Paper 90-27

Secret Tools of the Ad Hoc Programmer

Robert Burnham, Amos Tuck School at Dartmouth College, Hanover, NH
Kathryn Sabadosa, Dartmouth Hitchcock Medical Center, Lebanon, NH
William Marble, Fidelity Investments, Marlboro, MA

ABSTRACT

Adhoc programmers are often deluged with hundreds of requests for a variety of reports in different formats: the head of sales needs an MS Excel spreadsheet with updated figures by the end of the day, and marketing needs a report of high value customers for a mailing. To survive in this world of critical response time and accuracy, adhoc programmers need to develop a library of tested and efficient code. This paper is targeted at the experienced programmer who is looking to try some proven approaches to adhoc programming.

There are two tools critical to adhoc programming: autosource macros and the Output Delivery System (ODS). Autosource macros allow SAS programmers to define code that is stored in a central library and included when required by a program, thus enabling users to easily access complex business logic in their programs. ODS, often used for web development, is a powerful tool for generating professional looking reports and tables in MS Excel and MS Word.

INTRODUCTION

It is typical for a high volume adhoc programmer to answer over 300 requests per month from a multitude of consumers: other programmers, customer representatives, managers, etc. Success as an adhoc programmer requires building a flexible and tested library of routines that can be quickly combined. It also requires the ability to deliver timely and accurate information in the form of data, reports, or both. There are two tools critical to adhoc programming: autosource macros and ODS.

MACROS: THE KEY TO SPEED & ACCURACY

Macros are the indispensable key to speed and accuracy in adhoc programming. By taking full advantage of the SAS macro facility, especially the autosource macros, adhoc programmers can encapsulate complex routines and write code that is easy to understand, implement, and maintain. Macros also allow us to develop programs that are data driven.

Despite the value of macros as a programming tool, they are often misunderstood. Wandering through many companies you will overhear experienced programmers saying, “my program worked until he messed it up with that macro stuff,” and “she went way overboard using macros – I don’t get what she was trying to prove.” Some programmers are put off by the syntax of the macro facility, in particular it’s use of special characters like % and & and it’s special rules for quoting. Adopting macros into mainstream programming and demonstrating increases in productivity is crucial to successful adhoc programming and winning over colleagues.

The first PROC encountered by most SAS programmers is PROC PRINT. A typical invocation of PROC PRINT looks something like this:

```
proc print data=test noobs uniform label;
        var name city state;
        run;
```

For many programmers, creating a macro to encapsulate this code would not make sense. The code is not complex enough to warrant hiding away the details in a macro and streamlining (from three lines to one) is minimal. Different rules apply for adhoc programmers however, and this is a perfect piece of code for a macro for the following reasons:

- This code is likely to be repeated. While you may choose to use other PROCs for your final output, PROC PRINT is commonly used while debugging when you want to look at data throughout the course of a program. Saving two lines of code in one instance will not prevent anyone from getting repetitive stress injuries, but it does add up in commonly used code.

- Most of the code stays constant regardless of where it is used. The two elements that would vary are the name of the dataset to print and the list of variables, both of which can easily be passed in as macro parameters. Placing this code in a macro eliminates any risk of mistyping an option, forgetting a semicolon, and other common mistakes that are sometimes missed.

- Reducing our code from three lines to one line does improve the readability of the program; in particular, it hides away a couple of trivial lines that are not important to the overall flow of the program. Once we have developed a macro and tested it, we can be confident that it works as we continue to write, test, and develop programs.

- If you hope to develop programs (perhaps in other languages) that generate SAS code to handle adhoc requests, using macros can simplify the complexity of quotation marks, semicolons, and parentheses between program code and generated code.

