N73-19209

THE INTEGRATION OF SYSTEM SPECIFICATIONS AND PROGRAM CODING

William R. Luebke
Computer Sciences Corp.

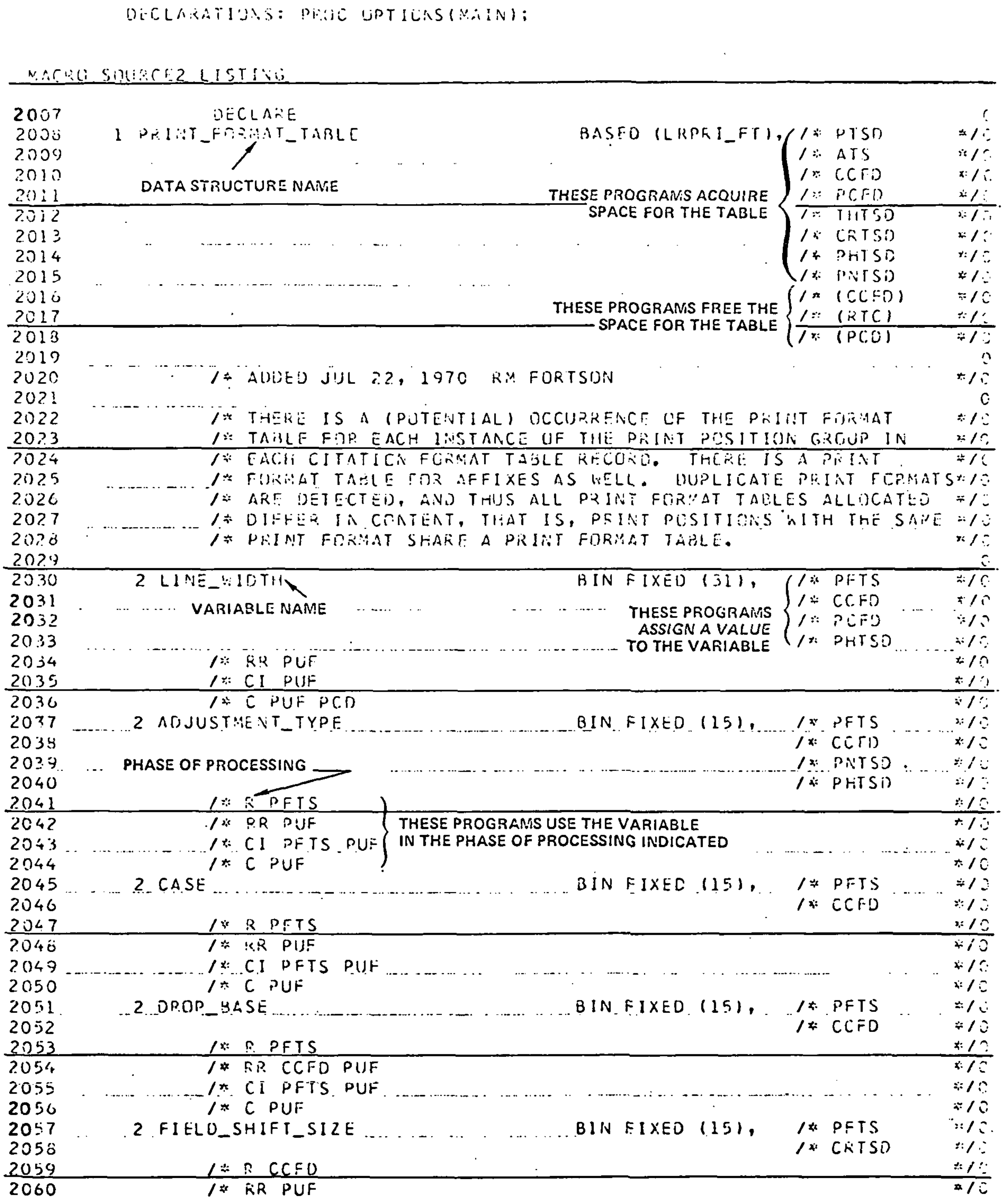
This report is a description of experience in maintaining up-to-date documentation for one module of the large-scale Medical Literature Analysis and Retrieval System II (MEDLARS II). Several innovative techniques have been explored in the development of this system's data management environment, particularly those that use PL/I as an automatic documenter. The PL/I data description section can provide automatic documentation by means of a master description of data elements that has long and highly meaningful mnemonic names and a formalized technique for the production of descriptive commentary. The techniques to be discussed are not common or unusual but are, instead, practical methods that employ the computer during system development in a manner that assists system implementation, provides interim documentation for customer review, and satisfies some of the deliverable documentation requirements.

