Presented at the Northwest 76 ACM/CIPS
Pacific Regional Symposium, Seattle, WA,
June 24 - 25, 1976

LBL-4876

Conf-760634--3

PROFESSIONAL LEVELS OF COMPUTER PROGRAM DOCUMENTATION

Fredric Gey

June 1, 1976

Prepared for the U. S. Energy Research and
Development Administration under Contract W-7405-ENG-48

LBL-4876

LBL-4876

PROFESSIONAL LEVELS OF COMPUTER PROGRAM DOCUMENTATION

Fredric Gey

Computer Science and Applied Mathematics
Lawrence Berkeley Laboratory
Berkeley, California 94720

June 1, 1976

LBL-4876

PROFESSIONAL LEVELS OF COMPUTER PROGRAM DOCUMENTATION

by

Fredric Gey

Computer Science and Mathematics Department
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Abstract

Computer program documentation is more than a collection of techniques for manipulating code for readability, more than a flow diagram of program logic, and more than a block of comments cards at the beginning of a module. A professional level of documentation derives from the systematic synthesis of technique tempered with good judgment and lucid composition. The overriding goal of documentation is understanding, by managers, by users, and by maintenance programmers. Documentation requires a level of precision rarely required in programming itself.

Key Words and Phrases

Computer Program Documentation, Coding Techniques, Documentation

CR Categories 2.2, 4.43

This report was done with support from the United States Energy Research and Development Administration.

This report was done with support from the United States Energy Research and Development Administration. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the United States Energy Research and Development Administration.

TABLE OF CONTENTS 19 MAY 76

1.1 INTRODUCTION

Recently I had the unfortunate experience of having to examine and evaluate a completely undocumented computer program. Not only were there 10 pages of FORTRAN code, but no written user's instructions existed. An important accounting function depended upon this program, and yet the only information available about the program was locked up in the heads of two people, the person who used the program for monthly accounting, and the person who was responsible for its maintenance. If misfortune were to fall to the latter, chaos would result. This state of affairs can be found to exist in almost any programming group, save those in which rigidly enforced documentation standards have been established by the group leader. For a small program as the above mentioned, the problem might not be significant, but when all the information concerning a large system which has taken several years to develop is also locked up in one brain, the problem is indeed significant. A case in point happened at this installation a few years ago when a computer programmer was killed in a motorcycle accident, and nearly six person-months were expended in a vain attempt to salvage his work.

Furthermore, as each programmer gains experience and remains at his job for a longer period of time (assuming he doesn't indulge in frequent job-hopping) the number of computer programs under his control and their relative complexity increases to saturation point. Many programmers agree that they usually cannot remember what they did in a given program for more than six months after they did the work. Just as businesses require written bookeeping proceedures to keep track of their day-to-day, week-to-week (or any other 'time frame') operations, a programmer to remain effective needs his own accounting system. The essence of such a system may be simply stated. It consists of a professional level of computer program documentation. The purpose of this paper is to define, in a loose way, what this phrase might mean.

1.2 THE ELEMENTS OF DOCUMENTATION

A computer program might be said to be fully documented if written descriptions are available which answer all questions concerning the following four elements

design
user's instructions
demonstration problems
programmer's instructions

For small programs, all of these elements might be taken care of via comment cards within the program. Most moderate sized programs can get by with the first three described in a single user's manual, and the last within the program as comment cards.

The extent to which these items are described is a matter of careful individual judgment. One-shot production programs might get by with little or no documentation. However, it has been my experience that production programs tend to be ressurected at just that point where the original programmer thought they were forever six feet underground. Thus the minimal documentation for such an effort should be extensive commentary within the program, so one can resuscitate the program without having to repeat the original effort.

Utility programs which will be used over and over, either by other programmers or by non-programmers, should have written instructions describing the use of the program (or subroutine as the case may be), as well as adequate internal documentation (in the form of comments) so that any programmer can maintain or modify the program to suit individual use. A user's manual for large utility programs which will be used extensively should include demonstration problems, or examples of the use of the program. A good rule of thumb is two examples, an easy one and a hard one.

A program as large as NASTRAN (the NASA computer program for engineering structural analysis) which consists of more than 150,000 source statements, mostly in FORTRAN and over 1000 distinct subroutines, requires all four elements to be described in separate manuals. Thus NASTRAN documentation is contained in the following

The NASTRAN Theoretical Manual
The NASTRAN User's Manual
The NASTRAN Demonstration Manual
The NASTRAN Programmer's Manual [3, 4, 5, and 6]

All this in addition to extensive comments within the actual code. nASTRAN deserves special mention because I consider it to be an example of the highest professional level of computer program documentation. (Kudos to Computer Science Corporation who developed it).