EXAMPLE 1: PROC PRINT

Since macros can encapsulate complex code, they require interfaces that behave as users expect them to. For example, a SAS programmer expects these two code segments to be equivalent:

```
proc print noobs uniform label;
  run;

proc print data=&syslast noobs uniform label;
  var _all_;
  run;
```

When we do not specify a dataset name or list of variables for VAR statement in PROC PRINT, we trust that SAS will use the appropriate default values. To be consistent, any macro that we develop to encapsulate PROC PRINT should also check for default values. For example, here is a simple version of a macro that calls PROC PRINT:

```
%macro print(dsn, varlist);
  %local data;
  %if %length(&dsn) %then %let data = &dsn;
  %else %let data = &syslast;
  proc print data=&data noobs uniform label;
```

```
    %if %length(&varlist) %then %do;
      var &varlist;
    %end;
    run;
%mend;
```

The macro %length function is used as a very simple boolean test to check that a value has been passed as an argument. If the length of the value passed to the macro is greater than zero, then the statement resolves as true and the value of the macro variable is used, else a default value is substituted. This means that we can use our macro in the following ways:

```
%print()                      /* use defaults */
%print(testdata);             /* specify data */
%print(testdata, name city); /* specify both */
```

The previous example used positional parameters and the %length function to check if values had been passed into the function. SAS offers another mechanism, keyword parameters, that allow the code to be simplified greatly at the cost of requiring the user to specify the name of the parameter in the macro invocation. For example, our macro could be coded as following:

```
%macro print(data=&syslast, var=_all_);
  proc print data=&data noobs uniform label;
    var &var;
    run;
%mend;
```

The keyword version of the macro would be invoked as:

```
%print()                /* use defaults */
%print(data=testdata); /* specify data */
%print(data=testdata,
  var=name city); /* specify both */
```

Starting from the simple example of a PROC PRINT macro, complex examples can be developed. For example, the UNIX head command displays the first ten lines of a file by default, although users are able to specify another number as an argument. We can easily extend our macro to serve as a head command for datasets by doing the following:

```
%macro head(dsn, count, varlist);
  %local n;
  %if &count gt 0 %then %let n = &count;
  %else %let n = 10;
  proc print data=&dsn(obs=&n) noobs uniform l;
    %if %length(&varlist) %then %do;
      var &varlist;
    %end;
    run;
%mend;
```

Another useful modification allows the user to print out a percentage of records in a data set, perhaps to facilitate testing.

```
%macro printrnd(dsn, pct, varlist);
  proc print data=&dsn
    (where=(ranuni(0) > &pct)) uniform l n;
    %if %length(&varlist) %then %do;
      var &varlist;
    %end;
    run;
%mend;
```

EXAMPLE 2: DATA STEP

Encapsulating the business logic stored in DATA steps can increase the productivity of adhoc programmers. For example, in a financial services corporation, an adhoc programmer’s directory may contain the following programs:

```
balance_report.sas
balance_labels.sas
balance_excel.sas
balance_ascii-tab.sas
```

If you look through the programs, you will probably find that 90% of the code is common across all four; the main difference being in the format of the final output. When a program is requested for a specific client, the programmer edits the program and runs it. If more than one type of output is needed, the programmer will run multiple programs. Moreover, if there is ever a change in the business rules for how ‘balance’ is calculated, then all four programs will need to be changed. By building a library of routines that calculates commonly requested items independent of the output, you can increase efficiency by solving these problems.

Imagine that your firm has a dataset containing the first and last name of its clients along with an identification number. A macro to request this data may do a number of useful things including concatenating two names into a single field and then return the total number of observations as a global macro variable. For example:

```
%macro personal(dsn);
  %global per_obs;
  libname corpdat “$CORPDAT”;
  data &dsn;
    set corpdat.personal nobs=count;
    if (_N_ = 1) then
      call symput(‘per_obs’, count);
    length name $40;
    label name=”Name”;
    name = trim(lname) || “, “ || trim(fname);
    run;
  proc sort data=&dsn;
    by id;
    run;
%mend;
```

Now imagine that a marketing rep has asked you for a report in the form of a multi-column document in alphabetical order by name if there are more than 25 clients, but just a simple printout if there are less. Our code can be extremely concise:

```
%let MAXOBS=25;
%personal(pers_info);

%macro adhoc;
  %if (&per_obs > &MAXOBS) %then %do;
    proc report data=pers_info headline
        missing panels=99;
      column name id;
      define name / order format=$20.;
      define id / display format=6.;
      run;
  %end;
  %else %do;
    %print(pers_info, name id);
  %end;
%mend;

%adhoc;
```

Our %personal macro has enabled us to choose at run-time which block of code to execute and concatenated the first and last names for us. This is a simple example of using autosource macros to speed up the development process, but it does illustrate the power of the approach.

