Abstract

This paper is an introduction to using the C language and SAS/TOOLKIT software to write your own SAS® procedures. The paper uses a popular 1991 SASware Ballot item, replacement of missing values (item 12) to illustrate some important SAS/TOOLKIT concepts and functions. The paper assumes that you have some experience with C language and with using SAS procedures. Previous experience with writing SAS procedures is not needed.

The example procedure reads a SAS data set and replaces missing values for all numeric variables or user-selected numeric variables. The syntax is as follows:

```c
proc nomsis data=inputdataset
   out=outputdataset
   newvalue=number;
   var variables;
   exclude variables;
```

The defaults will be discussed in detail below, but they are what most users would expect.

The procedure is portable to any hardware supported by SAS/TOOLKIT software, with no source code changes required. Complete listings of the procedure and the grammar are at the end of the paper.

Initialization

First, the example defines the SASPROC and MAINPROC symbols. These tell the SAS/TOOLKIT software that this source file is a SAS procedure (rather than a format or function, for example) and that it contains the main procedure (the entry point from the SAS System). The U_MAIN macro is used to define the entry point from the SAS System. The first thing the procedure does is to call UWPRCC to initialize the SAS/TOOLKIT environment. The UWPRCC call is always required as the first statement in a user-written C procedure.

Obtaining the Input Parameters

After initialization, the example calls SAS_XSPARSE to parse the input statements, passing the address of the grammar function. The grammar module will be discussed in detail later. Once the input statements are parsed, the procedure checks the error field in the procedure structure. If there were any syntax errors, it quits at this point. Note how easy the parsing is -- the procedure doesn't have to read in and understand the statements themselves.

The next thing the example does is obtain pointers to the variable lists, using the SFLP macro. This procedure uses three variable lists: the one from the VAR statement, the one from the EXCLUDE statement, and a list that will be set up to contain all other variables in the data set so that they can be copied. The grammar puts these list pointers at positions 4, 5 and 6 in the statement structures in the proc header.

The procedure accepts either a VAR statement (list of variables to process) or an EXCLUDE statement (list of variables not to process). It doesn't allow the user to provide both VAR and EXCLUDE, because it would be hard to ascribe a meaning to this combination and handling it would complicate processing. So the next thing the procedure does is check whether the user supplied both VAR and EXCLUDE. If the user supplied the statement, the associated list pointer (list4 and list5) is non-NULL. If the user supplied both VAR and EXCLUDE, the procedure writes an error message to the SAS log with SAS_XPSLOG and quits.

Next, the procedure obtains the value that is to replace missing values, using the SFF macro. If the user didn't supply a NEWVALUE= parameter, then a default value of zero (the second operand of the macro) is used. The grammar allows the user to supply either a number or a missing value for NEWVALUE=, so the procedure can be used to replace one type of missing value with another.
The file pointers for the input and output SAS data sets are obtained next, using the SFFILE macro. (The grammar placed these pointers at statement positions one and two in the proc header). The input data set will always be present, because the grammar specifies a default input data set (_LAST_) if there was no DATA= parameter.

The Output Data Set

The output data set is optionally supplied by the user. If not supplied, its file pointer (outfile) will be NULL. In this case, the procedure rewrites the input data set (as PROC SORT does). The next block of statements handles this case. SAS_XOINFO is used to copy the libname, member name, data set type, and label from the input data set to an XOOPNSTR structure, outstr. The open mode is set to sequential output, and SAS_XOPEN is called to open the data set for output. If any problems occur, SAS_XPRLOG is used to write to the SAS log a message that corresponds to the return code from SAS_XOPEN, and the procedure quits.

Setting Up SAS Variable Lists

Setting up the lists of variables comes next. The procedure uses two lists. The first contains all the numeric variables that are to be processed (i.e., have missing values replaced). The second list contains all the other variables in the input data set, such as any character variables and any numeric variables that are not to be processed. This list is used to copy the variables across into the output data set.

SAS_XVDFLST (stands for "default list") is used to set up the variable lists. First, it's called to set up list 4, the list corresponding to the VAR statement. If the VAR statement was present, SAS_XVDFLST does nothing, but if there was no VAR statement, it sets up a list of all the numeric variables in the data set, excluding any that were in list 5. (Recall that list 5 came from the EXCLUDE statement). Thus, if there's no VAR statement, the procedure uses a reasonable default, and if the user wants to exclude particular variables from processing, he or she can use the EXCLUDE statement. For simplicity, the rest of the paper will refer to this list as the "VAR list" whether it was obtained from a VAR statement or from SAS_XVDFLST; subsequent processing is the same for both cases.

