No experienced programmer doubts the value of documentation. At least, any programmer who did is likely to be made a believer by their first experience with a large poorly documented project or, worse, a project whose documentation is wrong. On the other hand, documentation is always a low priority task, and, therefore, is always being put off until after more important work is finished.

Moreover, probably because documentation is dull beyond belief, few programmers seem to enjoy it. The result is that documentation is often done poorly if it is done at all. One way to avoid the tedium of documentation and still get the benefits is to automate the programming function. This paper presents a set of conventions, techniques, and programs that allow the programmer to automate the documentation function to a considerable extent. More important, several of the techniques will produce documentation that must be correct. Handmade documentation is under no such constraint. Indeed, on a project of any size there will almost certainly be errors in any handmade documentation.

The purpose of documentation is to allow someone other than the programmer, or the programmer himself at a later date, to understand a set of programs and data well enough to be able to duplicate the results without error. In other words, documentation must provide the programmer with whatever information is necessary to run the programs, make modifications, interpret the algorithm if questions arise and do whatever else must be done to maintain the code. This means at least two things. First, each output must be linked with the data, programs and other actions that produced it. Second, each program must be described as to its purpose, structure and requirements. We will deal with each of these problems in turn.

1. The most obvious task of program documentation is, at least, to tie specific outputs to the programmer who produced them and, hopefully, the program or programs that produced them.

What is wanted, ideally, is an audit trail, that is, a way of tracking every piece of information through the entire programming process. This is obviously a demanding requirement. However, it is a requirement that many businesses, such as banks, meet as a matter of course. In fact, if all of the data manipulation are done by machine, then the places in the programming sequence that the data is altered must be recorded. For example, at Syntex we occasionally need to text edit program output. When that is done, the fact that that is done is text edited onto the output on the source line. Also recorded on the output source line, automatically in this case, is the job number and execution date of the program that produced the output and the programmer’s initials. This information is captured in the program fragment in Exhibit 1, which, in turn, is %INCLUDEed into the program. This information is placed in the output using put statements if the output is created in a DATA _NULL_ or FOOTNOTE statements if a PROC is used. An example of the later is: FOOTNOTE1 *SOURCE: BIOCLIP ASOURCE DATE T-1 1/5; This results in this footnote:

SOURCE: BIOCLIP FSG№1234 (01/23/86) T-1 1/5

which means that the output was produced at Syntex' BIOCLIP computing facility, by programmer FSG, that the the program was run on 01/23/86, that it was table 1 of the study and it was page 1 of the table's 5 pages.

This information can obviously be related to the program that produced it by a programming log. A programming log is a file that matches the job numbers of the programs run with the program name past programs needed to build the necessary data sets and other information. Once such a file is started it can be automatically maintained by %INCLUDEing a program fragment on the end of each program that appends the necessary information to a SAS data set (see Exhibit 2). While the program fragment need never be modified, the information passed to the fragment and then to the data set must be written separately for each program. One way of doing this is to set up a series of %LET statements at the beginning of the program and store the necessary information there (see Exhibit 3).

2. Programs can be described as to their purpose, structure and requirements in either of two ways.

First, the actions taken can be documented. For example, "A plot of A and B was produced." Second, the actions taken can be self-documenting. For example, "PLOT A*B;".

If some of the manipulation is not done by machine, then the places in the programming sequence that the data is altered must be recorded. For example, at Syntex we occasionally need to text edit program output. When that is done, the fact that that is done is text edited onto the output on the source line. Also recorded on the output source line, automatically in this case, is the job number and execution date of the program that produced the output and the programmer’s initials. This information is captured in the program fragment in Exhibit 1, which, in turn, is %INCLUDEed into the program. This information is placed in the output using put statements if the output is created in a DATA _NULL_ or FOOTNOTE statements if a PROC is used. An example of the later is: FOOTNOTE1 *SOURCE: BIOCLIP ASOURCE DATE T-1 1/5; This results in this footnote:

SOURCE: BIOCLIP FSG№1234 (01/23/86) T-1 1/5

which means that the output was produced at Syntex' BIOCLIP computing facility, by programmer FSG, that the the program was run on 01/23/86, that it was table 1 of the study and it was page 1 of the table's 5 pages.

This information can obviously be related to the program that produced it by a programming log. A programming log is a file that matches the job numbers of the programs run with the program name past programs needed to build the necessary data sets and other information. Once such a file is started it can be automatically maintained by %INCLUDEing a program fragment on the end of each program that appends the necessary information to a SAS data set (see Exhibit 2). While the program fragment need never be modified, the information passed to the fragment and then to the data set must be written separately for each program. One way of doing this is to set up a series of %LET statements at the beginning of the program and store the necessary information there (see Exhibit 3).

