Paper CC04

One Macro to Convert Variables with Formats to Character

Magnus Mengelbier, Limelogic Ltd, London, United Kingdom
Jan Skowronski, Limelogic Ltd, London, United Kingdom

ABSTRACT

User defined formats are common within Data Management and SAS® programming, but there are situations were the use of formats may be less appropriate and the cumbersome task of decoding all the variables with user defined formats is required. The %fmt_to_chr() is a simple utility macro that will automate the decoding of the formatted data sets and, at the same time, provide ample control of the output data sets.

The first iteration of the %fmt_to_chr() macro is discussed with a closer look at the role and reasoning behind the use of the CALL EXECUTE() statement. Further enhancements are considered and implemented through subsequent updates to provide a macro with flexible features that can select and exclude data sets, variables and formats, and define variable and label conventions.

INTRODUCTION

The decode macros %fmt_to_chr() and %llg_ds_fmt2c() are utilities that will enable an automated approach to decode formated variables to text for a single or all data sets in a library. The first generation macro %fmt_to_chr() was intended as a small simple utility that would provide a starting point for the decoded data sets. Additional tweaks and adjustments would be acceptable to provide the final output data sets.

The macro adds user configurable options around a simple decode mechanism based on the put() function. The control data set is derived and contains all the information required to perform the actual decode, which includes decode variable names, labels, lengths and other attributes.

The approach and conventions for the decode processing is based on the control data set and CALL EXECUTE() statements to generate the necessary SAS code.

The second generation macro %llg_ds_fmt2c introduces several new options that significantly decreases the amount of tweaks that is required to provide consistent output data sets. The more advanced options and error reporting provides a better overview of the format issues and the resulting output data sets.

Both macro generations make use of CALL EXECUTE() statements that are equivalent to a more standard SAS macro approach. The intent is to use the CALL EXECUTE() statements to simplify the macro code, add stability, and avoid unnecessary transfer of information between data set and macro variables.

The result is a useful second generation macro that provides a simple tool to decode a single data set or multiple data sets across SAS data libraries.

A SINGLE VARIABLE

The approach to decode any variable, both numeric and character, with a format to a text string is very straight forward using the put() function.

```
data work.out ;
  set work.in ;

  length decoded_var $ 10 ;

  *  if coded_var is numeric ;
  decoded_var = put( coded_var, numfmt.);

  *  if coded_var is character ;
```

```
  decoded_var = put( coded_var, $charfmt.);

run;
```

The approach to decode more than one variable at the same time follows the same syntax iterated for each variable. If we are to decode just a few variables within a data set, manually typing the SAS code statements is a quick exercise.

FIRST MACRO

The first macro is an attempt to automate most, if not all of the, decode steps for projects and data set standards that require data sets without user defined formats. The approach for a single variable is extended to include all variables with formats in a single data set or possibly all data sets in a library.

The decode is performed in a sequence of steps.

i. Selection of data to process
ii. Selection of variables with an associated format
iii. Finding the formats and deriving information required for the decode step
iv. Performing the actual decode
v. Generating the finished output data set

These steps are preserved as more functionality and flexibility is added to the macro. The added functionality requires a greater degree of preparation prior to the decode statements as the actual decode step is retained almost intact.

Instead of providing the entire verbose macro, we consider the steps involved through examples that directly represent the macro function.

EXAMPLE DATA

An input macro parameter specifies a single data set or library to process by the macro. If a library is selected, one or more data set names can be specified to be excluded from processing. As data set names are specified, the list acts as a filter where the actual data sets to exclude do not have to exist. Processing multiple libraries in this first macro version requires one macro call per data library.