After the VAR list is set up, the procedure uses SAS_XVDFLST again to set up a list (list 6) containing all the other variables. This call is slightly different; first it uses 3 as the second parameter, to ask for both numeric and character variables, and second the grammar provides no way for the user to provide list 6. Thus, SAS_XVDFLST always sets up a list. This time list 4 is excluded; a variable should be in either the VAR list or list 6, but not both.

When both lists are set up, the procedure uses the SFN macro to obtain the number of entries in each. Note that one of the two lists might have no entries; for example if the input data set has only numeric variables and all of them are to be processed. In general SAS procedures need to allow for the possibility of empty lists; if a particular list has to have entries, the procedure should test the list pointer and quit if it's NULL.

Scatter Input and Gather Output Setup

Scatter input and gather output are important concepts for SAS procedure writing (both version 6 and earlier versions). Scatter input reads selected SAS variables from a data set into memory. Often (although not in this example), variables will be read into non-contiguous areas of memory, hence the term scatter input. The variables are processed in memory; for this procedure, processing consists of replacing missing values. Then, the values are written from memory to output data sets; this is gather output (it can be thought of as gathering together the values from memory into one place in the output data set). (Procedures that don't create new SAS data sets use scatter input but not gather output).

This procedure uses scatter input and gather output for the variables in the VAR list. At the same time, necessary setup is done to copy the other variables to the output data set.

First an observation buffer is allocated with SAS_XMEMGET. One slot is needed for each variable in the VAR list, and all the variables are
numeric and thus of C data type double. So to get the space required, the procedure simply multiplies the number of variables in list 4 by the size of a double. It's better to use sizeof(double) rather than 8, since then the procedure will port to any future hardware platforms where doubles are a different size. The procedure checks the buffer value returned from SAS_XMEMGET to make sure that memory was available. While such a small allocation is unlikely to fail, it's always possible, and it's good programming practice always to check that necessary resources are available. If the memory wasn't available, the procedure can't do its job, so it quits.

This example uses SAS_XMEMGET, but be aware that there are other memory allocation routines available. If your procedure needs to allocate many small chunks of memory, then the routines that allocate from a pool (SAS_XMPOOLC, SAS_XMALLOC, etc.) are a better choice, because they have less overhead per chunk.

Use of C library memory allocation routines (malloc, etc) is not recommended. The principal reason is that this may cause memory fragmentation, because the C library is outside the control of the SAS Supervisor's memory management. The fragmentation might cause memory shortages either for your procedure or for subsequent DATA or PROC steps, particularly if a procedure needs a large amount of contiguous memory.

After the observation buffer is obtained, SAS_XVGETI and SAS_XVPUTI are used to initialize for gather input and scatter output. SAS_XVGETI is called once for each input data set (this procedure has only one, of course) to initialize gather input, and SAS_XVPUTI is called once for each output data set. Both SAS_XVGETI and SAS_XVPUTI return file pointers (via their third parameter) for use with other routines.

The procedure then loops through the VAR list. For each variable, it calls SAS_XVGETD to allocate space for the variable in the observation buffer. The "offset" form of SAS_XVGETD is used; the first variable is at offset zero, the second at offset (sizeof(double)), the third at offset(2*sizeof(double)), etc. Then, SAS_XVNAME is used to obtain a pointer to the NAMSTR (name structure) for the variable. The pointer is stored in namptr, and passed to SAS_XVPUTC, which defines the variable in the output data set and provides its offset in the observation buffer (using the "offset" form of SAS_XVPUTC).

Although no processing is done on the remaining variables, they do need to be written to the output data set. The SAS_XVPUTCV ("copy variable") call accomplishes this. SAS_XVPUTCV is passed the pointer to the list of variables (list6) and sets up to copy all the variables in the list from the input data set (infile) to the output data set (xvputptr).

The setup process is completed by calling SAS_XVGETE (end definition of scatter input) and SAS_XVPUTE (end definition of gather output).