2. Programs can be described as to their purpose, structure and requirements in either of two ways.

First, the actions taken can be documented. For example, "A plot of A and B was produced." Second, the actions taken can be self-documenting. For example, "PLOT A*B;".

If some of the manipulation is not done by machine, then the places in the programming sequence that the data is altered must be recorded. For example, at Syntex we occasionally need to text edit program output. When that is done, the fact that that is done is text edited onto the output on the source line. Also recorded on the output source line, automatically in this case, is the job number and execution date of the program that produced the output and the programmer’s initials. This information is captured in the program fragment in Exhibit 1, which, in turn, is %INCLUDEed into the program. This information is placed in the output using put statements if the output is created in a DATA _NULL_ or FOOTNOTE statements if a PROC is used. An example of the later is: FOOTNOTE1 *SOURCE: BIOCLIP ASOURCE DATE T-1 1/5; This results in this footnote:

SOURCE: BIOCLIP FSG№1234 (01/23/86) T-1 1/5

which means that the output was produced at Syntex' BIOCLIP computing facility, by programmer FSG, that the the program was run on 01/23/86, that it was table 1 of the study and it was page 1 of the table's 5 pages.

This information can obviously be related to the program that produced it by a programming log. A programming log is a file that matches the job numbers of the programs run with the program name past programs needed to build the necessary data sets and other information. Once such a file is started it can be automatically maintained by %INCLUDEing a program fragment on the end of each program that appends the necessary information to a SAS data set (see Exhibit 2). While the program fragment need never be modified, the information passed to the fragment and then to the data set must be written separately for each program. One way of doing this is to set up a series of %LET statements at the beginning of the program and store the necessary information there (see Exhibit 3).

2. Programs can be described as to their purpose, structure and requirements in either of two ways.

First, the actions taken can be documented. For example, "A plot of A and B was produced." Second, the actions taken can be self-documenting. For example, "PLOT A*B;".

If some of the manipulation is not done by machine, then the places in the programming sequence that the data is altered must be recorded. For example, at Syntex we occasionally need to text edit program output. When that is done, the fact that that is done is text edited onto the output on the source line. Also recorded on the output source line, automatically in this case, is the job number and execution date of the program that produced the output and the programmer’s initials. This information is captured in the program fragment in Exhibit 1, which, in turn, is %INCLUDEed into the program. This information is placed in the output using put statements if the output is created in a DATA _NULL_ or FOOTNOTE statements if a PROC is used. An example of the later is: FOOTNOTE1 *SOURCE: BIOCLIP ASOURCE DATE T-1 1/5; This results in this footnote:

SOURCE: BIOCLIP FSG№1234 (01/23/86) T-1 1/5

which means that the output was produced at Syntex' BIOCLIP computing facility, by programmer FSG, that the the program was run on 01/23/86, that it was table 1 of the study and it was page 1 of the table's 5 pages.

This information can obviously be related to the program that produced it by a programming log. A programming log is a file that matches the job numbers of the programs run with the program name past programs needed to build the necessary data sets and other information. Once such a file is started it can be automatically maintained by %INCLUDEing a program fragment on the end of each program that appends the necessary information to a SAS data set (see Exhibit 2). While the program fragment need never be modified, the information passed to the fragment and then to the data set must be written separately for each program. One way of doing this is to set up a series of %LET statements at the beginning of the program and store the necessary information there (see Exhibit 3).

2. Programs can be described as to their purpose, structure and requirements in either of two ways.

First, the actions taken can be documented. For example, "A plot of A and B was produced." Second, the actions taken can be self-documenting. For example, "PLOT A*B;".
Clearly, the second procedure is by far the better one. Partially, this is because the work does not have to be done twice. But more important, if the SAS statements are self-documenting, then the documentation must be correct. On the other hand, comments can be and frequently are wrong.

The easiest way to document individual programs, therefore, is to write self-documenting programs, that is, complete and literate code. Literate programming code is well styled code. Writing literate code means doing such things as: planning programs well, keeping programs simple, and making the program logic obvious.

Unfortunately, even the best styled programs need comments. Partially, this is because SAS is not really structured for structured programming. More important, perhaps, no matter how well structured a program is, once it passes beyond a certain level of complexity it is impossible or almost impossible to understand it from the code itself. One way to help insure that comments will be correct is to keep the comments in or near the code itself. Comments in the program itself are much more likely to be modified when the program itself is modified. Comments elsewhere are not.