DOCUMENTATION THROUGH DATA DEFINITION

MEDLARS II is a very complex system involving more than 50 PL/I programs that must share approximately 500 separate data variables. Most of the programmers assigned to the implementation phase had only limited experience in PL/I programming, and only a few had participated in the design of the system. The delivery schedule required that coding begin before the entire design had been completed and thoroughly reviewed. Therefore, an efficient mechanism for communicating the original design and any modifications of it to the programmers was essential.

With the number of programmers involved, each having responsibility for and knowledge of only a segment of the system, it was necessary to have ways to guarantee consistency in data definition and usage. The PL/I language has features that support these objectives. The data declaration statement in PL/I was chosen as the basis for design documentation to achieve data consistency, minimize coding and keypunching, and establish a basis for computer-produced portions of the final system documentation.

The DECLARE statement is a compiler instruction that defines the attributes and relationships of data variables. The data in the system are arranged in over 30 separate data structures or arrays of structures, each defined by a DECLARE statement. Figure 1 shows a portion of one of these structures, PRINT_FORMAT_TABLE. Line 1 names the structure and gives its attributes. The lines beginning with the number 2 name the data variables in the structure and establish their individual attributes. Comments in the PL/I language are



```
DECLARATIONS: PROC OPTIONS(MAIN);

 MACRO SOURCE2 LISTING

                DECLARE                                                      C
       1 PRINT_FORMAT_TABLE                  BASED (LRPRI_FT),/* PTSD        */C
                                                              /* ATS         */C
                                                              /* CCFD        */C
                                                              /* PCFD        */C
                                                              /* THTSD       */C
                                                              /* CRTSD       */C
                                                              /* PHTSD       */C
                                                              /* PNTSD       */C
                                                              /* (CCFD)      */C
                                                              /* (RTC)       */C
                                                              /* (PCD)       */C
                                                                             C
                /* ADDED JUL 22, 1970  RM FORTSON                            */C
                                                                             C
                /* THERE IS A (POTENTIAL) OCCURRENCE OF THE PRINT FORMAT     */C
                /* TABLE FOR EACH INSTANCE OF THE PRINT POSITION GROUP IN    */C
                /* EACH CITATION FORMAT TABLE RECORD.  THERE IS A PRINT      */C
                /* FORMAT TABLE FOR AFFIXES AS WELL.  DUPLICATE PRINT FORMATS*/C
                /* ARE DETECTED, AND THUS ALL PRINT FORMAT TABLES ALLOCATED  */C
                /* DIFFER IN CONTENT, THAT IS, PRINT POSITIONS WITH THE SAME */C
                /* PRINT FORMAT SHARE A PRINT FORMAT TABLE.                  */C
                                                                             C
         2 LINE_WIDTH                        BIN FIXED (31),  /* PFTS        */C
                                                              /* CCFD        */C
                                                              /* PCFD        */C
                                                              /* PHTSD       */C
                /* RR PUF                                                    */C
                /* CI PUF                                                    */C
                /* C PUF PCD                                                 */C
         2 ADJUSTMENT_TYPE                   BIN FIXED (15),  /* PFTS        */C
                                                              /* CCFD        */C
                                                              /* PNTSD       */C
                                                              /* PHTSD       */C
                /* R PFTS                                                    */C
                /* RR PUF                                                    */C
                /* CI PFTS PUF                                               */C
                /* C PUF                                                     */C
         2 CASE                              BIN FIXED (15),  /* PFTS        */C
                                                              /* CCFD        */C
                /* R PFTS                                                    */C
                /* RR PUF                                                    */C
                /* CI PFTS PUF                                               */C
                /* C PUF                                                     */C
         2 DROP_BASE                         BIN FIXED (15),  /* PFTS        */C
                                                              /* CCFD        */C
                /* R PFTS                                                    */C
                /* RR CCFD PUF                                               */C
                /* CI PFTS PUF                                               */C
                /* C PUF                                                     */C
         2 FIELD_SHIFT_SIZE                  BIN FIXED (15),  /* PFTS        */C
                                                              /* CRTSD       */C
                /* R CCFD                                                    */C
                /* RR PUF                                                    */C
```