1.3 THE RIGHT COMMENT IN THE RIGHT PLACE

A pithy quote from Bill Hogan is appropriate to start this section **"a computer program without comment cards is like a rosetta stone without the Greek translation."** If we are given, then, that comment cards are necessary, what should be commented and where. The important factor is not how much commentary, for, to quote [1] 'a program can consist of 70 percent comments and still be undocumented." The key to good documentation is a lucid description of program flow, i. e. The right comment in the right place.

My experience has been that the most useful organization of comments is into the following three classes -

1. At the beginning of the program put a general, but fairly detailed description of program (or subroutine) flow, i. e. An algorithm description.

The depth of detail to which to carry the algorithm description is also a matter of individual judgment. The length of the program is not the determining factor, but rather the complexity -- the more complex the tasks performed, the more detail and clarity needed to describe the process for performing the tasks.

2. This should be followed by a complete description of the relevant variables utilized by the program, i. e. a dictionary of variables.

The importance of a variable dictionary cannot be overemphasized. Many subroutines can be understood with the aid of a variable dictionary even if no other documentation exists beyond user's instructions. My own preference is to organize variables into three distinct blocks, those associated with named or unnamed common blocks, those variables which are calling parameters to a subroutine, and those variables local to the routine, i. e. those which are not used outside the program.

For readability, it is best to use some fancy keypunching to set off the variable definition block from other parts of the program. My personal convention has been to place a * (star) in column 2 for this purpose. The definitions are most readable when set up in tabular format. The variable name can be started in column 10, and its definition in column 25. Subscripted or array variables should also have the meanings of the various subscripts defined.

Main programs or subroutines doing input-output should have a directory of I-O files included with the dictionary of variables.

The following is an example of the first two classes of documentation.

```
      SUBROUTINE CVRT360(MODE,WINP,NWD,ICODE,NDEL,IWOUT)
C
C    CVRT360 IS A SUBROUTINE TO CONVERT AN IBM-360 BINARY RECORD TO AN
C   EQUIVALENT CDC-6600 (OR 7600) RECORD TO BE USED IN FORTRAN PROGRAMS
C   OPERATING UNDER THE BKY SYSTEM.
C
C         CVRT360 CALLS
C              SUBROUTINE UNPACK
C                    WHICH UNPACKS 15 32-BIT 360 WORDS WHICH FIT
C                    EXACTLY INTO 8 60-BIT WORDS AND PLACES THEM
C                    RIGHT-JUSTIFIED (BUT OTHERWISE UNCHANGED) INTO
C                    15 60-BIT WORDS
C              FUNCTION ICNVRT(I)
C                    WHICH CONVERTS A 32-BIT IBM-360 INTEGER
C                    TO ITS CDC EQUIVALENT (TWO'S COMPLEMENT TO
C                    ONE'S COMPLEMENT TRANSLATION)
C              FUNCTION FCNVRT(I)
C                    WHICH CONVERTS A 32-BIT IBM-360 FLOATING POINT
C                    NUMBER TO ITS CDC EQUIVALENT
C              SUBROUTINE ACVRT
C                    WHICH CONVERTS FROM IBM-360 EBCDIC 8-BIT CHARACTER
C                    CODE TO CDC DISPLAY CODE VIA TABLE LOOKUP
C              FUNCTION IBITS
C
C************************
C*
C*         DICTIONARY OF VARIABLES
C*
C*         CALLING PARAMETER DEFINITIONS
C*         VARIABLE          DEFINITION
C*
C*         MODE              TYPE OF CONVERSION
C*              =0           UNPACK RECORD RIGHT-JUSTIFIED INTO 60-BIT
C*                           WORDS
C*              =1           INTEGER CONVERSION
C*              =2           FLOATING POINT CONVERSION
C*              =3           FULL RECORD CHARACTER CONVERSION
C*              =4           WORD-BY-WORD CONVERSION ACCORDING TO ICODE
C*
C*         WINP              ARRAY HOLDING INPUT RECORD OF 360 WORDS
C*
C*         NWD               NUMBER OF WORDS TO BE CONVERTED
C*
C*         ICODE(I)          WORD-BY-WORD CONVERSION CODE FOR ITH WORD
C*                   =-1     DELETE THIS WORD FROM OUTPUT ARRAY
C*                   =0      UNPACK RIGHT-JUSTIFIED IN A 60 BIT WORD
C*                   =1      FIXED POINT CONVERSION
C*                   =2      FLOATING POINT CONVERSION OF WORD I
C*                   =3      CONVERT 4 360 CHARACTERS INTO 4H FORMAT
C*
C*         NDEL              NUMBER OF WORDS TO BE DELETED IN WORD-BY-WORD
C*                           CONVERSION
C*
C*         IWOUT             OUTPUT ARRAY FOR CONVERTED RECORD
```

```
C*
C*
C*
C*        LOCAL VARIABLE DEFINITIONS
C*        VARIABLE       MEANING
C*
C*        UNPK()         15 WORD ARRAY TO HOLD UNPACKED WORDS
C*        FUNPK()        FLOATING POINT ARRAY EQUIVALENCED TO UNPK
C*
C*        NWDI           NUMBER OF 60 BIT WORDS COMPRISING WINP
C*
C************************
C
C
C
C
C
```

3. Finally, comments must be placed within the code itself.

This is best done by placing a short commentary in front of each collection of lines of code which perform a distinct task within the program flow.