As our example data set, consider a slightly modified version of the SASHELP.CLASS data set.

```
proc format  library = work ;
  value $gender  "M" = "Male"
                 "F" = "Female"
  ;

  value car    1 = "BMW"
               2 = "Ford"
               3 = "Volvo"
  ;
run;


data mylib.mydata ;
  set sashelp.class ;

  attrib name    label = "Name"
         sex     format = $gender.
                 label = "Sex"
         car     format = car.
                 label = "Car"
         height  format = 5.1
                 label = "Height"
         weight  format = 5.1
                 label = "Weight"
  ;
```

```
  *  generate an example variable car with the values 1, 2 and 3 ;
  car = mod( _n_, 3) + 1;
run;
```

VARIABLES TO PROCESS

All variables with an associated user defined or SAS internal format is assumed in the variable list to process. As with the library selection and data set names, a list of variable names can be specified to be excluded. Informats will be ignored by default.

```
proc contents  data = mylib.mydata  out = work.contents noprint;
run;

*  select variables with formats ;
proc sql noprint ;
   create table work.variable_list as
      select * from work.contents
        where not missing(format) or  /* <-- all character and
                                             user defined numeric formats */
              ( formatl ne 0 )        /* <-- SAS internal numeric formats */
   ;
quit;
```

DECODE INFORMATION

The macro defines the decode process through a control data set. The control data set contains all the information required to perform the actual decode, which includes decode variable names, labels, lengths and other attributes. The aim for the control data set is still to generate the below SAS statements based on user specified inputs and library data sets, regardless of how flexible the macro becomes.

```
  length decoded_var $ 10 ;

  decoded_var = put( coded_var, numfmt.);
```

The decode variable length is determined by the format label length, either using the *length* variable in the WORK.FORMATS data set from below or deriving the maximum label length in the format definition. The format reference variable *fmtreference* contains the complete format name that will be used when merging decode information and performing the actual decode step.

```
proc format  library = work   cntlout = work.formats ;
run;

proc sql noprint ;
   create table work.format_list as
      select distinct fmtname, type, length from work.formats
   ;
quit;


data work.decode_length ;
  set work.format_list ;

  *  fmtreference is the actual format name as used in your SAS code ;
  attrib  fmtreference   label = "Format"
                         length = $ 200 ;
```

```
  *  add $ to the format name if format type is character ;
  if ( upcase( type ) = "C" ) then fmtreference = "$" || strip(fmtname) || ".";
                              else fmtreference = strip(fmtname) || "." ;

  keep fmtreference fmtname type length ;
run;
```

A decode variable name *decodevar* and decode variable label *decodelabel* is derived based on the source variable name and label, respectively. The variable *varnum* is the variable position in the data set, which will be used in the last step to preserve the variable order in the final output data set with the decode variables positioned accordingly.

```
data work.variables ;
  set work.variable_list ;

  attrib  fmtreference   label = "Format"
                         length = $ 200
          decodevar      label = "Decode variable name"
                         length = $ 200
          decodelabel    label = "Decode variable label"
                         length = $ 200
  ;

  *  + 0.1  -- show decode variable to the right of source variable ;
  *  - 0.1  -- show decode variable to the left of source variable ;
  varnum = varnum + 0.1 ;

  *  define the decoded variable name and label as the original
     variable name with the suffix TXT and label (Text) ;
  decodevar = upcase( strip(name ) ) || "TXT" ;
  decodelabel = strip( label ) || " (Text)" ;

  *  generate the format reference to use in the decode step ;
  *  fmtreference is the actual format name as used in your SAS code ;
  fmtreference = strip( format ) ;

  *  add length to the format name ;
  if ( formatl ne 0 ) then
     fmtreference = strip(fmtreference) || strip( put( formatl, 8.) ) ;

  fmtreference = strip( fmtreference ) || "." ;

  *  add length and decimals to the format name ;
  if ( formatd ne 0 ) then
     fmtreference = strip(fmtreference) || strip( put( formatd, 8.) ) ;

  keep libname memname name fmtreference decodevar decodelabel varnum ;
run;
```

