'FUNCTION'AL MACROS
Carol Mingst, SBC Services Inc., San Ramon, CA

ABSTRACT
While Base SAS contains many extremely useful functions, occasionally data requires more unusual manipulation. This paper will explain how to create shared MACROS that will provide most of the abilities of a base function. These can be extremely useful for converting media, such as dates and times, that are not in a SAS standard format.

INTRODUCTION
Ever become frustrated because you can't find the function you really need? Do you write the same code repeatedly because SAS just doesn't quite do what you need it to do in one step? Maybe the solution is to make a 'functional' MACRO.

I got tired of coding and re-coding important routines and decided it was time for me to actually use a MACRO other than %INCLUDE or %LET. I wanted to simplify my later coding efforts so I chose to make these MACROS extremely flexible so that they mimicked the abilities of a base SAS function.

This paper will explain how to create and use MACRO 'function's in base SAS on an MVS mainframe. All of this work has been done in batch mode.

The problem: YMMDD dates
In my line of work, there are types of records with date fields that are not influenced by change of century but, instead, change of decade. These gems have a one digit year! (YMMDD). This means that the code that reads these dates needs to be able to translate them into workable SAS dates.

Example of date: 10101 - January 01, 2001 - at least at this moment. These are in YMMDD format.

SAS understandably does not have a function that knows what to do with YMMDD dates. In the short term, a programmer can slap on the current century and decade by doing something such as this:

\[
\text{odd\_date = INPUT(PUT(odd\_date + 20000000),8.)); YMMDD8.;}
\]

But this creates problems as the decades pass, requiring extra maintenance.

What I needed was a way to translate from YMMDD to SAS date in one simple step. What I came up with was:

\[
\text{%ASA2SAS\_DATE(odd\_date);}\]

Where the value of odd\_date going in is 10101 and when it comes out is converted to a SAS date value representing January 01, 2001. This makes future programs much easier to code and follow since the above line does all of the date conversion.

Before: odd\_date = 10101 (simple number)
After: odd\_date = 14976 (SAS date for 01/01/2001)

Preparation
I didn't want to have to copy the same code into many different programs. In fact, I didn't want other programmers, who might need this code, to have to learn it or even see it, and I wanted to avoid the possibility of programmers corrupting it. This made hiding it in a MACRO a good idea. In addition to putting it into a MACRO, I put it into a source library so that the user would only need to include the source library in order to be able to use this macro whenever they need it.

\[
\%INCLUDE SASLIB(MACROS);
\]

Using this include statement before the actual use of the MACROS will do the trick. I tend to have these include statements at the beginning of my SAS code, right after the OPTIONS statement.

I keep other MACROS in the same library, grouping them into members so that one include statement will bring in all of the related 'functional' macros that might be used in the program. Another example of a tool in that library is a MACRO that will convert a SAS date back into YMMDD format.

Simple 'function' MACRO
The most basic of this kind of macro is one that can actually be used in 'if' statements and other comparisons just like a base SAS function.

An example of this would be to add ten minutes to an input time.

\[
\text{%MACRO ADD\_TEN(IN\_TIME);}\]
\[
\text{\$IN\_TIME + HMS\((0,10,0)\);}\]
\[
\text{%MEND ADD\_TEN;}\]

This example can later be used like a regular SAS function. For example:

\[
\text{\$out\_time = %ADD\_TEN(in\_time);}\]

or

\[
\text{IF TIME() GT %ADD\_TEN(in\_time);}\]

This is because the MACRO contains a statement that could actually be used to replace the MACRO call. There is no semicolon at the end of the statement inside of the MACRO (between the MACRO and MEND). Since SAS essentially replaces the code with what is in the MACRO, a semicolon would cause problems if this was used within an 'if' statement.

While this is a very simple example, the line of code could be something very complex that is done frequently in one or more programs.

Notice that the variable being passed to the MACRO is not fixed. The MACRO will work with any variable in use.
The programmer does not need to know what is going on within the MACRO, only how to use it.

Since this MACRO does not use any standard SAS variables, it does not add to the variable list for the program.