Comments on individual program segments should obviously be kept near the sections commented. Less obviously, comments on the entire program should be kept in the JCL or the top of the program and somehow distinguished from other comments. JCL allows two types of comments. First, a short line can be commented out by placing // in the first three columns. Second, comments can be placed in any line containing an operand. The comment must be placed to the right of the operand and a blank space. In Example 3 comments on the program as a whole are kept in the JCL in the lines beginning //.*. Comments on the JCL, such as ‘make changes here’ in Example 3, use the second style.

If this is done and the program is saved in a card image library it is relatively easy to automatically document all of the individual programs in the library. In this case automatic program documentation means writing a program that uses PROC SOURCE to dump the library to a flat file and DATA steps to identify those lines in the JCL that contains comments or information on non-system data sets. The procedure changes somewhat if OS is not used. When this is the case two or more styles of comments are needed. For example, comments on the program as a whole could be placed in asterisked boxes while comments on individual code fragments could use the slash asterisk format (i.e., /* . . . */).

Obviously, the easier it is to comment code the more frequently comments will be maintained. On the other hand, if comments are well structured, program comments can be manipulated by other programs. There are obviously tradeoffs here.

If the comments are well structured, it is possible to insure that the comments are correct. For example, if you are working on program ‘John’ it is of some value to know that you are really working on program ‘John’ and not program ‘Ralph’. Putting a ‘line in the program in the line ‘%LET PRGMNAME=’. Interestingly enough, it is possible to insure that these two lines are always on the same line. Moreover, these lines specified in the program is set up so it can be run by hitting a single key, and this is done consistently, the probability of the program being misidentified is extremely small. Moreover, it is possible to cross check this information. A SAS program can easily be written that, after the library is dumped with a PROC SOURCE, compares the name SAS identifies with the name identified in the JCL, and the name identified in the macro statement. Such a program can and should be run on a regular basis, say, weekly.

The example above illustrates an important concept: documentation is nothing more than the manipulation of character strings and numbers. Moreover, when the manipulations are relatively well understood, which is more often than not the case, there is no reason not to make the machines do it.

When programs are considered in isolation, the character manipulations can be performed relatively easily. More important, more difficult and much more susceptible to error is the documentation of entire libraries or projects. Project documentation is more than the documentation of individual programs, of course. It is also the documentation of the relations between programs.

Currently, there is no way of documenting a program library by running a single program. Such a program is clearly do-able but it would be difficult to write and I have yet to write it. Nevertheless, with only a moderate amount of work it is possible to write programs that do much of the work for you, and if they do not keep you from all errors, they will at least keep you from some of the most stupid ones.
In an OS operating system the relationships between the programs can be seen in the JCL of the individual programs, the use of two level SAS names, and the use of SAS key words such as "SET" to indicate whether the data is being read or written. It is obviously possible to write a program that traces the relations between files and programs and presents that information in any desired manner.

Figure 5 shows part of the output from one such program. The big problem in writing such a program is the large number of alternative ways that SAS can be told to read to write data. Still, it is possible to get much of the information with only a modest amount of programming by manipulating only the JCL and the two level SAS names. Again, the procedure is to use PROC SOURCE to dump the library to a flat file and use DATA steps to identify data set names in the JCL and their corresponding two level SAS names in the program body.

1Dale Peters, a SAS consultant, wrote the program in Exhibit 4. SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA.

```
EXHIBIT 1.

*THIS PRODUCES MACRO VARIABLES SOURCE AND DATE*;
GLOBAL SOURCE DATE;
DATA _NULL_;
IF _N_ = 1 THEN
DO;
   INFILE WORK JFCB=JOBNUM;
   CALL SYMPUT('SOURCE',SUBSTR(JOBNUM,24,B));
   DATE='('PUT(TODAY(),MMDDYYB. )')';
   CALL SYMPUT('DATE',DATE);
END;
ELSE STOP;
```