The only information not available in the data set WORK.VARIABLES is the decoded variable length, which is added to create the control data set WORK.VARIABLES_DECODE. Note the use of coalesce(b.length, 200), which sets a default length of 200, if our decode variable length is missing. The default decode length is a macro parameter option with default value 45.

```
*  create decode control data set ;
*  note the coalesce() function. if length is missing, we set default length to 200 ;
proc sql noprint ;
   create table work.variables_decode as
       select a.*, coalesce( b.length, 200) as length
         from work.variables a left join work.decode_length b
```

```
        on ( strip( a.fmtreference ) = strip( b.fmtreference ) )
   ;
quit;
```

PERFORMING THE DECODE

The task performing the decode is simplified as the control data set contains all the information required. The actual decode is performed using one DATA step per data set that contains variables to decode. The DATA step performing the decode is dynamically created using CALL EXECUTE() statements.

```
data _null_ ;
  set work.variables_decode  end = eof ;

  if ( _n_ = 1 ) then do;
    call execute( 'data work.temp ;' ) ;
    call execute( '  set ' ||
                     strip(libname) || '.' || strip( memname ) || ';'
                );
  end;

  *  add a commment to the SAS code ;
  call execute( '  *  -- decode variable ' || strip( name ) || ' -- ;' );

  *  define the attributes for the decode variable ;
  call execute( '  attrib ' || strip( decodevar ) );
  call execute( '              label = ' || quote( strip(decodelabel) ) );
  call execute( '              length = $ ' || put( length, 8.-L ) || ';' );

  *  decode statement, e.g decode variable = put( variable, format ) ;
  call execute( '  ' || strip( decodevar ) ||
       ' = put( ' || strip( name ) || ', ' || strip( fmtreference ) || ');' );

  call execute( '  * . ;' ); * <-- add space so code in log is easier to read ;

  if eof then
    call execute( 'run;' );

run;
```

The generated code from the CALL EXECUTE() statements is displayed in the SAS log, which greatly simplifies traceability and any required debugging.

```
1   + data work.temp ;
2   +   set MYLIB.MYDATA;
3   +   *  -- decode variable Sex -- ;
4   +   attrib SEXTXT
5   +               label = "Sex (Text)"
6   +               length = $ 6       ;
7   +   SEXTXT = put( Sex, $GENDER.);
8   +   * . ;
9   +   *  -- decode variable Weight -- ;
10  +   attrib WEIGHTTXT
11  +               label = "Weight (Text)"
12  +               length = $ 200     ;
13  +   WEIGHTTXT = put( Weight, 5.1);
14  +   * . ;
15  +   *  -- decode variable Height -- ;
16  +   attrib HEIGHTTXT
17  +               label = "Height (Text)"
18  +               length = $ 200     ;
19  +   HEIGHTTXT = put( Height, 5.1);
20  +   * . ;
```

```
21  +   *  -- decode variable car -- ;
22  +   attrib CARTXT
23  +               label = "Car (Text)"
24  +               length = $ 5       ;
25  +   CARTXT = put( car, CAR.);
26  +   * . ;
27  + run;
```

Notice the output data set WORK.TEMP is a temporary location. Since the new decode variables are appended to the right of the original set of variables, the data set structure may not be in the expected format. The macro preserves a more sensible order of the variables where the decode variables are inserted either to the left or right of the corresponding source variable. The variable order is controlled by the *varnum* variable in the control data set. A small fraction is added or subtracted to *varnum* to position the decode variable to the right or left, respectively, in relation to the source variable. The example code and the macro inserts the decode variables to the right of the source variable by default by adding 0.1 to *varnum* (see derivation of variable *varnum* in the data set WORK.VARIABLES).

```
proc sql noprint ;
   create table work.variable_sequence as
      select distinct libname, memname, name, varnum from work.contents
          union
      select distinct libname, memname, decodevar as name, varnum
            from work.variables
      order by libname, memname, varnum, name
   ;
quit;


data _null_ ;
  set work.variable_sequence   end = eof ;

  if ( _n_ = 1 ) then do;
     call execute( 'proc sql noprint ;' );
     call execute( '   create table mylib.out as ' );
     call execute( '     select ' );
  end;

  if not eof then   call execute( '            ' || strip(name) || ", ");
             else   call execute( '            ' || strip(name) );

  if eof then do;
      call execute( '       from work.temp ; ' );
      call execute( 'quit ;' );
  end;

run;
```

