A Flexible System for Conversion of DBF Tables into SAS®

Mel Widawski, UCLA, Los Angeles, California

ABSTRACT

The aim of this system is to import the tables from a dBase® database system as automatically as possible under adverse conditions. When the entry system is constantly changing and in control of another, you need a convenient way to bring in all available tables. Besides doing this, the system allows for the substitution of variable names, special transformations, and conversion of look-up tables to SAS® formats. It is completely modular, and allows for an environment where multiple programmers may be working on importing the same data. The program runs regardless of which modules are present. Switching from a test environment to the production, and back, requires only a single string change in the driver. You may also produce a mapping of variables to formats across data sets.

INTRODUCTION

Perhaps, if I let you know the situation that produced this system, you will understand the need for it. We needed to import all of the tables in a database that was maintained by a very busy individual, who was hard to reach. The database project started before the involvement of anyone familiar with statistical package programming, and this created a series of problems. The database was under construction and in flux; this meant that tables could appear, change names, change structure, and disappear at any moment. But research had to begin due to deadlines, and the database was the only source of information vital to the research. In addition, the database was extensive, as it consisted of around 35 tables at any given time. A number of the tables contained labeling information for other tables. There were three of us working on the project and eight additional programmers, statisticians, and researchers waiting for the data. The data had to be imported quickly. We would defer niceties, such as renaming variables to give them more reasonable names and transforming variables, and do them as analysis proceeded. I had the responsibility of organizing the system and creating any tools that were necessary to make it work.

The process of importing the data from DBF type files is relatively easy using SAS/ACCESS® to convert the tables to SAS data sets. But who wants to do 35+ cuts and pastes to do the job and end up with a program yards long? A macro would seem to be the appropriate solution. But what do we do about the transformations and renames necessary for each database table? These are specific to the specific tables. I have no particular fondness for looking through many lines of repetitive code to determine if transformations had been done for a table.

I decided the answer was a system that would automatically pick up any tables available and import them using any transformations and renames available. It would allow for small modules with this information so programmers could work on different parts of the problem simultaneously. For success I had to organize the pieces of the programs in directories and let the macro do the work. I no longer had to scan directories visually to see the changes, nor spend my time cutting and pasting lists or typing names. All I had to do was to write a few macros and a driver, and that is a lot more fun.

The entire system consists of a group of macros, a program to drive the macros, files containing portions of code, and a directory structure to hold it all together. The directory structure is of primary importance, so that will be discussed first. The driver code logically comes next, followed by the macros. The final pieces of the system are the program files specific to each table and grouped by function. Though not part of the system, I include a section on “Avoiding Problems” which presents guidelines to the database programmer. If followed, these guidelines will simplify your job greatly.

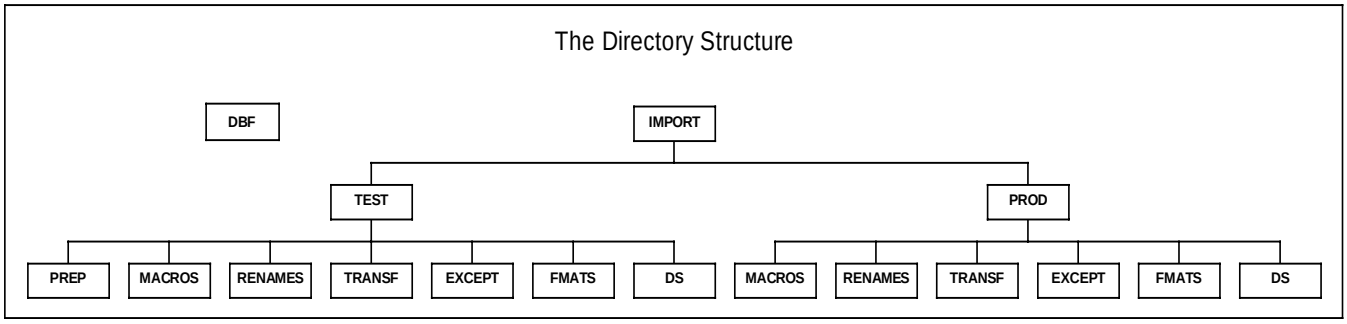
One key to the process was a macro to get a list of files of a given extension from a directory by just supplying directory pointers. This macro, called **GETLIST,** is paper 82 in a Coder’s Corner session at SUGI 23 called, “A General Purpose Macro for Obtaining a List of Files: Plus Macro Programming Techniques.”