**EXHIBIT 2.**

```
OPTIONS DQUOTE ERRORABEND;
DATA JOBLOG;
   LENGTH COMMENT $ 80;
   DATE = PUT(TODAY( ), DATE7.);
   CLOCK = TIME();
   INFILE WORK JFCB=JOBNUM;
   JOBNUM=SUBSTR (JOBNUM,24,8);
   SOURCE=' SOURCE: BIOCLIP '[JOBNUM;
   PRGNAME = "PRGNAME";
   LASTPRGM = LASTPRGM;
   COMMENT = "COMMENT";
   IF PRGNAME = " THEN ABORT;
   PROC APPEND BASE=&JOBLOG JOBLOG NEW=JOBLOG;
DATA TEST;
   SET &JOBLOG JOBLOG END=EOF;
   IF PRGNAME = "PRGNAME" THEN KOUNT +1;
   IF EOF;
   INFILE WORK JFCB=JOBNUM;
   TESTVAR=SUBSTR (JOBNUM,24,8);
   IF INDEX(SOURCE,TESTVAR) = 0 THEN ABORT;
   LABEL KOUNT= 'ITERATION';
   TITLE1 = "NEW OBS IN DATA SET #JOBLOG";
   TITLE3 = "NEW OBS IN DATA SET #JOBLOG";
   FOOTNOTE1 = "NEW OBS IN DATA SET #JOBLOG";
   FOOTNOTE3 = "NEW OBS IN DATA SET #JOBLOG";
   PROC PRINT DATA=TEST LABEL;
   ID SOURCE;
   VAR PRGNAME KOUNT DATE CLOCK COMMENT LASTPRGM;
```
EXHIBIT 3.

// JOB (R7152,R7152,1RDA01), 'FSG A8-434',
// MSGLEVEL=(2,0)
%SASJOB EXEC SAS, OPTIONS='MACRO'
%SASDATA DD DSN=MEN.R7152.CLN.RS69216.S1100.I0187.SASLIB, MAKE CHANGES HERE
// UNIT=SYSBA,DISP=OLD,VOL=SER=SAS803
//TT31FOO1 DD DSN=MEN.R7152.CLN.TT31FOO1,
// UNIT=DISK,SPACE=(TRK,(500,500)),
// DCB=(RECFM=FBA,LRECL=133,BLKSIZE=19019),
// DISP=(NEW,DELETE,DELETE),VOL=SER=SWORK1
//CARDLIB DD DSN=MEN.R7152.FSG.REFER.CARDLIB,DISP=SHR
//PROGLIB DD DSN=MEN.R7152.CLN.RS69216.S1100.I0187.LIB,DISP=SHR
//*****************************************************************************
//** SAVE MEN.R7152.CLN.RS69216.S1100.I0187.LIB#CARDINTI ON INTER3
//** THIS PROGRAM PRODUCES LIST OF PT'S WITH CARDiac INTERCUR- *
//** RENT EVENTS, BY PT FOR EACH PATTERNMENT. *
//*****************************************************************************
//SYSIN DD *
OPTIONS DQUOTE NODATE NONUMBER;
%INCLUDE CARDLIB(DIAGPRNT) /NOSOURCE2;
%INCLUDE CARDLIB(MSOURCE) /NOSOURCE2;
%LET JOBLOG=SASDATA;
%LET PRGMNAME=CARDINTI;
%LET LASTPRGM=N/A;
%LET COMMENT =TO CREATE COUNTS OF INTERCURRENT EVENTS;
%LET KEEPCOPY =;
%LET DIAGPRNT = OFF;
EXHIBIT 4.

```
SET VAL SB ESC
SET ESC /
POINT '//' IN 0/50 NOL
READ STR SO USING *
SET VAL SO SUBSTR(SO,6)
S50
SET ESC S5B
```

EXHIBIT 5

**LISTING OF LIBRARY MEMBERS**

**AND CORRESPONDING ONE LEVEL AND TWO LEVEL NAMES**

**IF TYPE IS MISSING DATA SET IS PROBABLY PROGRAM FRAGMENT**

**THAT HAS BEEN PERCENT INCLUDED**

**MEMBER = INPUT15**

<table>
<thead>
<tr>
<th>DDNAME</th>
<th>TYPE</th>
<th>BOTHVAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>PROGLIB</td>
<td>MEN.R7152.CLN.RS99B47.RAW</td>
</tr>
<tr>
<td></td>
<td>JOBLOG</td>
<td>MEN.R7152.CLN.RS99B47.S1044.IO986.LIB</td>
</tr>
<tr>
<td>SASDATA</td>
<td>CARDLIB</td>
<td>MEN.R7152.CLN.RS99B47.S1044.IO986.SASTAPE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEN.R7152.FSG.REFER.CARDLIB</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>SASDATA.DATAFM15</td>
</tr>
<tr>
<td></td>
<td>INPUT</td>
<td>RAW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CARDLIB.JOBLOG</td>
</tr>
</tbody>
</table>

597