The generated code from the CALL EXECUTE() statements is again displayed in the SAS log, which demonstrates that the original variable order has been preserved with the new decode variables in the correct location.

```
1   + proc sql noprint ;
2   +    create table mylib.out as
3   +      select
4   +             Name,
5   +             Sex,
6   +             SEXTXT,
7   +             Age,
8   +             Height,
9   +             HEIGHTTXT,
10  +             Weight,
11  +             WEIGHTTXT,
12  +             car,
```

```
13  +             CARTXT
14  +        from work.temp ;
NOTE: Table MYLIB.OUT created, with 19 rows and 10 columns.

15  + quit ;
```

FINAL MACRO DESIGN

The principles of the final macro design should eliminate, as much as possible, the time consuming conversions, corrections, touch-ups and tweaks required to decode formatted variables as well as implement flexible features that can help automate the entire process for all data sets within one or more libraries. The second generation macro %llg_ds_fmt2c() introduces several options that were not available in the first macro design, which most often forced a user to tweak and align the resulting data sets.

The macro uses nested parameters in an effort to simplify the user interaction and logically group parameters. The nesting is entirely handled within the main macro and is beyond the scope of this paper.

```
%llg_ds_fmt2c(  data = ( library = ,
                         select = ,
                         exclude =
                       ),

                variables = ( select = ,
                              exclude =
                            ),

                formats = ( search = ,
                            select = ,
                            exlcude = ,
                            system = ,
                            systemlength = exact
                          ),

                coded = ( variable = ,
                          label = ,
                          drop = n
                        ),

                decode = ( variable = ,
                           label = ,
                           sizeblocks = 1 3 5 10 15 20 exact
                         ),
                out = ( data = ,
                        order = code decode
                      ),

                strict = y,

                validate = n,

                debug = n );
```

As a general rule, multiple choices are space delimited. For example, if you wish to decode all data sets except the data sets FORMATS and CODELIST in the three libraries STUDY01, STUDY02 and STUDY03, the macro call is configured accordingly.

```
%llg_ds_fmt2c(  data = ( library = STUDY01 STUDY02 STUDY03,
                         select = ,
                         exclude = FORMATS CODELIST
                       ),
                ...
             );
```

FINDING THE FORMATS

Both versions of the macro needs to understand what formats are available in our SAS session. We divide our formats into two disjoint sets, SAS internal and user defined formats, where the latter does not include picture formats. By default, the %llg_ds_fmt2c() macro will only process user defined formats.

The macro parameter formats = (system = …) adds the option to decode SAS internal formats. The parameter captures the list of SAS internal formats to decode. For example, specifying formats = (system = date9 ddmmyy10, …) would include all variables with the formats *DATE9* and *DDMMYY10* unless a variable is explicitly excluded by variables = (exclude = …). Note that the length of the format is explicit. The SAS internal formats are also assumed to exist with any potential errors managed accordingly, which simplifies the approach as the macro does not have to account for possible SAS version differences.

The user defined formats are defined to be those available to the current SAS session. The SAS System uses the FMTSEARCH system option to define locations to search for a user defined format. The FMTSEARCH option is a space delimited list of libraries or SAS catalogs that may contain format definitions. The list of formats and their corresponding definitions are gathered by sequentially traversing the FMTSEARCH system option.