An example of a more ‘Function’al MACRO
The real power of a MACRO ‘function’ is to handle far more complex code. This kind of manipulation provides routine, necessary conversion of data for any program without the programmer needing to know how it is done.

The more complex MACROS can not be used within a comparison statement. This is because there are complete SAS statements within the MACRO (including semicolons). These functions are set up to translate the value of the field provided to the MACRO.

The call of such a MACRO is extremely easy. In the case of translating the YMDM date to a SAS date, the program needs only this one line:

%AMA2SAS_DATE(odd_date);

The value of odd_date before this line is a number representing a YMDM date. After this, it is a SAS date value. This makes things very simple and clear for programmers that want to use this function and not have to worry about how it does the work.

This is the actual source code for the MACRO that converts the YMDM date to a SAS date value. You can probably see why it is hidden from most programs and programmers.

%MACRO AMA2SAS_DATE(AMADATE);
%LET FIXED = &AMADATE;
%LET NOW = MOD(PUT(DATE(),YMDM6.),100000);
%LET DECADE = FLOOR(YEAR(DATE())/10)*10;

IF &AMADATE GT &NOW THEN DO;
  &FIXED = MDY(MOD(FLOOR(&AMADATE/100),100),
    MOD(&AMADATE,100),
    &DECADE - 10 + FLOOR(&AMADATE/10000));
END;
ELSE DO;
  &FIXED = MDY(MOD(FLOOR(&AMADATE/100),100),
    MOD(&AMADATE,100),
    &DECADE + FLOOR(&AMADATE/10000));
END;
%MEND AMA2SAS_DATE;

This is far more complex than the initial example. By putting this into a MACRO, later programmers never have to worry about all of this code. It is hidden from them, and all they need to know is how to use it.

How to make one of these
This type of code converts the value of the given field. This is done through the use of the statement:

%LET FIXED = &AMADATE;

which sets the MACRO variable %FIXED to the name of the variable passed from the main code. Later on %FIXED is set to the end value of the MACRO, effectively converting the value of the field passed to the MACRO.

Once again, I was able to avoid using any extra SAS variables within the MACRO. This means that any program that uses this can have any variable names without having more added or ones in use unexpectedly changed.

More variables
Sometimes there are reasons to include more than one variable in the mix. This is if you want to leave the original value alone and set a new variable.

%MACRO DISC_TIME(timeout,datin,timin,elps);

This MACRO will calculate the disconnect time. Timeout is a variable that will contain the disconnect time while datin is the initial date, timin is the start time, and elps is the elapsed time. This would take the following:

datin = MDY(01,01,2001);
timin = HMS(23,55,00);
elps = HMS(00,10,00);
%DISC_TIME(timeout,datin,timin,elps)

Timeout now equals 00:05:00. If this had generated a disconnect date, it would be January 2, 2001. (The output variable could be a datetime variable as well.)

The code for this conversion is:

%MACRO DISC_TIME(AMA_DTIME,AMA_DATE,AMACONN);
%LET FIXED = &AMA_DTIME;
&FIXED = TIMEPART(DHMS(&AMA_DATE,
  HHOUR(&AMA_CONN),
  MINUTE(&AMA_CONN),
  SECOND(&AMA_CONN)) +
  &AMA_ETIME);
%MEND DISC_TIME;

This MACRO includes one function to the MACRO that will contain the disconnect time while datin is the initial date, timin is the start time, and elps is the elapsed time. This would take the following:

datin = MDY(01,01,2001);
timin = HMS(23,55,00);
elps = HMS(00,10,00);
%DISC_TIME(timeout,datin,timin,elps)
Another example of multiple field manipulation
In this example, the idea is to adjust forward or backward the date and the time field. The field ADJUST is a SAS time field containing the amount of adjustment. This can be hours or days or seconds in either direction.