DATA STRUCTURE NAME
THESE PROGRAMS ACQUIRE SPACE FOR THE TABLE
THESE PROGRAMS FREE THE SPACE FOR THE TABLE
VARIABLE NAME
THESE PROGRAMS ASSIGN A VALUE TO THE VARIABLE
PHASE OF PROCESSING
THESE PROGRAMS USE THE VARIABLE IN THE PHASE OF PROCESSING INDICATED

Figure 1.–Sample data declaration.

delimited by /* */. Notice that most of the DECLARE statement is comprised of comments. It is through the comments that the system's programs are related to the data.

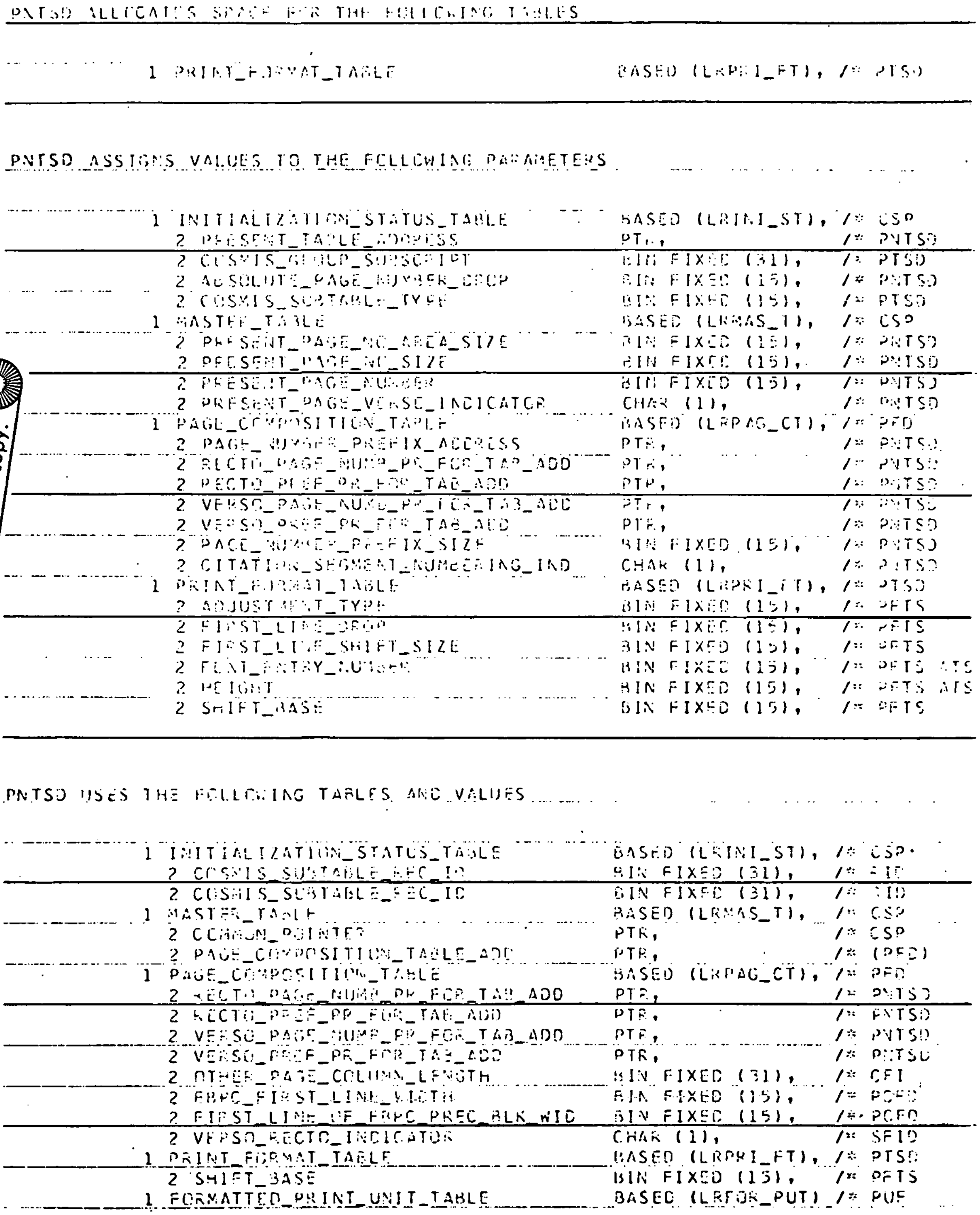
The comments following the first line on the right side of the figure identify the programs that acquire storage or free storage (the latter indicated by parentheses) for the data structure. Below this is a comment paragraph that discusses the occurrence of the table within the system. Each data element in the structure is named on a line beginning with the number 2. Notice that long descriptive names are used in the declaration. PL/I permits up to 31 characters. At the right edge of a data-element line are comments that identify which programs set a value for that data element. For example, PFTS assigns a value to LINE_ WIDTH. Below the data-element name are comment lines that identify the programs which use the data element's value. In the first case, the RR signifies one of the four phases of processing. In the RR phase, one routine (PUF) uses the element. In the C phase of processing, two routines (PUF and PCD) use the data element. The same type of information appears for all the data elements in all data structures.

Each programmer has a book of all the data structures used by the system. The book is produced by the PL/I compiler, and, in addition to the DECLARE statements, it contains an alphabetical listing of all data elements in the system. This listing, a normal compiler product, identifies the attributes of the element and the structure in which it appears.

The comments in the declaration are rigidly defined by position to permit simple manipulation of these data by a computer program. At present, a PL/I program processes all the declarations as input data and produces a listing of data involvement for each program. Figure 2 is an example of this program's output. The designer is now able to express any additions or modifications to the system design through changes in the declarations. These changes are quickly communicated to the programmer by means of the program data involvement sheets. Both the DECLARE statements and the program data involvement sheets will be part of the final documentation of the system.