It occurred to me that other people could benefit from this system both by using the programs I developed and by adapting the techniques and programs to other uses. Due to space limitations I will present only snippets of the programs in this article, but a complete sample program will be available on request.

STRUCTURE

As I mentioned, the whole system is held together by the directory structure. I hope the purpose of this structure will become obvious as the structure is revealed. The DBF files should be stored in a single directory. This is not entirely under your control, but since it is human nature to save files for a project under a single directory, you will probably not have to worry about it. By the way, I use the terms ***folder*** and ***directory*** interchangeably. I may also refer to a subdirectory when trying to indicate a directory or folder contained in another directory.

At this time I would like you to look at a diagram of the directory structure I am suggesting. You will find this diagram below.



This structure allows for ease of programming, order, the cooperation of multiple programmers, and quick location of program components. It also avoids the risk of accidentally deleting code. Notice that the TEST and PROD directories are nearly identical. I will now discuss the purpose of each directory.

DBF The DBF directory floats off by itself, and it contains the DBF files you are trying to import. It may exist on another machine, on a file server, or on your hard drive. This is the directory that you have no control over.

IMPORT You may want to name it after the project you are working on instead of IMPORT. I have given it the generic name IMPORT here as the system is used to import the database. The only file I like to store under this directory is a "read me" file containing information about the project. The IMPORT directory contains two main subdirectories TEST and PROD.

PROD & TEST With two exceptions the PROD directory mirrors the TEST directory in structure. The production directory, PROD, contains only fully tested code, while TEST contains code modules that are still under development. This scheme allows you to run the system under PROD and always get expected results. PROD and TEST both contain the following directories: MACRO, RENAMES, TRANSF, EXCEPT, FMATS, and DS. I store the driver programs at this level; with production drivers stored directly in PROD, and test drivers in TEST. All of the directories except DS contain SAS programs or program segments, while DS contains the SAS data sets produced by the system. It is possible to store the production output data sets anywhere you find convenient. Occasionally programs run under TEST will fail, and that is why the test system is kept separate. In a later section on "Driver Programs," I will show you a simple way to switch between TEST and PROD.

MACROS As the name implies, this directory contains macros. The macros in this library perform various tasks. These are the workhorses of the system performing the repetitive tasks and freeing you to write macros or analyze the data. The primary macro uses the ACCESS procedure to import the files, and performs transformations on the variables as necessary. More information on these macros will be presented in the section below titled "Macros and Macro Programming."

RENAMES Files in this folder contain sections of code used for renaming variables in the ACCESS procedure in the DBFACC macro. Each file in this directory will have the same name as the DBF file it applies to. Thus, all renames for person.dbf will be in person.sas in the RENAMES directory. It is not necessary to have a rename file for every table in the DBF directory. In fact, you should only include a file if you need to rename variables. If you want to see if any renaming was done for a data set, you simply look at the list of the files in RENAMES. This can be done using file manager or explorer.

TRANSF DATA step transformations reside in the TRANSF folder. Once again each set of transformations has the same name as the table it applies to, but these have the sas extension. These are rather small files unless there are a large number of transformations to perform. Only create a TRANSF file if variables in a table require transformations.

EXCEPT Files in this directory contain code used by the FMAT macro to provide exceptions to the rule that format names match the variables they are used with. This is necessary when two or more variables in the file have the same format applied to them. Here again, the names of the files match the names of the database tables. You will soon see how this is used to automatically bring in the code at the appropriate time.

FMATS Files in this directory contain code used in a FORMAT statement to connect the value labels to the variables. It is automatically generated by the FMAT macro based on the variable names and values in the CNTL data sets, as long as the variable names match the format names. Exceptions to this rule are supplied in the EXCEPT directory. The names of the files in this directory match the names of the tables.

DS The products of your labors, these are the SAS data sets created by running your programs and macros. They have the same filenames as the original tables, but they have the sd2 extension. You may want to keep a subdirectory under DS to hold the output data sets created by the CONTENTS procedure. These files have a number of purposes and as they are SAS data files this would be the logical place to store them.

PREP I keep the PREP subdirectory only in the TEST directory, and I use it to hold programs I run to determine information about the database. I find it useful to have a directory like this, but it is not required for the system to work.