Reading, Processing and Writing Data

Now the procedure is set up to read, process, and write observations. SAS_XBYNEXT sets up to read from the start of the input data set. For procedures that support a BY statement, SAS_XBYNEXT starts a new BY group each time it's called. This procedure doesn't support a BY statement, but SAS_XBYNEXT can still be used. If there are no BY groups, it simply treats the entire data set as one large BY group.

SAS_XBYGET moves to the next (or first) observation. Again, this routine can be used whether or not there are actually any BY groups. If there aren't, it just returns NULL when the end of the input data set is reached.

SAS_XVGET reads the observation into the observation buffer. Since the "offset" form of SAS_XVGETI was used, SAS_XVGET is passed the pointer to the buffer so that it knows where to put the observation.

Once the observation is in the buffer, each variable in the VAR list is processed in turn. This is very straightforward since the variables are
contiguous in the buffer. Each variable's value is tested with SAS\_ZMISS. If the value is missing, the replacement value is substituted. SAS\_ZMISS is a portable test for missing values and should always be used in place of comparing for specific values. SAS\_ZMISS handles both standard and special missing values.

When all variables have been processed, SAS\_XVPUT and SAS\_XOADD are used to write the observation to the output data set.

**Termination**

Once SAS\_XBYNEXT returns NULL, all observations in the input data set have been read and the procedure has no more work to do. It returns to the SAS System with SAS\_XEXIT. Note that it is not necessary to free any resources such as memory; the SAS Supervisor takes care of this automatically when SAS\_XEXIT is called. SAS\_XEXIT should always be used to end a procedure, whether normally or with errors, since it allows the SAS Supervisor to clean up. Do not use C statements such as exit or return to end the main procedure module; the results are unpredictable.

**The Grammar Module**

The SAS procedure grammar language is very powerful. It removes almost all of the burden of reading, parsing, and validating procedure input statements from the procedure, allowing the procedure writer to concentrate on the task for which the procedure is being written. Because of its great power, the grammar language may need some learning time, particularly if you haven't experienced other grammar specification languages such as yacc.

The example grammar will be explained statement by statement. Understanding this relatively simple grammar will help you use the full power of the grammar language for more complex input that your own user-written procedures might require.

The grammar consists of a series of definitions that break down the input statements into successively finer pieces. First, it %INCLUDEs STUBGRM. STUBGRM contains a number of productions that all procedure grammars need, so you must always %INCLUDE it as the first statement in your grammar. Don't be put off by the FLA-like syntax of %INCLUDE. The grammar language is the same regardless of the language you choose to write your procedure in.

The next statement says that the program consists of statements (ANYSTMT) terminated by an ENDDJ (a RUN statement or another DATA or PROC step). In general you can use any names you like on the left-hand side of your grammar statements, as long as you don't duplicate the predefined productions and lexicals described in the SAS/TOOLKIT manual. ANYSTMT is such an arbitrary name (PROGRAM is not, however). You can use names that are meaningful to you.

ANYSTMT is then defined to be a NOMICISSSTMT ("NOMISS statement"), a VARSTMT ("VAR statement") or EXCLUDESTMT ("EXCLUDE statement"). Again, these names are arbitrary. NOMISSSTMT is then broken down into the word "NOMISS" followed by NOMICISSOPTS ("NOMISS options"). NOMISS is the procedure name; it is placed in quotation marks to indicate that it is a keyword rather than another production to be further broken down. The asterisk after NOMICISSOPTS specifies that it can be repeated one or more times. This statement also introduces some semantic actions, identifiable because they start with an at sign. When the SAS parser comes to a semantic action, it takes appropriate action. (This occurs when the procedure is run, not when the grammar is compiled). @PROCINIT does initialization for the procedure. @STMTINIT(6) sets up 6 slots in the procedure's statement structure; these will be used later in the grammar. @DSDFLTI provides a default input data set and stores information about it in slot 2 of the statement structure. (You'll probably need to change some of the numbers if you use this -- see the detailed descriptions in the example). @STMTEND completes the setup of the procedure's statement structure.

NOMICISSOPTS is then broken down further into OUT=, DATA=, and NEWVALUE= parameters. These are keywords, not productions to be further broken down, so again they're in
quotation marks. OUT= and DATA= specify SAS data sets, so each has a DSFIELD production associated with it (DSFIELD comes from STUBGRM). Also, each has a @DS semantic action associated with it to store the data set information in the allocated slot in the procedure statement structure, 1 for OUT and 2 for DATA.