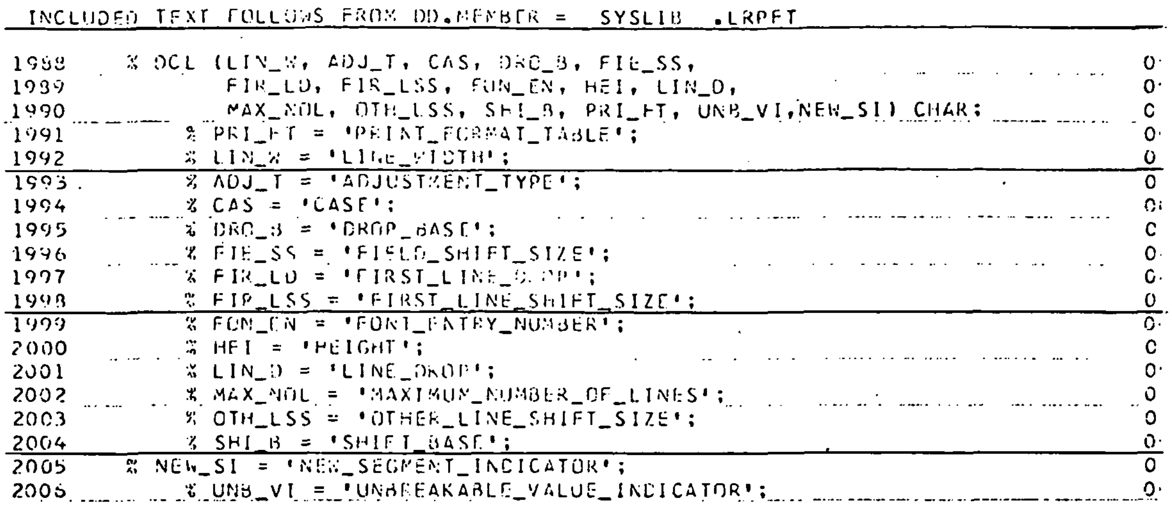
There is another feature of the PL/I compiler that has been a very valuable aid in the use of the declarations. Before program compilation, a preprocessor scan is made of the compiler input, the source deck. The preprocessor phase permits inclusion of data from libraries in the system and certain procedural operations to take place before compilation. Presently, the declarations are being cataloged into the source library under a member name algorithmically derived from the name of the data structure. The preprocessor system is assigned two codes (LR and LP) to identify it within the system. LR is interpreted as data; LP, as procedure. The declaration statements are cataloged in the library with an LR prefix, followed by the first letter of each word in the name of the table. LRPFT is the library member name of the PRINT_FORMAT_TABLE. When programs are cataloged, their names are preceded by LP. For example, on the first line of the example declaration (fig. 2), it is indicated that the core area for the table was acquired by a routine named PTSD. In the library and in coding, the actual name of the routine is LPPTSD.

