A Technique for Importing External Data into a Relational Database Using PROC SQL, the Pass-Through Facility and the Macro Language

Paul Doucette, PAREXEL International Corporation, Media, Pennsylvania

Abstract

Data conversions are not fun, no doubt about that. But if you're lucky, you can usually learn a thing or two from them. This effort was no different. It was not a lot of fun but a lot was learned from it, including a few pretty neat PROC SQL and macro techniques. Credit must be given to SAS Tech Support for providing the basis from which this code was developed.

The goal of this conversion was to create an automated process (the "bridge") that extracted recently entered or updated safety (adverse event) data stored in a clinical data management system (Oracle tables) residing on a VAX and migrated it to an adverse event tracking system built on a SQL/DS database residing on an IBM. Once the data was extracted from the Oracle database and exported to the IBM, SAS was used to perform the data conversion and insert the data into the adverse event (SQL/DS) database. This was accomplished through the use of PROC SQL statements contained in macros. The macros generated and executed standalone SAS programs that contained the SQL code used to insert, update and delete data to/from the database. The SQL code checked database status codes and performed rollbacks when errors were encountered during the database update steps.

All of the code was written in SAS 6.07 in the CMS environment and was developed using the Display Manager interface. The PROC SQL Pass-Through facility was used to directly access the SQL/DS database tables.

Introduction

The goal of the technique outlined in this paper is to import data into a relational database using SAS. SAS was chosen because most of the reports and analysis programs already in use with the existing database were coded in SAS. Now I don’t profess to be an expert with all of the areas touched on in this paper, so you may certainly find a better and or easier way to do this, but this worked well and enabled me to polish my PROC SQL and macro skills.

At this point, you may be asking yourself, "Why didn’t you just use SAS/ACCESS?" and it's a question I asked myself from the very start. First, though it was installed, it was not available to us as we were contractors "on the outside, dialing in" and getting the proper accounts and privs, views, etc. set up would've have held us up as we were up against extremely aggressive timelines (aren't we always). Second, it's been a while since I've worked with ACCESS and I wasn't sure that I could programmatically get ACCESS to conditionally insert, update and modify tables based on data-driven parameters. Besides, tech support said this was the way to go!

Process Overview

To help give you a better idea of what we were trying to do, Figure 1, below, shows the process (greatly simplified) that was followed for the entire conversion. The portion of the process being examined in this paper is a subset of the part of the process that starts with the data being “converted” by SAS and “inserted” into the database. The program that performed this process will be referred to as “DBMAN”.

**Figure 1 - Process Overview**

Extract data from Oracle (VAX/Alpha)  
Transfer data (ASCII text) to IBM  
Insert data into SQL/DS database  
Convert data and reformat

**Figure 2 - PIDn Dataset structure**

<table>
<thead>
<tr>
<th>PATID</th>
<th>TABLE</th>
<th>STATUS</th>
<th>LABVARn</th>
<th>DMOGVARn</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>DEMOG</td>
<td>UPD</td>
<td>null</td>
<td>dmgdata</td>
</tr>
<tr>
<td>10000</td>
<td>DMOGAUD</td>
<td>INS</td>
<td>null</td>
<td>dmgdata</td>
</tr>
<tr>
<td>10000</td>
<td>LABAUD</td>
<td>INS</td>
<td>labdata</td>
<td>null</td>
</tr>
<tr>
<td>10000</td>
<td>LABAUD</td>
<td>INS</td>
<td>labdata</td>
<td>null</td>
</tr>
<tr>
<td>10000</td>
<td>LABAUD</td>
<td>INS</td>
<td>labdata</td>
<td>null</td>
</tr>
<tr>
<td>10000</td>
<td>LABAUD</td>
<td>INS</td>
<td>labdata</td>
<td>null</td>
</tr>
<tr>
<td>10000</td>
<td>LABAUD</td>
<td>INS</td>
<td>labdata</td>
<td>null</td>
</tr>
<tr>
<td>10000</td>
<td>LABTEST</td>
<td>UPD</td>
<td>labdata</td>
<td>null</td>
</tr>
<tr>
<td>10000</td>
<td>LABTEST</td>
<td>DEL</td>
<td>labdata</td>
<td>null</td>
</tr>
</tbody>
</table>