NEWVALUE= is treated as a numeric parameter. The value after "=" can be either a number or a missing value (written just as it would be in a DATA step). The @PARM semantic action stores the value in slot 3 of the statement structure.

VARSTM breaks down quite simply into "VAR" followed by a list of SAS variables, specified by the VARLIST production from STUBGRM. @STMTPROC tells the parser to anchor the list in the proc statement structure. This is needed because it is also possible to anchor the list elsewhere, which you would need to do if your procedure accepts multiple statements of the same type (for example, PLOT statements). @STMMLST anchors the list in slot 4 of the statement structure.

EXCLUDESTM is very similar to VARSTM, except that it's anchored in slot 5 of the statement structure.

SAS and SASware ballot are registered trademarks and SAS/TOOLKIT is a trademark of SAS Institute, Inc., Cary, NC, USA.
Appendix

/*-----------------------------------------------*/
/* NAME: NOMISS */
/* TYPE: PROCEDURE */
/* PURPOSE: Example procedure. Replaces missing values in */
/* a SAS dataset with a user-specified value. */
/*-----------------------------------------------*/

/* The syntax of the procedure is as follows: */
/* PROC NOMISS options; */
/* VAR variables; */
/* EXCLUDE variables; */
/* where the options can be */
/* */
/* OUT-data set specifies an output data set. The default is */
/* to rewrite the input data set. */
/* */
/* DATA-data set specifies an input data set (_LAST_ is the */
/* default if none is specified). */
/* */
/* NEWVALUE=number numeric parameter. The value that will replace */
/* missing values. The default is zero. It is also */
/* possible to specify a special or standard */
/* missing value. */
/* */
/* VAR and EXCLUDE specify which variables missing values are to */
/* be replaced for. If neither is specified, missing values are */
/* replaced for all numeric variables. Only one of VAR and EXCLUDE */
/* can be specified. */
/*-----------------------------------------------*/

/* The MAINPROC flag should be set to 1 for the main procedure */
/* source file. The SASPROC flag should be set to 1 if this is a SAS */
/* procedure being compiled. Be sure that these flags are set before */
/* include <wpdoc.h>, this h file contains all the necessary */
/* definitions for SAS/TOOLKIT interfacing. */
/*-----------------------------------------------*/
#define MAINPROC 1
#define SASPROC 1
#include <wpdoc>

/* Account for some slight differences between the first */
/* experimental tape and subsequent releases, so that this example */
/* will work with either. */
#include <wpconfig.h> /* Not on first experimental tape */
ifndef SFLP /* Not on first experimental tape */
define SFLP(h, l) SFLV(l)
define SFLV(h, l) SFLV(l)
define SAS_XPUTUT1 */
ifndef SAS_XPUTUT1 /* Called SAS_XPUTUT1 on first tape */
define SAS_XPUTUT1(prr, nvar, cptr) \ 
SAS_XPUTUT1(prr, nvar, 0, cptr)
define SAS_XPUTUT2 */
ifndef SAS_XPUTUT2 /* Called SAS_XPUTUT2 on first tape */
define SAS_XPUTUT2(prr, nvar, 0, cptr)

/* Grammar function prototype---------------------------*/
ptr nominate U_PARMS((void));
/* The U_MAIN symbol should be used exactly once within all source */
/* for the procedure. It will be expanded into host-specific code. */
/* The name that will be given to the SAS procedure executable (in */
/* this case, NOMISS) should appear in parentheses. */
/*-----------------------------------------------*/
void U_Main(NOMISS) {

{)
  int i;
 aycast rc;
  double newvalue;  /* Value to replace missing values */
  ptr infile, outfile;
  short * list4;  /* Pointers to lists of variables */
  short * list5;  /* Pointers to lists of variables */
  short * list6;  /* Number of variables in each list */
  int nvar4, nvar6;
  double * bufptr;  /* Pointer to read/write variable buffer */
  ptr xvgetptr;  /* Pointer returned from SAS_XVGET */
  ptr xvputptr;  /* Pointer returned from SAS_XVPUT */
  struct XORPSTR outstr;  /* File parameters to open output dataset */