DRIVER PROGRAMS

Driver programs are SAS programs that set up variables, do jobs that only need to be done once, and call macros. I store them in the TEST and PROD directories but you could store them in a separate directory if you wish

AUTCONV The main driver in the system, it calls the macros that create SAS data sets out of DBF files. This driver calls the DBFACC macro and the GETLIST macro to set up lists of files that DBFACC needs. Alternately, the calls to GETLIST can be generated from within the DBFACC macro.

FMATDRV The driver creates CNTL format data sets, and writes names of variables and formats for a FORMAT statement. It then combines the separate format control data sets into a single CNTL data set. And it creates a FORMAT catalog containing labeling information for the entire database.

FMATCHKD This program calls the macro that generates output showing the relationship between formats and variables. It also calls the GETLIST macro a number of times to create lists needed by the FMATCHK macro. You use it to aid in programming the value label control data sets. For this reason it probably belongs in the TEST directory.

VMATCHD This program calls the macro that generates output listing all variables for all the data sets in a project. It allows you to tell at a glance which variables match which files. It is useful in determining keys for merging the data sets.

The Auto Convert Driver

I will use the AUTOCONV driver as a model to demonstrate how the system works. It has three main functions. First it establishes the macro variables that define the environment. Next it calls GETLIST to create the lists of data sets in the DBF, RENAMES, and TRANSF directories. Finally, it calls the DBFACC macro that loops through the tables in the database and imports them into SAS. Here is a stripped down version of this driver, as I have stripped out the comments for brevity.

```
/* ****************************************** */
/* ****     Define the Environment        **** */
/* ****************************************** */

%let envrn=test;
%*let envrn=prod;

options MAUTOSOURCE
    sasautos=("d:\dg\sas\test\macros", sasautos)
    MRECALL;

%let majpath=d:\import;

%let dir=&majpath.\&envrn.\; /* dir .sas files */
%let sasdir=&dir.\ds\;       /* dir .sd2 files */
%let dbfloc=p:\dbf\;         /* dir .dbf files */

/* ****************************************** */
/* *******     macro calls here      ********* */
/* ****************************************** */
%getlist (in2,&dir,renames,&dir,renames,sas);
run;
%getlist (in2,&dir,transf,&dir,transf,sas);
run;
%getlist (in1,&dbfloc,,&dir,dbffils,dbf);
run;
%let numfs=&tnumfs;
run;

%dbfacc (&dbfloc,&dir,&sasdir,dbffils,
         &renames,&transf,&numfs) ;
run;
```

You can change from the test environment to the production environment easily; all you have to do is simply move the asterisk from one of the two %LET statements to the other in the driver above.

You can either include the code in the driver with a %INCLUDE statement or you can use an OPTION statement to establish a SASAUTOS library as I have here.

I always like to define things that change at the beginning of any program. This allows me to make one change at the top and avoid searching the whole program for every use of that specification. Paths, directory names, and file names are examples of things that change readily. This program is short, but it would be reasonable to string together a number of drivers, and at that point the program is long enough to make this practice worthwhile.

All you have to do is point to the appropriate directories and you are in business. The GETLIST macro makes a list of the files, and the DBFACC macro uses the lists to import the data. The DBFACC only needs the number of files for the dbf directory so that is the only one captured by this driver. The beauty of this is that the programs tend to be in short manageable chunks.

MACROS AND MACRO PROGRAMMING

Macros are the workhorses of the system; they take on the repetitive tasks and keep chugging away until they are done. They make programming worthwhile. I will present a discussion of the main macros in the system, what they are good for, and the information you need to supply them. Then I will present the DBFACC macro as an example of how they work.

DBFACC This macro creates SAS data sets out of DBF files. It does it over and over until all of the tables are converted. It will be discussed in detail below, but the following is the macro call.

```
%dbfacc (loc_dbf_files, dir_prog_segs,
         sas_dir_out_files,
         macro_varname_dbf_list,
         renames_file_list, transf_file_list,
         num_dbf_files);
```

GETFILES It does not do repetitive actions but takes the housekeeping burden off of the programmer's shoulders. Regardless of the flux of DBF tables, transformations, or renames; this macro relieves the programmer from the task of painstakingly comparing names and typing.

```
%getfiles (in_fileref, dir_input,
         input_sub_dir, dir_output,
         macro_varname_for_list, file_ext);
```