Once the datasets were set up properly, the DBMAN program inserted the data into the appropriate tables. It accomplished this by processing each individual patient’s dataset and separately committing the database changes only when the entire set of the patient’s data was successfully processed. Since we didn’t want to commit any data before the entire patient’s data was successfully entered into the database, it was necessary to set up the data in this fashion. This patient/table dataset structure enabled us to write a macro that could cycle through all of the data for a single patient, committing when successful and rolling back when an error was encountered.

A slimmed down version of the entire program appears in Appendix A. Each section of the code will be examined in the following paragraphs.
The first DATA _NULL_ , see lines 15-63 of the program code in Appendix A, creates some constants that are used later in the program. Lines 18-56 create some date and time literals which were used to identify the (current) date and time the tables were updated. This was necessary since these variables were defined as character fields in the database and SAS date/time variables are numeric. The formatting of these character fields also prevented us from doing a straight numeric-to-character conversion. Additionally, several more macro variables, including LOGID and CHNGREAS, were created to record the fact that changes made to the database were made by this automated process.

Line 17 uses the CALL SYMPUT statement to create the NUMOBS macro variable that is used to control the number of times the main looping construct is executed.

17 call symput ("numobs",n);

The number stored by this variable was generated from the number of observations in a dataset called PIDFILE (see line 16). PIDFILE was created in the data manipulation program, which is executed prior to DBMAN, and stored one observation per unique patient id (PATID). Armed with the knowledge of how many patients there were we could proceed to executing the main looping construct.

Lines 57-61 again use the CALL SYMPUT statement to declare the remaining macro variables that are used throughout the rest of the code. As this was an automated process, these variables were necessary to record the fact that the process was responsible for the data entry being done and to record the dates and times the data was manipulated. Additionally, these fields were required by the database tables and, thus, needed to be entered for a successful database entry to be executed.

Note that it is necessary to include a RUN at the end of this DATA NULL step to force the assignment of the NUMOBS macro variable, as well as the others, by the SYMPUT statement. See the SAS Guide to Macro Processing for a complete explanation of the SYMPUT statement.

Main Looping Construct (%do, &i)

Lines 66 and 67 begin the DOALL macro and the main looping construct (%do) which control the entire database manipulation process.

66 %macro doall;
67   %do i=1 %to &numobs;

The %do loop will execute once for each observation in the PIDN dataset using the previously created NUMOBS macro variable.

The %do loop counter "i", which is an implicitly created macro variable, plays an important role in the process and is called several times within the loop. In the sort in line 68, it is used to identify the patient data file PIDn.

68 proc sort data=pid&i;

In line 70 &i is appended to the filename(s), TEMP1…TEMPn, of the files that are used to store the SQL code generated by the DOALL macro.

70 filename temp&i "temp&i sas";

These files are then called via an include statement located in the RUNCODE macro (see line 164-170 of Appendix A). Figure 3 below is a very basic flowchart of the steps that generate the TEMPn programs.

Figure 3 - Program Flow Diagram

Lines 72 and 73 also contain references to the &i variable and are used to identify the proper external file (TEMPn) and SAS datasets to be used for the particular patient’s data being processed.

72 file temp&i;
73   set pid&i end=last;

Pass-Through Code

Lines 75-79 contain a short if statement that executes once for the first observation in the PIDn dataset and includes a put statement that writes the code that begins the PROC SQL statement. This statement includes the pass-through code (line78) that “opens” the database for processing.