The reason this MACRO is so helpful is that it keeps track of the dates as well as the times. This means that if you are adding 10 hours to the connect time, the date will automatically increment if the resultant time passes midnight. Similarly, subtracting time will adjust the connect date.

```sas
%MACRO ADJUST_DATE_TIME
(AMA_DATE,AMA_CONN,ADJUST);
%LET FIXED = &AMA_DATE;
%LET FIXED = &AMA_CONN;
&FIXED = DATEPART(DHMS('&AMA_DATE,'HOUR(&AMA_CONN),
MINUTE(&AMA_CONN),
SECOND(&AMA_CONN)) + &ADJUST);
&FIXED = TIMEPART(DHMS('&AMA_DATE,'HOUR(&AMA_CONN),
MINUTE(&AMA_CONN),
SECOND(&AMA_CONN)) + &ADJUST);
%MEND ADJUST_DATE_TIME;
```

The first two fields passed to this MACRO will be adjusted. The following is an example of its use.

```sas
connect_date = MDY(12,31,2000);
connect_time = HMS(25,00,00);
%ADJUST_DATE_TIME(connect_date, connect_time, adjust_time);
```

The resultant date would be January 01, 2001 and the connect time would be 11:10:00—adding one day and one hour to the original connect date and time.

This makes the adjustment of dates and times far easier than having to code the information every time it is needed. It keeps things correct while making it easy on subsequent programmers.

Sample code
This is a very simple example of the code and how it is used.

```sas
// jcl stuff here
* //STEP1 EXEC SAS
//SASLIB DD DSN=USERID.PDS.DATA,DISP=SHR
OPTIONS LS=80;
%INCLUDE SASLIB(AMAMACRO) // NOSOURCE;
DATA EXAMPLE;
AMADATE = 10415;
SASDATE = AMADATE;
%AMA2SAS_DATE(SASDATE);
PROC PRINT;
FORMAT SASDATE MMDDYY10.;
```

The output from this job would be as follows:

<table>
<thead>
<tr>
<th>Obs</th>
<th>AMADATE</th>
<th>SASDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10415</td>
<td>04/15/2001</td>
</tr>
</tbody>
</table>

The 'function' al MACRO has changed the value of SASDATE from the simple number to a SAS date value. The formatting in the PROC PRINT proves that this has worked.

Review
The MACROS that work as functions are stored all in one data library, making them available to any programmer who needs them. In this case, all of the related MACROS are in a partitioned dataset member called AMAMACRO. This allows all of these new 'functions' to be included in a program with a single include statement:

```sas
%INCLUDE SASLIB(AMAMACRO);
```

This goes at the beginning of the SAS code so that these MACROS are available throughout the rest of the program. (SASLIB is a reference to a DD statement in the batch JCL.)

After this has been included in the code, the programmer can make use of any of the included MACROS without having to actually see the code or even understand how it works. This is similar to base SAS functions in that the using programmer only needs to know the input and output of the MACRO.

The MACROS themselves are used somewhat like SAS functions, though the actual structure depends on the MACRO in use. A simple one can be very flexible as it can be used within a comparison statement such as

```sas
IF %TEST_MACRO(field_name) EQ 1 THEN...
```

where field_name is a variable in the code.

More complex MACROS would not work in such a case as they would contain complete SAS statements within the MACRO itself. They can be used to change the value in a field.

```sas
%TEST_MACRO(field_name);
```

would be the complete statement. The value of the variable, field_name, would be different after this statement. The example of changing the dates would mean that the data would be in a numeric format coming in and a numeric date value coming out of this code.

Multiple fields can be referenced and altered by this kind of MACRO. Information can be passed into the MACRO from any of the fields. The MACRO itself can change the information in any of the fields as needed.

```sas
%TEST_MACRO(first_field, second_field) ;
```

could change the first or second field. It could change just one, using the second only as a reference to bring in more information.
CONCLUSION
The use of pre-coded MACROs for programming simplifies the work for subsequent programmers. By making these MACROs available to all, this effectively adds new functions to the base SAS language.

ACKNOWLEDGMENTS
Special thanks to Jodi Hisamoto, who has survived many code walkthroughs and has also helped edit this paper.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the author at:
Carol Mingst
SBC Services Inc
2600 Camino Ramon, 3E500Q
San Ramon, CA 94583
Work Phone: (925) 823-9728
Fax:
E-mail Address: cxmings@pacbell.com