The first step is to build the list of locations, i.e. libraries and format catalogs, to search. The data set WORK.FMT2C_FMTSEARCH_LIST contains the locations specified by the FMTSEARCH system option.

```
data work.fmt2c_fmtsearch_list ;
  set sashelp.voption ;

  *  find option FMTSEARCH in the list of system options ;
  where ( lowcase( optname ) = "fmtsearch" ) ;

  length catalog temp_dataset $ 100 ;

  *  initialise key variables   sequence   and   catalog ;
  sequence = 1;
  catalog = scan( compress( setting, "()" ), sequence, " ");

  do while ( not missing(catalog) ) ;
     *  make reference a catalog to later make sure that the format catalog exists ;
     if ( index( catalog, "." ) = 0 ) then catalog = strip(catalog) || ".formats" ;

     *  generate a temporary data set name  work.fmt2c_temp_000n ;
     temp_dataset = lowcase( "work.fmt2c_temp" || strip( put( sequence, z4. ) ) )  );

     output ;  * generates one record per library/catalog in the FMTSEARCH option ;

     sequence = sequence + 1;
     catalog = scan( compress( setting, "()" ), sequence, " ");
  end;

  keep sequence catalog ;
run;
```

The resulting data set WORK.FMT2C_FMTSEARCH_LIST contains one record per location specified.

The macro uses CALL EXECUTE() and PROC FORMATS with the format procedure's *cntlout* parameter to collect the format definitions for each specified library or SAS catalog.

```
data _null_ ;
    set work.fmt2c_fmtsearch_list ;

    *  verify library specified in FMTSEARCH exists ;
    if ( libref( scan( catalog, 1, "." ) ) ne 0 ) then return ;

    *  verify format catalog exists ;
    if ( cexist( catalog ) = 0 ) then  return ;
```

```
    *  get contents of the format catalog ;
    call execute( 'proc format  library = ' || strip( catalog ) ||
                  '             cntlout = ' || strip( temp_dataset ) || ' noprint;' );
    call execute( 'run;' );

    *  add a unique identity to the temporary data set prior to appending ;
    call execute( 'data ' || strip( temp_dataset ) || ';' );
    call execute( '  set ' || strip( temp_dataset ) || ';' );
    call execute( '  temp_dataset = "' || temp_dataset || '";' );
    call execute( 'run;' );

    *  append temporary data set to the main format list ;
    call execute( 'proc append  base = work.fmt2c_formats_full ' ||
                  '             data = ' || strip( temp_dataset ) || ';' );
    call execute( 'run;' );
run;
```

The temporary data set name and variable *temp_dataset* is crucial to know where in the FMTSEARCH sequence the format is defined. As a format may have more than one definition in the FMTSEARCH sequence, the order is important as it is the first occurrence of a format that SAS will apply and all other definitions for the same format will be ignored, regardless if that was your intent.

The result is one temporary data set for each location in FMTSEARCH, one data set WORK.FMT2C_FORMATS_FULL with all the format definitions and one data set WORK.FMT2C_FORMATS with duplicate format definitions removed.

DEFINE THE DECODE

The next step is to define the decode that has been specified in the macro call as the macro now has a list of known user defined formats and a specified list of selected SAS internal formats. As the decode is defined, the code and decode variable name and label changes are determined as well as deriving the length of the decoded variable.

The macro parameter *strict* determines how unknown formats are treated. An unknown format is a format name that does not follow the SAS internal format name conventions and user defined formats not found in the FMTSEARCH path. If the format is not known and strict = yes, then a SAS ERROR will be generated. Otherwise, a SAS NOTE is displayed in the log. In both cases, the variable will not be decoded. Only known formats will be allowed during the decode process.

The changes to the code and decode variable names and labels are defined in the macro parameters *code* and *decode*. Both the variable and label nested parameter uses the same syntax. The keywords *prefix* and *suffix* is used to denote how the variable name or label is updated. For example, the phrase coded = (variable = suffix cd , …) would add a “cd”-suffix to the coded variable name, such that AESEV would become AESEVCD.