75 if _n_=1 then do;
76   put '%macro load;%let flag=0;';
77   put 'PROC SQL;' /
78       'CONNECT TO SQLDS (USER=ABCDE
79          PASSWORD=UVWXYZ );';
80 endif;

The DATA _NULL_ step that starts on line 71 contains all the PROC SQL code that is inserted into the TEMPn files. Note that the code included in Appendix A only demonstrates the code necessary to process the lab data. This code was repeated for a total of seven more tables which processed additional information including, among others, reaction, suspect drug, and demographic information.

The PUT Statements Generating PROC SQL Code

Starting on line 81, a series of put statements generates an INSERT statement that inserts the appropriate variables into the LABAUD table. Line 80 is the “if” portion of an if-then-else statement that queries the TABLE variable to determine what table is to be processed. This is important as the table structure varies from table to table (obviously). This if-then-else statement is continued on lines 101 and 149.

Note that the placement of the quotes and double quotes is very important as this governs what actually gets written out to the external file by the PUT statements. This was quite a tedious and time consuming effort but once the code was completed and working for one table, it was just a matter of changing table and variable names to get it to work with the other seven tables.
In lines 81-84, the single quote placement is very straightforward as the goal of these lines is to generate some straight text with no variable values or embedded quotes.

81 put 'execute(insert into libref.labaud ';
82 put '{patid, labvar1, labvar2, labvar3, labvar4, labvar5, ';
83 put 'labvar6, logid, dedate, detime, ';
84 put 'chlogid, chgdate, chgtime, recstat, chngreas) values ('000001', '9999', 'mmol', '
100.3', 'data', 'data', 'data', '
PGD', '100997', '
01:00:00', 'PGD', '
052098', '01:00:00', '
1'; XXXX ADDENDUM 052098') by sqlds;"
85 put 'execute(insert into libref.labaud ';
86 put '{patid, labvar1, labvar2, labvar3, labvar4, labvar5, ';
87 put 'labvar6, logid, dedate, detime, ';
88 put 'chlogid, chgdate, chgtime, recstat, chngreas) values ('000001', '9999', 'mmol', '
100.3', 'data', 'data', 'data', '
PGD', '100997', '
01:00:00', 'PGD', '
052098', '01:00:00', '
1'; XXXX ADDENDUM 052098') by sqlds;"

execute(insert into libref.labaud
(patid, labvar1, labvar2, labvar3, labvar4, labvar5, labvar6, logid, dedate, detime,
chlogid, chgdate, chgtime, recstat, chngreas) values
('000001', '9999', 'mmol', '
100.3', 'data', 'data', 'data', '
PGD', '100997', '
01:00:00', 'PGD', '
052098', '01:00:00', '
1'; XXXX ADDENDUM 052098') by sqlds;

The Double Ampersand (...varname)

Also used in this code, as seen on lines 89-91, is the dreaded double-ampersand macro variable reference.

91 put '{'&&detime +(-1) '}', '&logid', ';
91 put '{&&dedate', '&detime';
91 put '{&voidstat', '&chgreas'}) by sqlds;"

This can be an extremely frustrating and confusing (albeit extremely necessary) feature for even the more experienced macro coder. Basically, this is necessary because of the end result we are trying to achieve in the external programs (TEMPn) which is to have the macro variable resolved to it's actual value and surrounded by single quotes. If we only use one ampersand, we'll get the actual variable name (i.e. dedate or dtime) enclosed in quotes*. To resolve the variable to it's value, rather than it's name, you need the double ampersand. The SAS Guide to Macro Processing does a sufficient job of explaining it and I've also found the Macro Tips and Techniques guide to be very useful in demonstrating these, and other, macro techniques.

*This was true when using 6.07 in the CMS environment, but upon running this code in the VAX/OpenVMS environment using 6.09, a single ampersand worked fine.

Error Trapping

Lines 92-99 represent the error trapping part of the code.

