As String, txtB As String)
Sub mcf_SubTotalLine(col_offset As Integer, row As Integer,  row_ht As Integer, font_ht As Integer, txtA As String, txtB As String)
Sub mcf_OverallTotalLine(col_offset As Integer, row As Integer,  row_ht As Integer, font_ht As Integer,  txtA As String, txtB As String)
Sub mcf_SectionHeading(col_offset As Integer, row As Integer, txtA As String,  row_ht As Integer, font_ht As Integer)
Sub mcf_WriteBarDataHdrs(col_offset As Integer, row As Integer,  txtB As String)
Sub mcf_AddTextBoxToChart(cht As Chart, sLeft As Single, sTop1 As Single,  sWidth As Single, sHeight As Single, txt As String)
Sub mcf_AddBarSeries(cht As Chart, Series_Name As String, Rvalues() As Variant, Lvalues() As Variant,  RGB_R As Integer, RGB_G As Integer,
RGB_B As Integer)
Sub mcf_AddLineSeries(cht As Chart, Series_Name As String, Rvalues() As Variant, Lvalues() As Variant,  RGB_R As Integer, RGB_G As Integer,
RGB_B As Integer)
Sub mcf_FormatAxes(cht As Chart, iMin As Integer, iMax As Integer, iMajor As Integer)
Sub mcf_AddLegend(cht As Chart, sLeft As Single, sTop1 As Single, sWidth As Single, sHeight As Single,  sFontSize As Single, sFontName As String)
Sub mcf_AddTitle(cht As Chart, sFontSize As Single, sFontName As String, txt As String)
Sub mcf_Chart_Init(cht As Chart, sLeft As Single, sTop1 As Single, sWidth As Single, sHeight As Single)
```

APPENDIX C – SAS CODE TO CREATE A COMMA DELIMITED STRING

Our standard tabular report described in Figure 2 requires a value for each of the 12 months plus a last 3 month value and a calendar year to date value. Hence, we need to pass 14 values to a VBA subroutine. The task is further complicated because the value for future months must be filled in with a dash. To simplify interface to the VBA code, I decided to write SAS code to pack the 14 values into a comma delimited string which can then be passed as one string parameter to the VBA subroutine. Below is the SAS code to create the comma delimited string. Mr. Repole, despite your best efforts, I am still somewhat of a dinosaur [3].

```
length str $ 140;
length txt $ 8;
array s(i) &metric.1-&metric.14;
str='"';
 do i =1 to 14;
    if "&int."="1" then
       do;
         if s=. then txt='-';else txt=put(s,8.0);
       end;
    else
       do;
         if s=. then txt='-';else txt=put(s,8.4);
       end;
    if str='"' then str=trim(left(str)) || trim(left(txt));
    else
        str=trim(left(str)) || ',' || trim(left(txt));
 end;
 str=trim(left(str)) || '"';
```

REFERENCES

[1] Vyverman, K. (2001), Using dynamic data exchange to export your SAS data to MS Excel - Against all ODS, Part I, Proceedings of the Twenty-Sixth SAS Users Group International Conference, paper 011-26.

[2] Green, J., Bullen, S., Bovey R., Alexander M. (2007), Excel 2007 VBA Programmer's Reference.

[3] Repole. (2007), Modernizing Your SAS Code, or How to Avoid Becoming a SAS Dinosaur, Proceedings of the Eighteenth Midwest SAS Users Group Conference, paper SAS-08.

CONTACT INFORMATION

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

Name: Mike Frick
Enterprise: Retired, October 1, 2009
Address: 30238 Underwood Drive
City, State ZIP: Warren, MI 48092
Work Phone: N/A
Fax: N/A
E-mail: mcf_for_mwsug@yahoo.com
Web: N/A