FMAT This macro creates CNTL format data sets, and writes names of variables and formats for a FORMAT statement. It then combines the separate format control data sets into a single CNTL data set. And it creates a FORMAT catalog containing labeling information for the entire database. It does this by combining the various look-up data sets, and comparing them with the output of the CONTENTS procedure. It self types the variables using information from contents.

```
%fmat(list_dat_files, num_dat_files, fmat_fls,
         num_fmt_files, name_cntl_ds, sas_dir,
         list_all_files, num_all_files,
         list_except_files);
```

FMATCHK This macro generates output showing the relationship between formats and variables. You use the output as an aid in programming. You can see the connections between the formats and the variables. You do have to read the output to determine what exceptions are needed.

```
%fmatchk (list_dat_files, num_dat_files,
         fmat_fls, num_fmt_files, sas_dir,
         list_all_files, num_all_files,
         list_except_files) ;
```

VMATCH This macro generates output listing all variables for all the data sets in a project. It allows you to tell at a glance which variables match which files. It is useful in determining keys for merging the data sets. You pass the output on to the analysts, or use it yourself for combining files.

```
%vmatch (sas_dir, list_all_files,
         num_all_files);
```

More on the DBFACC Macro

Here is a stripped down copy of the DBFACC macro for reading DBF tables and creating SAS data sets. I have stripped it of most comments and some of the PUT statements that give me step-by-step information on how the macro is performing.

```
%macro dbfacc (dbfloc, dir, sasdir, dbffils,
               renames, transf, numfs) ;

/* ************************************** */
/* ****             DBFACC               **** */
/* ************************************** */

 libname sasout "&sasdir";
 %local numf fname;

 %do  numf=1 %to &numfs; /* (1) */
      %let fname=%scan(&dbffils,&numf,%str( ));
      %put file is &fname;
      proc access dbms=dbf;
        create work.&fname..access;
        /* Create Access Descriptor for .dbf */

        path="&dbfloc.&fname..dbf";
        assign=yes;
                     /* (2) */
        /*  renames conditionally included  */
        %if %index(%str( &renames ),
                   %str( &fname ))>0
            %then %do;   /* (3) */
                %include
                   "&dir.renames\&fname..SAS";
            %end;
                     /* (4) */
        create work.&fname..view;
        /* Create .dbf View      */
          select all;
          list view;
         run;
         quit;
                     /* (5) */
      /* ******************************** */
      /* ****  creates sas data sets  **** */
      /* ****  with transformations   **** */
      /* ******************************** */
        data sasout.&fname;
              set work.&fname;

           /*  transforms conditionally   */
           /*  included here              */
                     /* (6) */
           %if %index(%str( &transf ),
                      %str( &fname ))>0
             %then %do;
                 %include
                    "&dir.transf\&fname..SAS";
             %end;
        run;
 %end;         /*   end of Do      */
%mend dbfacc;  /*   end of Macro   */
```

I placed **bold** numbers in the above example to aid in the discussion. The macro is one large do loop that executes once for each DBF table in the target directory **(1)**. The %SCAN macro function extracts each filename from the macro variable using a "blank" delimiter and a pointer that it increments each time through the loop.

The macro uses the ACCESS procedure to create a VIEW descriptor of each table. SAS variable names are assigned. The program checks to see if there is a rename data set associated with the DBF table, and if there is, brings the code into the ACCESS procedure **(2)**. The %INDEX macro function checks the list of rename data sets to see if the current table (&fname) matches any data set in the RENAMES folder. Both the list of renames (&rename) and the current table (&fname) are enclosed in a %STR function and bracketed by blanks, above **(3)**. This is so that the program can differentiate between filenames like PT and PTMED. Without this, the renames for PTMED could be brought in when the current table was PT.

At the end of the ACCESS procedure a VIEW is created. This is not yet a SAS data set, though the view can be used as if it were one. The SAS data set is created in the DATA step that follows, and this allows for the transformations. While a VIEW might suffice and certainly saves on storage, researchers need a stable data set to work with, and the data needs to be frozen at a time so results can be compared. Mission accomplished! The business of importing the DBF files is done. In the next section I will present information on the code for customized renames and transformations.

CODE SEGMENTS

These files contain pieces of code rather than whole programs. They provide the exceptions when importing data or creating formats. These compact single-purpose units are easy to maintain and update. You can tell at a glance if something is missing. These are the task- and target-specific workers that allow for the customization of the system. If they are missing you still get files, but with less readable variable names, and necessary transformations left to be done.