  /*******************************************************************************
   ** UPROC is called to initialize the SAS/TOOLKIT SAS procedure **
   ** environment. This is a required call, and should be made exactly **
   ** once, and should be the first executable statement. The proc **
   ** structure will have been included based on the setting of the **
   ** MAINPROC and SASPROC flags above. The address of this proc **
   ** structure should be passed. **
   **----------------------------------------------------------------------**
  UPROC(&proc);

  /*******************************************************************************
   ** SAS_XPARSE should be called with the grammar function's returned **
   ** pointer, a NULL, and the proc structure address. The proc.error **
   ** value will be set to indicate parsing success. If it indicates **
   ** that parsing problems occur, we simply abandon ship. **
   **----------------------------------------------------------------------**
  SAS_XPARSE(nomiss(), NULL, &proc);
  if (proc.error > 0) XEXITERROR);
    SAS_XEXIT(EXITSYNTAX, 0);

  /*******************************************************************************
   ** We made it past parsing. Announce our intentions. **
   **----------------------------------------------------------------------**
  SAS_XPSLOG("NOTE; NOMISS -- example procedure "
    "written in C using SAS/TOOLKIT.");

  /*******************************************************************************
   ** Get the pointers to the VAR, EXCLUDE and (all variables) lists --*/
   ** from the statement structures in the proc header. Note that --*/
  list4 = SFLP(proc.head, 4);
  list5 = SFLP(proc.head, 5);
  list6 = SFLP(proc.head, 6);

  /*******************************************************************************
   ** Check that the user didn't specify both a VAR list (variables --*/
   ** to include) and an EXCLUDE list (variables to exclude). It is --*/
   ** hard to assign a meaning to this combination and handling it --*/
   ** would complicate later processing. --*/
  if (list4 != NULL && list5 != NULL)
    SAS_XPSLOG("ERROR: Only one of VAR and EXCLUDE can be specified.");
    SAS_XEXIT(EXITSYNTAX, 0);
  
  /*******************************************************************************
   ** Obtain the new value that is to replace missing values. If the --*/
   ** user didn't supply the NEWVALUE= parameter, use zero (the --*/
   ** second operand of SFF). --*/
  newvalue = SFF($3, 0, 0);

  /*******************************************************************************
   ** Get the fileids --*/
  infile = SFFILE(2);
  outfile = SFFILE(1);

  /*******************************************************************************
   ** If no output dataset was specified, use the input dataset. --*/

89
/*------ Use SAS_XOINFO to copy the input libname, member name and */
/*------ other Information supported by SAS_XOPEN. */
/*------ Note that passwords are not copied. */
if (outfile == NULL)
{
    rctype =;
    char8 type;
    char40 label;
    SAS_XOINFO(infile,NO_LIB,&outstr.libname);
    SAS_XOINFO(infile,NO_MEMBER,&outstr.member);
    outstr.opmode = NO_OUTPUT|NO_SEQ; /* Sequential output open */
    outstr.err = 0; /* Don't handle I/O errors */
    SAS_XOINFO(infile,NO_TYPE,&type);
    outstr.type = &type; /* put its address and */
    outstr.type = 0; /* length in the XOOPNSR */
    SAS_XOINFO(infile,NO_LABEL,&label);
    outstr.label = &label; /* put its address and */
    outstr.label = 40; /* length in the XOOPNSR */
    rc = SAS_XOPEN(&outfile, &outstr);
    if (rc != 0)
    {
        SAS_XPRLOG(rc);
        SAS_XEXIT(XEXITERROR,0);
    }
/* Set up the list of variables for which missing value */
/* replacement is to be done. This is the list from the VAR */
/* statement. If there was no VAR statement, set up a default */
/* list that contains all the numeric variables in the dataset */
/* except for those specified in the EXCLUDE statement (if any). */
rc = SAS_XVDFLST(infile, 1, &SFLD(4), 1, &SFLD(5));
if (rc != 0)
{
    SAS_XPRLOG(rc);
    SAS_XEXIT(XEXITERROR,0);
}
list4 = SFLP(proc.head,4);