92 put '%put &sqlxrc;';
93 put '%put &sqlmsg;';
94 put '%if &sqlxrc %then %let flag=1;'
95 put '%if %flag %then %do:'
96 ' execute(rollback) by sqlds;'
97 ' reset noexec;''
98 ' reset errorstop;''
99 '%end;';

If any SQL errors occur during the execution of the PROC SQL statements generated in lines 81-91, this code will rollback any changes made to the database since the start of the last PROC SQL statement. Looked at a different way, because a commit is not executed until the end of the PROC SQL statement, the rollback would cancel any changes made to the database for this particular patient. In addition, the RESET NOEXEC option used in conjunction with the ERRORSTOP option will prevent any further SQL statements within this PROC SQL from being checked syntactically and executing. If this occurs, when the subsequent SQL statements for this patient are compiled, they will generate warnings stating that the code will not be executed because of the NOEXEC option.
As it was especially important when debugging these SQL errors, it is important to note here that when looking at any errors you encounter in SAS, you must find the first error encountered and debug from that point down. As any SAS coder can tell you, one error can cause many more errors in subsequent (and very often correct) code.

When looking at this code, you’ll notice a bit of overkill in the use of the FLAG variable. This method was the way it was recommended to me and I left it as is because of the “if it ain’t broke, don’t fix it” adage. You could have easily left out line 94 which assigns a value to the FLAG variable if a SQL return code is returned and in line 95, replaced the “&flag” with “&sqlxrc” and it will work just the same. The main reason for using the FLAG variable is that it is somewhat easier to understand and manipulate than a macro variable.

One last word about the error-trapping section. You’ll notice that we used the SQLXRC and the SQLXMSG macro variables. These are system-generated variables that contain standard SQL return codes and the associated messages generated by the host SQL, in this case I/SQL, system. Or at least they’re supposed to. As we quickly found out, there is a known bug (I can say that because tech support acknowledged it) with version 6.07 in the CMS environment that prevents an accurate message from being returned by the SQLXMSG variable. The text of the message seemed to only include those characters of the message that were capitalized. In some cases, this was enough to decipher the message but in others (most of them) it was necessary to refer to the SQL/DS system manual to figure out what the message was that corresponded to the numeric SQL return code. The codes returned by SQLXRC were fine and you should check with tech support if you’re concerned about this “feature” in your database and operating system.

Moving On From the LABAUD Table

We’ve now examined all of the code responsible for updating the LABAUD table. Since this was an audit table and it recorded and retained every single update to the database, it was the easiest type of table to code for because there was only the requirement to insert data. The “live” data tables were different in that they required update and delete actions as well as insert. So now we need to add some conditional processing based on what type of observation was being processed for what specific table. Without getting into the details of this process, we took the eight different tables, excluding the audit table for each, and coded for the three types of data entry. Add to that the 8 audit tables and you’ve got 32 different database manipulation scenarios that needed to be coded to insert, update and delete data as appropriate.

Conclusion

At this point, let’s look at a before and after snapshot of the entire section of the DOALL macro code from DBMAN that generated an externally stored, standalone SAS program

Before:

```
set pid&_i end=last;
by table status;
if _n_=1 then do;
  put '%macro load;%let flag=0;';
  put 'PROC SQL:'/
    'CONNECT TO SQLDS (USER=ABCDE
    PASSWORD=UVWXYZ )';
  end;
if table='LABAUD' then do;
  put 'execute(insert into libref.labaud ';
  put '(patid, labvar1, labvar2, labvar3, labvar4, labvar5, ';
  put 'labvar6, logid, dedate, detime, ';
  put 'chglogid, chgdate, chgtime, recstat, chngreas) values ';
  put '("" patid +(1) "," labvar1 +(1) "," labvar2 +(1) ",";
  put " labvar3 +(1) "," labvar4 +(1) ",";
  put " labvar5 +(1) "," labvar6 +(1) ",";
  put " delogid +(1) "," dedate +(1) ",";
  put " deltime +(1) "," &logid ",";
  put "&&&date', &&&time';
  put "&recstat', &&chngreas')by sqlds;';
  put '%put &sqlxrc;';
  put '%put &sqlxmsg;';
  put '%if &sqlxrc %then %let flag=1;';
  execute(rollback)by sqlds;
  ' execute(rollback)by sqlds;
  ' reset noexec;'
  ' reset errorstop;'
  ' %end;';
  end;
  .
  .
  .
  if last then do;
    put 'quit;';
    put '%mend load;';
    put '%load;';
  end;
%end;
run;
%mend doall;
```