PROVIDING DATA TO THE BUSINESS: HTML TO MS EXCEL & WORD

Adhoc programmers exist to serve the business and to get data to users in a format that they can work with. While requests for long printouts and ASCII files will always be significant, a majority of requests in many departments are for reports in MS Excel or MS Word format. In the past SAS programmers have exported data to these applications using Dynamic Data Exchange (DDE), StatTransfer, or various ingenious hacks for writing out tab delimited or RTF data. However, Microsoft's re-tooling of it's applications for the Internet have given programmers an extremely powerful and flexible new tool – the ability to use HTML (and HTML tables in particular) as a universal language for reporting. For example, here is a simple HTML table:

```
<html>
  <body>
    <table>
      <tr><th>Col1</th><th>Col2</th></tr>
      <tr><td>One-1</td><td>Two-1</td></tr>
      <tr><td>One-2</td><td>Two-2</td></tr>
    </table>
  </body>
</html>
```

MS Excel and MS Word can read this data and parse it into worksheets and tables respectively. This means that you can focus on one solution that generates HTML tables and be confident that you will be able to import your work using either application. There are other benefits to using this technique as well:

- The applications support most (if not all) of the formatting in HTML, so your work can be delivered with a professional look.
- HTML is plain text, not a complex proprietary binary format. SAS has always been a powerful tool for manipulating text files and it is a great for writing HTML.
- Using SAS to output HTML is a cross-platform solution and, of course, an Internet enabled solution. No matter where you actually create the HTML file (e.g. on a Unix machine or mainframe), you can download it to your desktop machine and import it into the application.

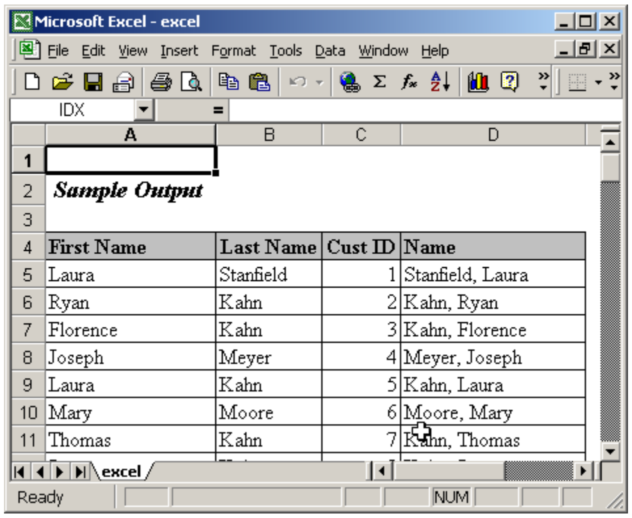
Prior to SAS7, there were limited tools available for outputting SAS data as HTML, even fewer tools created HTML tables. The introduction of the Output Delivery System (ODS) gave adhoc programmers extremely powerful tools for generating HTML, both in and out of a web environment. To demonstrate, here is a basic example of using ODS:

```
ods html body=”report.html”
%print(pi);
ods html close;
```

While ODS is extremely simple to use, productivity can still be gained by encapsulating the interface as autosource macros. In these examples, the ODS code is enhanced by providing default values for the output files including the appropriate extensions for both MS Excel and MS Word and using defaults styles are clean and professional. While both sets of files are identical HTML documents internally, using the standard extensions, .XLS and .DOC respectively, allows you to easily open the files in the correct application by double-clicking the icons.

To generate an HTML file to load into MS Excel:

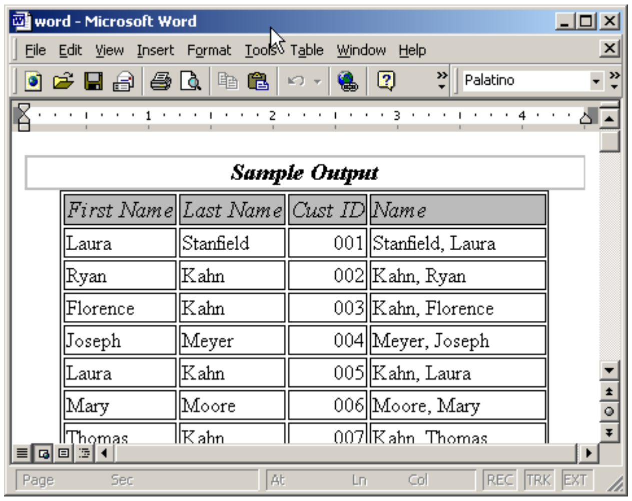
```
%macro ods_excel(filename);
  data _null_;
    length cmd fn $80;
    if (length("&filename") > 1) then do;
      if (index(lowcase("&filename"),
          ".xls") = 0) then
        fn = trim("&filename") || ".xls";
      else fn = trim("&filename");
    end;
    else fn = "output.xls";
    cmd = "ods html style=Printer body='" ||
      trim(fn) || "';";
    call execute(cmd);
    run;
%mend;
```



	A	B	C	D
1				
2	***Sample Output***			
3				
4	**First Name**	**Last Name**	**Cust ID**	**Name**
5	Laura	Stanfield	1	Stanfield, Laura
6	Ryan	Kahn	2	Kahn, Ryan
7	Florence	Kahn	3	Kahn, Florence
8	Joseph	Meyer	4	Meyer, Joseph
9	Laura	Kahn	5	Kahn, Laura
10	Mary	Moore	6	Moore, Mary
11	Thomas	Kahn	7	Kahn, Thomas

To generate an HTML file to load into MS Word:

```
%macro ods_word(filename);
  data _null_;
    length cmd fn $80;
    if (length("&filename") > 1) then do;
      if (index(lowcase("&filename"),
          ".doc") = 0) then
        fn = trim("&filename") || ".doc";
      else fn = trim("&filename");
    end;
    else fn = "output.doc";
    cmd = "ods html style=FancyPrinter body='"
      || trim(fn) || "';";
    call execute(cmd);
    run;
%mend;
```



Sample Output

First Name	Last Name	Cust ID	Name
Laura	Stanfield	001	Stanfield, Laura
Ryan	Kahn	002	Kahn, Ryan
Florence	Kahn	003	Kahn, Florence
Joseph	Meyer	004	Meyer, Joseph
Laura	Kahn	005	Kahn, Laura
Mary	Moore	006	Moore, Mary
Thomas	Kahn	007	Kahn, Thomas

To close an open ODS HTML process:

```
%macro ods_close;
  ods html close;
%mend;
```

By combining the new ODS macros with the business logic that we defined earlier, we now have a complete data-driven adhoc system in only a few lines of code. Imagine that the marketing rep returns with a new request for a spreadsheet of names if there are over 250 individuals and a report with there are fewer. Our new report system would allow us to quickly generate a program such as the following:

```
%let MAXOBS = 250;
%personal(pi);

%macro adhoc;
  %if (&per_obs > &MAXOBS) %then
    %ods_excel(excel.xls);
  %else %ods_word(word.doc);
  %print(pi, name id);
  %ods_close;
%mend;

%adhoc;
```

CONCLUSION

Being an adhoc programmer is a challenging position, deadlines are tight and requests are always marked “urgent.” However, the SAS System provides a remarkable toolkit for helping programmers meet the needs of business partners. By taking advantage of the power of the macro facility and combining it with ODS, programmers can develop a stable and efficient library of routines.

ACKNOWLEDGEMENTS

Thanks go to the Amos Tuck School of Business at Dartmouth College, Dartmouth Hitchcock Medical Center, and Fidelity Investments for their support. Thanks to the participants of SAS-L for being such a great resource to the community.

CONTACT INFORMATION

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

Robert Burnham
Amos Tuck School of Business
100 Tuck Hall
Hanover, NH 03755
Email: robert.a.burnham@dartmouth.edu
Web: http://www.dartmouth.edu/~bburnham

William Marble
Fidelity Investments
397 Williams Street MC1W
Marlboro, MA 01752
Email: bill.marble@fidelity.com

Kathryn Sabadosa
Dartmouth Hitchcock Medical Center
One Medical Center Drive, HB7505
Lebanon, NH 03756
Email: kathryn.a.sabadosa@dartmouth.edu