A HANDY TIP ABOUT ASSEMBLY LANGUAGE DOCUMENTATION

Documentation of programs written in assembly languages is always a difficult chore. Too detailed commentary can be confusing, while insufficient detail can lead to disastrous misunderstandings of the function of the code. A valuable practice (where it can be done) is to place the FORTRAN equivalent to a block of assembly language code in the comment field to the right of the code on the card.

1.4 TO FLOWCHART OR NOT TO FLOWCHART

Some people have a mania about flowcharting, I do not. A proper job of commenting a program usually obviates the need for a flowchart. For quick-and-dirty jobs, a flowchart is never necessary, otherwise the job wouldn't fall in that class but rather into the category of slow, well-considered development jobs. Proper modularization of programming tasks, at least for FORTRAN programs, will usually replace the function of the flowchart. More will be said on this in the section on design. Most applications programmers I know draw a flowchart about every three years, when they are faced with an assignment whose logical complexity precludes handling all the variables involved within their head. Probably a good rule of thumb is

> if it didn't require a flowchart to write it, it doesn't need one to document it.

As every good rule has exceptions, this one has two. First it is quite difficult to document assembly language systems without flowcharting. Second, sometimes a computer program which has been developed by the seat-of-pants technique becomes so unwieldy, as more options are added, as to require flowcharting for the programmer to keep track of what he is doing. *

---

* An alternative to this is to utilize a 'cleanup' program such as TIDY [7].

1.5 CODING CLEAN AND CODING DIRTY

The following lines of code appear in [1],

```
      IF (     (I .LT. 1)
     1    .OR. (I .GT. N)
     2    .OR. (J .LT. 1)
     3    .OR. (J .GT. NPLUS1)
     4    .OR. (ABS(TEMP) .GT. BIGGST) ) OK = .FALSE.
```

Most amateur programmers would have coded this as follows

```
      IF((I.LT.1).OR.(I.GT.N).OR.J.LT.1).OR.J.GT.NPLUS1).OR.(ABS(TEMP)
     1.GT.BIGGST))OK=.FALSE.
```

Often as an applications consultant I have been asked by users to follow my way through statements like

```
      FI=COF*SN*ETA(I,J)-ZSQ**2*SN*(ETA(I+1,J)+ETA(I-1,J))/DZSQ-ETA(I,
     1J+1)*(COF1+COF2)-ETA(I,J-1)*(COF1-COF2)+SN*SLA*(NI(I,J)-NE)
```

instead of a much more distinct

```
      FI=             COF*SN*ETA(I,J)
     1                -ZSQ**2*SN*(ETA(I+1,J)+ETA(I-1,J)) / DZSQ
     2                -ETA(I,J+1)*(COF1+COF2)
     3                -ETA(I,J-1)*(COF1-COF2)
     4                +SN*SLA*(NI(I,J)-NE)
```

I'm sure you get the point, clean code is an important part of the documentation process. This subject has been covered at length in other places, so it won't be repeated here. We will only list most of the important aspects of coding cleanly in FORTRAN.

variable definitions

1. Intelligent variable mnemonics (SIGMA not V125)

arithmetic statements

2. Start all right-hand side expressions in the same column (I like 25)

3. Place all equal signs in the same column (optional)

4. Start scalar left hand quantities in the same column (subscripts)

statement numbers

5. Assign statement numbers in ascending order

6. Increment statement numbers by 10 or 20 while in the early stages of writing the program

7. Right-justify all statement numbers

8. Don't start a statement number in column 1

format statements

9. Place format statements at the end of the program. (This is a subject of some controversy. I find it more useful to locate the format at the position of most frequent use within the program.)

10. Assign blocks of statement numbers for format statement numbers (i.e. 1000-1999 for input formats, 2000-2999 for output formats)

miscellaneous

11. Indent do loops

12. Parenthesize fully

13. Favor the easier to understand code over the efficient or elegant in almost all cases.

1.6 GOOD DESIGN LEADS TO GOOD DOCUMENTATION

There is not too much to say in expansion on the above phrase. Proper design -- structuring programs and systems into comprehensible functional modules -- and straightforward implementation techniques can immensely ease the documentation task. I am fond of a phrase of Dijkstra [8] to the effect that good programming consists of recognizing 'how to avoid unmanageable complexity.' Proper design leads to manageable complexity, which in turn avoids unmanageably complex documentation.