If the preceding code were executed for the first patient (&i=1), then the resulting code would be stored in the file “TEMP1 SAS” and would look like this:

After:

```
%macro load;%let flag=0;
PROC SQL;
CONNECT TO SQLDS (USER=ABCDE
PASSWORD=UVWXYZ );
execute(insert into libref.labaud ';
(patid, labvar1, labvar2, labvar3, labvar4, labvar5, ';
labvar6, logid, dedate, detime, ';
chglogid, chgdate, chgtime, recstat, chngreas) values ';
('00000', 999,'mmol',
'chardata','chardata',
'chardata','chardata',
'chardata','chardata',
'LOGID', '051998',
'10:00:00', 'CHGLOGID',
'052098', '13:59:59'))by sqlds;
%put &sqlxrc;
%put &sqlxmsg;
%if &sqlxrc %then %let flag=1;
%if &flag %then %do;
execute(rollback)by sqlds;
```

Conclusion

At this point, let’s look at a before and after snapshot of the entire section of the DOALL macro code from DBMAN that generated an externally stored, standalone SAS program
reset noexec;
reset errorstop;
%end;
.quit;
%mend load;
%load;

The final step in the process is to execute the external programs TEMP1…TEMPn. This is accomplished when the RUNCODE macro in lines 164-168 is executed in line 170.

```
164 %macro runcode;
165 %do j=1 %to &numobs;
166 %include temp&j;
167 %end;
168 %mend runcode;
169
170 %runcode;
```

This macro calls and runs the external programs created in the main body of the program. Again, the NUMOBS macro variable is used to control how many times the %do loop is executed and, thus, how many external programs (TEMP1…TEMPj) are included and executed. In the same manner that we used the &i macro variable from the first %do loop, a second macro variable, &j, is created from the RUNCODE %do loop and is used for a similar purpose.

In summary, what we have looked at here includes using macro code to generate externally stored, independently executed programs that contain SQL code used to update, insert and delete data from tables in a relational database. Error processing was handled by examining the SQL macro variable values generated by the host databases SQL language. We utilized the suppress macro function to create global macro variables and used the PUT statement to write macro-generated code to the external programs. The PROC SQL Pass-Through Facility was used to access the database.

* Thanks Jyo and Ray.

Author:  Paul Doucette
Manager, Programming
PAREXEL International Corporation
Rose Tree Corporate Center
1400 N. Providence Road, Suite 2000
Media, PA  19063-2043
(610)  565-2622, X2262
Email:  paul.doucette@parexel.com

References: SAS® Language, Reference
SAS® Guide to the SQL Procedure
SAS® Guide to Macro Processing
SAS® Macro Facility Tips and Techniques