RENAMES

To reiterate, files in this folder contain sections of code used for renaming variables in the ACCESS procedure of the DBFACC macro. They follow the naming conventions of the database. It is most convenient to rename variables in the ACCESS procedure since at that point long names are still available and so the variables can be easily identified. Each file in this directory will have the same name as the DBF file it applies to. Thus all renames for person.dbf will be in person.sas in the RENAMES directory. If I want to see if any renaming was done for a data set, I would simply look at the list of the files in RENAMES. This can be done using file manager or explorer.

Variable or field names in DBF files can have as many as ten characters. SAS can only tolerate eight-character variable names. When SAS/ACCESS encounters a variable name that is greater than eight characters, it truncates the name. If that truncation matches one already in the system, then SAS/ACCESS does a further truncation and appends a number to the name. That number is incremented through the data set, and not just for the specific variable root. For example, consider the names: "finetimeav", "finetimemv", and "finetimevf". The first variable becomes "finetime", the second variable would also, but there is already one in the data set. Now since there are already four other names that matched after truncation in the data set, the second variable becomes "finetim5" even though it is only the second "finetime". You can live with this, but more reasonable names would be better. Fortunately, the original name is saved in the variable label. The following is an example of some rename code for one of the tables.

```
rename latncy_min    latncymn
          latncy_max    latncymx;
```

I think that this would be preferable to “latncy_m” and “latncy_8”. Since the rename command is in SAS/ACCESS you can use the long form of the name for these renames.

TRANSF

DATA step transformations reside in the TRANSF folder. They are brought in by the second step of the DBFACC macro. It doesn’t matter if none exists for a table — the program still works.

Some of the files were actually look-up tables to supply labels. The following is the transformation to prepare it to become a control data set readable by the FORMAT procedure as a CNTLIN= data set. This module changes the names to those recognized by SAS. It creates a truncated version of the names in case they are formatting character variables. “Start” and “label” are variables to define value formats for a control data set.

```
/* ************************************ */
/* ******    transf for CODES.DBF   ***** */
/* ************************************ */
   length  type $1  start $21  fmthold $8
           label $50  chrfmt $7;
   start=code;
   type=' ';
   fmthold=code_typ;
   chrfmt=fmthold;
   label=meaning;
   if fmthold=' '
               then put '*** bad fmthold *** '
                        fmthold= start= label= ;
   if code=' ' then put '*** bad code *** '
                        fmthold= start= label= ;
   if label=' ' then put '*** ?label?   *** '
                        fmthold= start= label= ;
   if fmthold^=' ';
   if code^=' ';
    /* ********************** */
    /* **  character formats ** */
    /* **   max len is 7 >> $** */
    /* ********************** */
   if substr(code_typ,1,7)
      in('MACHIN_','REASON_')
         then substr(chrfmt,7,1)=
                  substr(fmthold,8,1);
   keep type start fmthold label
        order code_typ chrfmt;

proc sort data=sasout.&fname;
     by code_typ start;
run;
```

Since the formats for a given variable were not grouped together, we sort the data set by “code_typ”, the original format name, and by “start”, the value being labeled. There are no other statements besides run following the transformation in the macro, so other procedures like the SORT procedure could be used at that point. This is another way to customize the system.

EXCEPT

The FMAT macro includes these exceptions while processing CONTENTS procedure output in preparation for merging with the format data sets. Thus, formats in the control data set are automatically given appropriate type, and the format statements are created correctly. Here is an example of matching three variables in a data set to the appropriate format. The format is SIDE_EFF, and the variables are SID_EFF1 through SIDE_EFF3. This assures the creation of appropriate FORMAT statements, which otherwise is accomplished automatically by the FMAT macro.

```
/* ************************************ */
/* ****** exceptions for formatting ***** */
/* ******      for DX.DBF           ***** */
/* ************************************ */
   if name in('SIDE_EF1','SIDE_EF2','SIDE_EF3')
                        then fmthold='SIDE_EFF';
   if name = 'FORM'     then fmthold='MED_FORM';
   if name = 'ROUTE'    then fmthold='MED_ROUT';
```

An exception is also necessary when the name of the variable is not the same as the format it needs. For example, FORM requires the MED_FORM format.

AVOIDING PROBLEMS

I believe that prevention is better than long and tedious programming solutions to problem data. With this in mind, I would like to start out by presenting a list of guidelines for database programmers whose data might have to be used for statistical analysis later.