/* Set up a list of all the variables in the dataset, except */
/* for those in the VAR list (as supplied by the VAR statement */
/* or set up above). */
/* Note that this list can have zero entries if all the variables */
/* in the dataset are numeric and all are selected either via */
/* the VAR statement or by default (no VAR or EXCLUDE statements). */
rc = SAS_XVDFLST(infile, 3, &SFLD(6), 1, &SFLD(4));
if (rc != 0)
{
    SAS_XPRLOG(rc);
    SAS_XEXIT(XEXITERROR,0);
}
list6 = SFLP(proc.head,6);

/* Now the two lists of variables are set up. */
/* List 4 contains the variables for which missing value */
/* replacement will be done. */
/* List 6 contains all other variables in the dataset (these will */
/* simply be copied across to the output dataset). */
/* Get the number of variables in each list. Note that either */
/* number (but not both) can be zero. */
ncvar = SPM(4);
ncvar = SPM(6);

/* There is no need to call SAS_XFILE since we aren't */
/* printing any of the variables. */
/* Get memory for the observation buffer. This buffer will hold all */
/* the variables for which we are doing missing value replacement. */
bufptr = (double *)SAS_WMEM(ncvar+4*sizeof(double));
if (bufptr == NULL) /* If we couldn't get the memory, */
    SAS_XEXIT(XEXITMEMORY,0); /* exit with appropriate error code. */
/** Initialize the gather input and scatter output processes for */
/** reading and writing observations. */
/** All the variables for which we are doing missing value */
/** replacement are placed in the observation buffer. For each */
/** variable, SAS_XVGETD sets up reading the variable from the */
/** input dataset. SAS_XVNAME obtains the NAMESTR which has */
/** information about the variable, and SAS_XVPUTD adds the */
/** variable to the output dataset. */
SAS_XVGETD(infile, nvar4+nvar6, &xvgetptr1);
SAS_XVPUTD(outfile, nvar4+nvar6, &xvputptr);
for (i = 0; i < nvar4; ++i)
{
    struct NAMESTR * namptr;
    SAS_XVGETD(xvgetptr, list4[i], *sizeof(double), NULL, sizeof(double), XV_HOFMT);
    SAS_XVNAME(infile, list4[i], &namptr);
    SAS_XVPUTD(xvputptr, namptr, *sizeof(double), NULL, 0);
}
/** Copy all other variables to the output dataset with */
/** Note that this means that the variables are */
/** probably not in the same order as they were in the input */
/** This could be avoided by processing each variable */
/** in input order, choosing either the XVGETD sequence or XVPUTD. */
if (list6 == NULL) /* If other variables exist */
    SAS_XVPUTD(xvputptr, infile, list6, nvar6);
/** Terminate the definition process for gather read and scatter */
/** write with SAS_XGETE and SAS_XPUTE. */
SAS_XGETE(xvgetptr);
SAS_XPUTE(xvputptr);
/** Read through the input dataset with SAS_XBYPACT and */
/** Even though this procedure doesn't allow BY */
/** groups, we can still use these routines -- they will just */
/** treat the entire dataset as one BY group. SAS_XBYPACT */
/** initializes for reading, and SAS_XBYPGET reads an observation. */
while (SAS_XBYPACT(infile) > 0)
    while (SAS_XBYPACT(infile) != NULL)
    {
        /*-- Place the observation values into our buffer with */
        /*-- Note that only those variables we are */
        /*-- processing are read into the observation buffer. */
        SAS_XBYPGET(xvgetptr, bufptr);
        /*-- For each variable in the observation buffer, use */
        /*-- SAS_ZMISS to see if the value is missing. */
        /*-- If so, replace it with the new value. */
        for (i = 0; i < nvar4; ++i)
        {
            if (SAS_ZMISS(bufptr[i]))
                bufptr[i] = newvalue;
        }
        /* Add the observation to the output dataset with SAS_XPUT */
        /* and SAS_XADD. */
        SAS_XPUT(xvputptr, Bufptr);
        SAS_XADD(outfile, NULL);
    }
/**-- If the procedure has reached this point, we have normal termi */
/**-- Call SAS_XEXIT with the XEXITNORMAL argument to indicate */
/**-- that we will not get control back after the call. */
SAS_XEXIT(XEXITNORMAL,0);
NAME: NOMISSG
TYPE: GRAMMAR
PURPOSE: Grammar file for PROC MODE

#include STUCCRM.