---

Appendix A – DBMAN.SAS

```sas
1 /* the missing= option forces zero(s) to be entered into numeric database fields when dataset fields are null as the period '.' is not acceptable to SQL/DS */
2 OPTIONS DQUOTE MACROGEN SYMBOLGEN CENTER NODATE PS=55 LS=132
3   missing='0';
4
5 OPTIONS DQUOTE MACROGEN SYMBOLGEN CENTER NODATE PS=55 LS=132
6   missing='0';
7
8 /* CONSTANT DEFINITION - DATA NULL Step to create macro variables to store constant (default) character string values for insertion into database -
9   DEDATE, DELOGON, DETIME, CHGDATE, CHGTIME, CHGLOGON, CHGREAS,
10   VOIDSTAT
11 The creation of textual date/time fields was necessary because the dates and times were stored in a particular, character string format. */
12
13 data _null_;
14   set pidfile nobs=n end=last;
15   call symput ("numobs",n);
16   if last then do;
17     format t_mnth c_mnth t_dy c_dy c_yr $2. t_yr $4. c_dt $6.;
18     dte=today();
19     mnth=month(dte);
20     dy=day(dte);
21     yr=year(dte);
22     t_mnth=mnth;
23     t_dy=dy;
24     t_yr=yr;
25     if (mnth < 10) then
26       c_mnth='0'||substr(right(t_mnth),2);
27     else c_mnth=t_mnth;
28     if (dy < 10) then
29       c_dy='0'||substr(right(t_dy),2);
30     else c_dy=t_dy;

31 /* the missing= option forces zero(s) to be entered into numeric database fields when dataset fields are null as the period '.' is not acceptable to SQL/DS */
32 OPTIONS DQUOTE MACROGEN SYMBOLGEN CENTER NODATE PS=55 LS=132
33   missing='0';
34
35 OPTIONS DQUOTE MACROGEN SYMBOLGEN CENTER NODATE PS=55 LS=132
36   missing='0';
37
38 /* CONSTANT DEFINITION - DATA NULL Step to create macro variables to store constant (default) character string values for insertion into database -
39   DEDATE, DELOGON, DETIME, CHGDATE, CHGTIME, CHGLOGON, CHGREAS,
40   VOIDSTAT
41 The creation of textual date/time fields was necessary because the dates and times were stored in a particular, character string format. */
42
43 data _null_;
44   set pidfile nobs=n end=last;
45   call symput ("numobs",n);
46   if last then do;
47     format t_mnth c_mnth t_dy c_dy c_yr $2. t_yr $4. c_dt $6.;
48     dte=today();
49     mnth=month(dte);
50     dy=day(dte);
51     yr=year(dte);
52     t_mnth=mnth;
53     t_dy=dy;
54     t_yr=yr;
55     if (mnth < 10) then
56       c_mnth='0'||substr(right(t_mnth),2);
57     else c_mnth=t_mnth;
58     if (dy < 10) then
59       c_dy='0'||substr(right(t_dy),2);
60     else c_dy=t_dy;
```
c_yr=substr(t_yr,3);
c_dt=c_yr||c_mnth||c_dy;
drop dte mnth dy yr t_mnth c_mnth t_dy c_dy c_yr t_yr;

format t_hh c_hh t_mm c_mm t_ss c_ss $2. c_tm $8. tm time.;
tm=time();
hh=hour(tm);
mm=minute(tm);
ss=second(tm);
t_hh=hh;
t_mm=mm;
t_ss=ss;
if (hh < 10) then
  c_hh='0'||substr(right(t_hh),2);
else c_hh=t_hh;
if (mm < 10) then
  c_mm='0'||substr(right(t_mm),2);
else c_mm=t_mm;
if (ss < 10) then
  c_ss='0'||substr(right(t_ss),2);
else c_ss=t_ss;
c_tm=c_hh':'||c_mm':'||c_ss;
drop hh mm ss t_hh c_hh t_mm c_mm t_ss c_ss tm;
end;
call symput(“dedate”,c_dt);
call symput(“detime”,c_tm);
call symput(“logid”,’XXXXX’);
call symput(“chngreas”,’XXXXX ADDENDUM: ’||c_dt);
call symput(“recstat”,’0’);
run;