In order for the guidelines presented below to make sense, you need to know how database programmers label values of variables (fields). There are two main ways that database programmers supply labels to the values of variables. The first way uses a table that contains three variables like “name”, “code”, and “label”. This corresponds to a CNTL data set produced by the FORMAT procedure with the variables FMTNAME, START, and LABEL. The other method database programmers use tables for labeling is to create separate tables. These tables hold each set of labels, and each table contains only value and label. The name of the table is linked to the field being labeled. Both types of these are called look-up tables.

Guidelines for Database Programmers

1. Confine variable (or field) names to eight characters or less.
2. If you are using look-up tables to supply labels to values, and the variable names of one table are linked to the values in a look-up table, then both the variable names and the corresponding values should be identical and restricted to seven characters in length.
3. If you are using look-up tables to supply labels, and the variable names in one table are linked to the names of other tables, then names of both should match. Also, please restrict the length of variable names and corresponding table names to seven characters.
4. If you have any of the above look-up tables and do not have matching names, please maintain a table of correspondence. And please keep the seven-character length restriction.
5. Please avoid the use of memo fields, and use character variables instead.
6. Please store all of the DBF files for your data base in the same directory, and store only those files in that directory.
7. Please avoid changing the names of your tables and variables.
8. If the records in some tables represent measurements on the same people at different points in time, then please include a date variable with the same name in each table. Also, the date recorded in those tables should be the date of the event, and not the date it was recorded.

Explanation of Guidelines

The purpose of the first guideline should be obvious, SAS variable names are restricted to eight characters in length.

Guidelines two through four are an easy way to ensure that format names will be the proper length even if the values labeled are character values. Remember to leave space for the dollar sign for character formats. If the format names are identical to the variable names, then you have an easy way to match them up.

Memo fields are not read by the ACCESS procedure. Attempts to bring them in with one of the available transfer packages require way too much post-processing.

You need to be able to locate the tables (point 6); consistency makes your job easier (point 7); and the data have to be meaningful (point 8).

There are probably more rules that should be included; but if the database programmer follows these simple rules, then importing the data becomes much more straightforward. Each deviation from these rules may make you produce hundreds of lines of code.

CONCLUSION

This system provides for automatic importing of DBF type files into the SAS system. Even though the system is automatic, customization is possible through the use of rename, transformation, and exception folders. If a file is stored in one of those folders the code will be automatically include in the program at the appropriate point. This has the advantage of reducing the amount of storage the system needs.

Since the code pertaining to each DBF table is stored in a separate file, this system supports a multi-programmer environment. A single programmer also benefits from having easily-locatable, small, and manageable chunks of code.

Maintaining separate test and production systems in parallel allows an ongoing programming effort to proceed with no loss in the ability to update the data. You don't have to worry when the test system temporarily breaks, because the production system is always ready to deliver the best data currently available. If a transformation module is missing, the program will still work. If an obsolete module is still there, with no associated table, the program will still work. This is not true of the single hard coded program, where an attempt to read a data set that is no longer there will crash the program. Consider the alternative of having to search through reams of code to find and remove the offending code.

You can create one large program with much repetitive code, but I wouldn't want to have to create or maintain it. In fact, if I had to create it to demonstrate my point, I would use a macro to put the pieces together.

I would like to summarize some of the positives and negatives of this system of programming.

Positives

1) A modular programming environment
2) Ease of use with multiple programmers
3) Flexible programs that adapt automatically
4) Removes much of the mundane work
5) Less chance of typing over other code
6) More robust with respect to errors
7) Module programming can be done by less experienced programmers
8) Automatic data-driven programs

Negatives

1) Chance of saving over the wrong module – they are named the same in each file
2) Requires the adherence to the structure
3) More files to look through when checking all the code

REFERENCES

SAS Institute Inc. (1990), *SAS® Guide to Macro Processing, Version 6, Second Edition,* Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1993), *SAS® Companion for the Microsoft Windows NT Environment, Version 6, First Edition,* Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1993), *SAS/ACCESS® Software for PC File Formats: Reference, Version 6, First Edition,* Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

A large portion of this system was developed while working for Dr. Donald Guthrie, UCLA. I would like to thank him for the opportunity to work on this project and the support he gave. This work benefited from discussions with Scott Komo, and Aaron Kaufman, both of UCLA. I would like to thank Barbara Widawski, without whose editing this manuscript would be illegible.

CONTACT INFORMATION

Mel Widawski
Office of Academic Computing
UCLA
Los Angeles, CA

Please contact through email at:

mel@ucla.edu