A FEW WORDS ON STRUCTURED PROGRAMMING

The important aspect, from the viewpoint of documentation, of current trends toward structured programming is that control structures are being incorporated at the language level which facilitate understanding of computer program code. In the past, a great deal of documentation has been concerned with clarifying code which under the newer control structures no longer needs to be documented.

1.7 THE GUIDE TO THE EMPIRE

There comes a time in the life of every good programmer when he finds that he has more computer programs under his control than he can remember. He has reached his intellectual saturation point. At this point he can either quit and start life anew as a real estate salesman, or he can write the guide to the empire.

The guide is a detailed reference catalogue of all computer programs under his control. It should include, the location of all source and object decks of all programs, together with listings of the latest control card sequences for running the program. For an applications programmer, the organization of the guide might be along the lines of major user's. For a systems man, the organization might best be along the lines of major program areas. Each area should include a brief description of the nature, purpose, and use of each program or system. If different versions of the same system are under development, the major differences between them must be explained. The guide should also include tables of all computer tapes and other permanent storage areas (such as permanent disk files, data cell space, etc.) under his control.

The guide should be retained in some easily modifiable form so that changes may be made as more programming is done. At the Lawrence Berkeley Laboratory, we are fortunate in having two utility programs which can handle all the requirements for developing writeups dynamically, the BARB formatting program, and the UPDATE card editor. BARB is an automatic editing program which operates on text entered on card images. Thus to change a BARB writeup, one merely has to be able to insert and delete cards, which is the facility provided by Control Data Corporation's UPDATE utility program. The salient features of BARB are automatic indexing and table of contents generation.

With the guide to his programming empire finished, the programmer can safely move from project to project with the minimal amount of disruption and inefficiency. He can now drop a programming system for several months to work on other programs and then return and pick up his work almost where he dropped it, since all the relevant information, as well as pertinent memory jogs are in writing.

1.8 IN CONCLUSION LET ME SAY

Various reasons have been advanced as to why computer programmers don't document their work. Job security is one. This may have been true in the early days of programming, but today development projects usually require the efforts of more than one man, and the person who doesn't explain his work in satisfactory detail is going to find his position increasingly precarious. To those programmers who feel lack of documentation makes them indispensable, Weinberg [9] has the following suggesion for their managers, 'if a programmer is indispensable, get rid of him as quickly as possible.'

My own feeling is that good documentation requires all the effort and precision that goes into any good technical writing, indeed it calls for a precision rarely needed in programming itself. This is not to say that skill cannot be gained with experience. As with all writing skills, capacity to document to a professional level develops with experience. The lucky programmer is the one who starts out in an environment where good documentation is encouraged, or even required. As with not smoking, a good habit begun in youth saves the trauma that develops when a bad habit must finally be broken.

ACKNOWLEDGMENTS

This summary is a synthesis of over a decade of computer programming. Many of the thoughts are not original, and are the result of valuable discussions with Dave Jensen, Bill Hogan, Marilyn Mantei, and Manny Clinnick, to name a few. I owe a special debt to Tom Cundiff of the Princeton Computation Center who first made me aware of the importance of readable output and adequate documentation when we worked together at Bell Laboratories ten years ago.

1.9 REFERENCES

1. D. McCracken and G. Weinberg, How to Write a Readable FORTRAN Program, Datamation, October 1972.

2. G. Perry and J. Sommerfeld, FORTRAN Programming Aids, Software Age, Oct.-Nov., 1970.

3. Richard MacNeal, ed., The NASTRAN Theoretical Manual, Scientific and Technical Information Division, NASA, Wash, D. C., 1970.

4. C. W. McCormick, ed., The NASTRAN User's Manual, Scientific and Technical Information Division, NASA, Wash, D. C. 1970.

5. The NASTRAN Demonstration Problem Manual, Scientific and Technical Information Division, NASA, Wash, D. C. 1970.

6. The NASTRAN Programmer's Manual, Scientific and Technical Information Division, NASA, Wash, D. C. 1972.

7. Harry M. Murphy, jr., TIDY, a Computer Code for Renumbering and Editing FORTRAN Source Programs, Technical Report no. AFWL-TR-66-93, Air Force Weapons Laboratory, Kirtland Air Force Base, New Mexico, October, 1966.

8. Edsger W. Dijkstra, Craftsman or Scientist?, Proceedings of ACM Pacific 75, San Francisco, California, April 17-18, 1975, p. 222.

9. G. M. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold Company, New York, N. Y., 1971.