PROGRAM = ANYSTMT
ENDB,

ANYSTMT = NOMISSSTMT
VARSTMT
EXCLUDESTMT

--- STATEMENT DEFINITIONS---
GSTINIT ALLOCates & FIELDS IN THE STATEMENT STRUCTURE FOR THE
PROCEDURE, THE 1ST 5 FIELDS HOLD INFORMATION FOR THE OUT
DATASET, THE INPUT (OR DEFAULT) DATASET, THE NOVALUE
PARAMETER, THE Vn VARIABLE LIST, AND THE EXCLUDE
VARIABLE LIST, IN THAT ORDER.
THE 6TH FIELD IS FOR A LIST OF ALL THE VARIABLES IN THE
INPUT DATASET, THIS LIST CANNOT BE SPECIFIED BY THE USER.
IT IS USED TO COPY THE VARIABLES TO THE OUTPUT DATASET.
GDSFLt DESCRIBES THE DEFAULT INPUT DATASET, THE NUMBERS MEAN,
IN ORDER:
2 - INFORMATION IS STORED IN FIELD 2 OF THE STATEMENT
STRUCTURE.
1 - THE DATASET IS INPUT
1 - THIS DATASET SHOULD BE USED TO LOOK UP VARIABLES
4 - RECORD-LEVEL ACCESS CAN BE USED, SINCE THE PROCEDURE
DOESN'T REREAD THE DATASET OR USE BY GROUPS.
2 - “ACCESS2”, ALWAYS 2 (PER THE DOCUMENTATION)
4 - ONLY SEQUENTIAL ACCESS IS USED (THIS PROCEDURE READS
STRAIGHT THROUGH THE DATASET ONE TIME ONLY).
0 - EXIT PROCEDURE IF I/O ERRORS OCCUR WITH THIS DATASET
0 - HONOUR ”WHERE” STATEMENTS AND ATTRIBUTE STATEMENTS

NOMISSSTMT = @PROCINIT
GSTINIT(6)
"NOMISS" NOMISSOPTS*
GDSFLT(2,1,1,4,2,4,0,0)
GSTINEND

92
--- THE FIRST BDS DESCRIBES THE OUTPUT DATASET. THE NUMBERS MEAN, 

- 1 - INFORMATION IS STORED IN FIELD 1 OF THE STATEMENT structure. 

- 2 - THE DATASET IS OUTPUT. 

- 0 - THIS DATASET SHOULD NOT BE USED TO LOOK UP VARIABLES. 

- 2 - MEMBER-LEVEL ACCESS IS REQUIRED (ALWAYS THE CASE FOR AN OUTPUT DATASET). 

- 2 - "ACCESS2", ALWAYS 2 (PER THE DOCUMENTATION). 

- 4 - ONLY SEQUENTIAL ACCESS IS USED (ALWAYS THE CASE FOR AN OUTPUT DATASET). 

- 0 - EXIT PROCEDURE IF I/O ERRORS OCCUR WITH THIS DATASET. 

- 0 - HONOUR "WHERE" STATEMENTS AND ATTRIBUTE STATEMENTS. 

--- THE SECOND BDS DESCRIBES THE INPUT DATASET. THE NUMBERS ARE THE SAME (AND MEAN THE SAME) AS FOR THE BDSDFLT DATASET. 

--- BPARAM DESCRIBES THE "NEWVALUE" STATEMENT, WHICH PROVIDES THE VALUE WHICH IS TO REPLACE NUMERIC MISSING VALUES. NOTE THAT THE REPLACEMENT VALUE CAN ITSELF BE A MISSING VALUE, ALLOWING THE REPLACEMENT OF ONE SPECIAL OR STANDARD MISSING VALUE WITH ANOTHER. THE NUMBERS MEAN: 

- 1 - INFORMATION IS STORED IN FIELD 3 OF THE STATEMENT structure. 

- 1 - THE VALUE OF THE PARAMETER IS NUMERIC.

NONMISSDPTS =

("OUT" = DSFIELD BDS(1, 0, 2, 2, 4, 0, 0))
| ("DATA" = DSFIELD BDS(2, 1, 4, 2, 4, 0, 0))
| ("NEWVALUE" = BPARAM(3, 1) (NUMBER|MISSING))

VARSINT =

"WAR" VARLIST,

EXCLUDESTMT =

"EXCLUDE" VARLIST