/***END OF DATA NULL STEP TO DEFINE CONSTANTS***********/

/***MACRO DOALL***************************/
%macro doall;
%do i=1 %to &numobs;
  proc sort data=pid&i;
    by tname status;
  filename temp&i “temp&i sas”;
data _null_; file temp&i;
    set pid&i end=last;
      by table status;
    if _n_=1 then do;
      put ‘%macro load;%let flag=0;’;
      put ‘PROC SQL;’;
      put ‘CONNECT TO SQLDS (USER=ABCDE PASSWORD=UVWXYZ );’;
    end;
    if table=’LABAUD’ then do;
      put ‘execute(insert into libref.labaud (patid, labvar1, labvar2, labvar3, labvar4, labvar5, labvar6, logid, dedate, detime, chglogid, chgdate, chgtime, recstat, chngreas) values ’;
      put “(‘” ;
      put “patid +(-1)” “, “labvar1 +(-1)” “, “labvar2 +(-1)” “, “labvar3 +(-1)” “, “labvar4 +(-1)” “, “labvar5 +(-1)” “, “labvar6 +(-1)” “, “delogid +(-1)” “, “dedate +(-1)” “, “&&logid’, “&&dedate’, “&&detime’, “&&recstat’, “&&chngreas’));by sqlds; 
      put ‘%put &sqlxrc;’;
      put ‘%put &sqlxmsg;’;
      put ‘%if &sqlxrc %then %let flag=1;’;
      put ‘%if &flag %then %do;’;
      put ‘execute(rollback)by sqlds;’;
      put ‘%put &sqlxrc;’;
      put ‘%reset noexec;’;
      put ‘%reset errorstop;’;
      put ‘%end;’;
    end;
    else if table=’LABTEST’ then do;
      %if %str(&status)=’DEL’ %then do;
        put ‘execute(delete from libref.labtest where 

6 of 7
"patid="" patid +(-1) "" and labvar1=" labvar1 +(-1)/
")by sqlds;"
put %put &sqlxrc;
put %put &sqlxmsg;
put %if &sqlxrc %then %let flag=1;"
put %if &flag %then %do;/
  execute(rollback)by sqlds;"
  reset noexec;"
  reset errorstop;"
%end;
end;
else if status='INS' then do;
  put 'execute(insert into libref.labtest '
  put '(patid, labvar1, labvar2, labvar3, labvar4, labvar5, '
  put 'delogon, dedate, detime) values '
  put "(" patid +(-1) "," labvar1 +(-1) "," labvar2 +(-1) ",";
  put "" labvar3 +(-1) "," labvar4 +(-1) ",";
  put "" labvar5 +(-1) ",";&logid");"
  put "&delate, &&date) by sqlds;"
  put %put &sqlxmsg;
  put %put &sqlxrc;
  put %if &sqlxrc %then %let flag=1;"
  put %if &flag %then %do;/
    execute(rollback)by sqlds;"
    reset noexec;"
    reset errorstop;"
  %end;
end;
else if status='UPD' then do;
  put 'execute(update libref.labtest set '
  put "labvar1=" labvar1 +(-1) ",labvar2=" labvar2 +(-1) ",";
  put "labvar3=" labvar3 +(-1) ",labvar4=" labvar4 +(-1) ",";
  put "labvar5=" labvar5 +(-1) ",";&logid");"
  put "&delate,&&date where ";
  put "patid="" patid +(-1) "" and labnum=" num +(-1);"
  put ")by sqlds;"
  put %put &sqlxrc;
  put %put &sqlxmsg;
  put %if &sqlxrc %then %let flag=1;"
  put %if &flag %then %do;/
    execute(rollback)by sqlds;"
    reset noexec;"
    reset errorstop;"
  %end;
end;
else if tname='REACAUD' then do;

...

if last then do;
  put 'quit;"
  put '%mend load;"
  put '%load;"
  end;
%end;
run;
%mend doall;
%doall;
%macro runcode;
%do j=1 %to &numobs;
  %include temp&j;
%end;
%mend runcode;
%runcode;
run;