The length of the decode variable is different for the user defined and SAS internal formats. The length when a variable has a user defined format associated is simply the maximum length of the format label.

The decode variable length when SAS internal formats are used is more difficult to derive as this may be data driven, such as the format *BEST*. The method to determine the length is defined by the formats = (systemlength = …) macro parameter. The parameter takes a space delimited list of either default lengths, e.g. formats = (systemlength = 45 55), or the keywords *exact* or *sizeblocks*. The *systemlength* list must be shorter or of the same length as specified for the formats = (system = …) parameter. If the *systemlength* list is shorter, the last value in *systemlength* is carried forward for all remaining items in the *system* parameter list.

The keyword *exact* will instruct the macro to determine the length by converting all distinct values and calculating the maximum length. The *exact* approach has two major drawbacks. The length may vary as data is updated, which can introduce difficulties to maintain a current and consistent specification. The *exact* approach can also be a time and space consuming process for large libraries with many distinct values as the SAS processing required is increased dramatically.

The macro supports a rounding option for the decoded variable length through the decode = (sizeblocks = …) option in order to derive a more presentable and consistent length. For example, if the macro call specifies decode = (sizeblocks = 5 10 15 25 exact, …), a maximum format label length of 22 will force the length of the decode variable to

be 25, e.g. rounded up to the nearest size block. The *exact* keyword specifies that the maximum length of the format label should be used, if the maximum label size is greater than 25.

CREATING THE DECODE VARIABLE

The increased flexibility and new functionality introduced in the new macro is confined to the preprocessing steps and set-up of the decode process with little impact to the decode statements. The effort to create the code and decode variables is still very simple as we retain the single data set to define the entire decode process.

The decode routine is essentially the same single DATA step calling CALL EXECUTE() as in our initial macro, which is updated to accommodate code variable name and label updates. As previously, the decoded variables along with the source variables are sent to temporary data sets that are processed in the final output step.

FINAL OUTPUT DATA SETS

The final output data sets are created in the destination specified by the out = (data = ...) parameter, which is a previously assigned library that must exist. The macro %llg_ds_fmt2c() again use the DATA step and CALL EXECUTE() statements to generate PROC SQL code, which will enable the macro to generate an output data set with the code and decode variables in the correct order. The order of the code and decode variables is defined by the out = (order = …) parameter. In addition, the data set labels and sort order of the original data sets are preserved.

CALL EXECUTE VERSUS SAS MACRO CODE

The use of CALL EXECUTE() is an attempt to simplify the macro code and decrease its complexity. The result is a decode macro that has become easier to follow and debug. The convention is to use CALL EXECUTE() statements when the macro creates SAS DATA steps and procedure calls from values that are contained in data set form.

For example, the decode step discussed previously can be written with macro code that eliminates the CALL EXECUTE() statements. Three macro coding approaches are common that address the issue but make use of different styles; the SAS function code, the macro loop to decode one variable at a time and the macro variable array.

The approach detailed below uses SAS functions available through %sysfunc() to implement a similar processing arrangement to that of the CALL EXECUTE() statements. The *noset* option for the fetch() function is intentionally used to provide ample control over what and how the information is retrieved from the data set.

```
%*  initialise macro variables ;
%local lib dataset decode_var decode_label decode_length
      source_var fmt dataset_open; ;

%let dataset_open = 0;

%*  open data set for processing ;
%let dsid = %sysfunc( open( work.variables_decode ) );

%if (&dsid = 0) %then %do;
   %*  if e.r.r.o.r opening data set, then exit ;
   %put %sysfunc(sysmsg());
   %goto exit ;
%end;

%*  process decode definitions in WORK.VARIABELS_DECODE ;
%do %while( %sysfunc( fetch( &dsid, noset ) ) = 0 );
  %let previous_lib = &lib ;
  %let previous_dataset = &dataset ;

  %let lib = %sysfunc(getvarc( &dsid, %sysfunc(varnum( &dsid, libname )) ));
  %let dataset = %sysfunc(getvarc( &dsid,
                              %sysfunc(varnum( &dsid, memname )) ));

  %let decode_var = %sysfunc(getvarc( &dsid,
```

```
                                    %sysfunc(varnum( &dsid, decodevar )) ) ;
    %let decode_label = %sysfunc(getvarc( &dsid,
                                     %sysfunc(varnum( &dsid, decodelabel )) )) ;
    %let decode_length = %sysfunc(getvarn( &dsid,
                                      %sysfunc(varnum( &dsid, length )) )) ;

    %let source_var = %sysfunc(getvarc( &dsid,
                                      %sysfunc(varnum( &dsid, name )) )) ;
    %let fmt = %sysfunc(getvarc( &dsid,
                            %sysfunc(varnum( &dsid, fmtreference )) )) ;


    %if ( ( &previous_lib ne &lib ) or
          ( &previous_dataset ne &dataset ) ) %then %do;

          %*  add run to open data step code, i.e. dataset_open = 1 ;
          %if ( &dataset_open eq 1 ) %then %do;
            run;
          %end;

          %*  start new data step ;
         data work.temp ;
             set %str(&lib).%str(&dataset) ;

          %let datastep_open = 1;
    %end;

    %*  add decode step for variable source_var ;
    %*  add attributes for decode variable ;
    attrib  &decode_var    label = "&decode_label"
                         length = $ &decode_length ;

    %*  add decode assignment ;
   &decode_var = put( &source_var, &fmt );

%end;

%*  add terminating run statement for last data set processed ;
run;


%*  point of exit if e.r.r.o.r ;
%exit:

%*  close decode defintions data set ;
%if ( &dsid > 0 ) %then  %let rc = %sysfunc( close( &dsid ) );
```

The log excerpt below depicts the level of detail attained that matches the information generated when using the CALL EXECUTE() statement. The below log is generated by the macro when the MPRINT, NOMLOGIC and NOSYMBOLGEN system options are in effect.

```
MPRINT(DECODE):   data work.temp ;
MPRINT(DECODE):   set MYLIB.MYDATA ;
MPRINT(DECODE):   attrib SEXTXT label = "Sex (Text)" length = $ 6 ;
MPRINT(DECODE):   SEXTXT = put( Sex, $GENDER. );
MPRINT(DECODE):   attrib WEIGHTTXT label = "Weight (Text)" length = $ 200 ;
MPRINT(DECODE):   WEIGHTTXT = put( Weight, 5.1 );
MPRINT(DECODE):   attrib HEIGHTTXT label = "Height (Text)" length = $ 200 ;
MPRINT(DECODE):   HEIGHTTXT = put( Height, 5.1 );
MPRINT(DECODE):   attrib CARTXT label = "Car (Text)" length = $ 5 ;
MPRINT(DECODE):   CARTXT = put( car, CAR. );
MPRINT(DECODE):   run;
```

More detailed information can be provided by also using MLOGIC and SYMBOLGEN. As both the CALL EXECUTE() statement and SAS function approaches are equally reasonable, the simplicity of both approaches may make the CALL EXECUTE() statements a valid alternative.

CONCLUSION

The two macros have become simple tools that allow for the decode step to be efficiently automated. The configurable options around the simple decode mechanism based on a control data set and the put() function does provide a basic structure, which was easily extended with new options.

The second generation macro %llg_ds_fmt2c() introduced several new options that significantly decreased the amount of tweaks that were required for consistent output data sets. The more advanced options and error reporting does provides a better overview of the format issues and the resulting output data sets.

The CALL EXECUTE() statement, which is used by both macros, simplifies the macro code, adds stability, and decreased the amount of transfer of information between a data set and macro variables.

CONTACT INFORMATION

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

Magnus Mengelbier or Jan Skowronski
Limelogic Ltd
London, United Kingdom

E-mail: papers@limelogic.com
Web: www.limelogic.com