To gain access to the material in the source library, the programmer writes a %INCLUDE statement in his code, naming the member name of the table (e.g., %INCLUDE LRPFT). The member name is derived from the name of the table in the documentation. This statement will automatically cause the acquisition of both that library member and the declaration statement itself (from the source library) and will cause the declaration statement to be



```
PNTSD ALLOCATES SPACE FOR THE FOLLOWING TABLES

          1 PRINT_FORMAT_TABLE                        BASED (LRPRI_FT), /* PTSD


PNTSD ASSIGNS VALUES TO THE FOLLOWING PARAMETERS

          1 INITIALIZATION_STATUS_TABLE               BASED (LRINI_ST), /* CSP
            2 PRESENT_TABLE_ADDRESS                   PTR,              /* PNTSD
            2 COSMIS_GROUP_SUBSCRIPT                  BIN FIXED (31),   /* PTSD
            2 ABSOLUTE_PAGE_NUMBER_CROP               BIN FIXED (15),   /* PNTSD
            2 COSMIS_SUBTABLE_TYPE                    BIN FIXED (15),   /* PTSD
          1 MASTER_TABLE                              BASED (LRMAS_T),  /* CSP
            2 PRESENT_PAGE_NO_AREA_SIZE               BIN FIXED (15),   /* PNTSD
            2 PRESENT_PAGE_NO_SIZE                    BIN FIXED (15),   /* PNTSD
            2 PRESENT_PAGE_NUMBER                     BIN FIXED (15),   /* PNTSD
            2 PRESENT_PAGE_VERSO_INDICATOR            CHAR (1),         /* PNTSD
          1 PAGE_COMPOSITION_TABLE                    BASED (LRPAG_CT), /* PFD
            2 PAGE_NUMBER_PREFIX_ADDRESS              PTR,              /* PNTSD
            2 RECTO_PAGE_NUMB_PR_FOR_TAB_ADD          PTR,              /* PNTSD
            2 RECTO_PREF_PR_FOR_TAB_ADD               PTR,              /* PNTSD
            2 VERSO_PAGE_NUMB_PR_FOR_TAB_ADD          PTR,              /* PNTSD
            2 VERSO_PREF_PR_FOR_TAB_ADD               PTR,              /* PNTSD
            2 PAGE_NUMBER_PREFIX_SIZE                 BIN FIXED (15),   /* PNTSD
            2 CITATION_SEGMENT_NUMBERING_IND          CHAR (1),         /* PNTSD
          1 PRINT_FORMAT_TABLE                        BASED (LRPRI_FT), /* PTSD
            2 ADJUSTMENT_TYPE                         BIN FIXED (15),   /* PFTS
            2 FIRST_LINE_DROP                         BIN FIXED (15),   /* PFTS
            2 FIRST_LINE_SHIFT_SIZE                   BIN FIXED (15),   /* PFTS
            2 FONT_ENTRY_NUMBER                       BIN FIXED (15),   /* PFTS ATS
            2 HEIGHT                                  BIN FIXED (15),   /* PFTS ATS
            2 SHIFT_BASE                              BIN FIXED (15),   /* PFTS


PNTSD USES THE FOLLOWING TABLES AND VALUES

          1 INITIALIZATION_STATUS_TABLE               BASED (LRINI_ST), /* CSP
            2 COSMIS_SUBTABLE_REC_ID                  BIN FIXED (31),   /* FID
            2 COSMIS_SUBTABLE_REC_ID                  BIN FIXED (31),   /* CID
          1 MASTER_TABLE                              BASED (LRMAS_T),  /* CSP
            2 COMMON_POINTER                          PTR,              /* CSP
            2 PAGE_COMPOSITION_TABLE_ADD              PTR,              /* (PFD)
          1 PAGE_COMPOSITION_TABLE                    BASED (LRPAG_CT), /* PFD
            2 RECTO_PAGE_NUMB_PR_FOR_TAB_ADD          PTR,              /* PNTSD
            2 RECTO_PREF_PR_FOR_TAB_ADD               PTR,              /* PNTSD
            2 VERSO_PAGE_NUMB_PR_FOR_TAB_ADD          PTR,              /* PNTSD
            2 VERSO_PREF_PR_FOR_TAB_ADD               PTR,              /* PNTSD
            2 OTHER_PAGE_COLUMN_LENGTH                BIN FIXED (31),   /* CFI
            2 FBPC_FIRST_LINE_WIDTH                   BIN FIXED (15),   /* PCFD
            2 FIRST_LINE_OF_FBPC_PREC_BLK_WID         BIN FIXED (15),   /* PCFD
            2 VERSO_RECTO_INDICATOR                   CHAR (1),         /* SFID
          1 PRINT_FORMAT_TABLE                        BASED (LRPRI_FT), /* PTSD
            2 SHIFT_BASE                              BIN FIXED (15),   /* PFTS
          1 FORMATTED_PRINT_UNIT_TABLE                BASED (LRFOR_PUT) /* PUF
```

Figure 2.–Program data involvement.



```
INCLUDED TEXT FOLLOWS FROM DD.MEMBER =  SYSLIB  .LRPFT

1988    % DCL (LIN_W, ADJ_T, CAS, DRO_B, FIE_SS,                          0
1989            FIR_LD, FIR_LSS, FUN_EN, HEI, LIN_D,                      0
1990            MAX_NOL, OTH_LSS, SHI_B, PRI_FT, UNB_VI,NEW_SI) CHAR;     C
1991          % PRI_FT = 'PRINT_FORMAT_TABLE';                             0
1992          % LIN_W = 'LINE_WIDTH';                                      0
1993          % ADJ_T = 'ADJUSTMENT_TYPE';                                 0
1994          % CAS = 'CASE';                                              0
1995          % DRO_B = 'DROP_BASE';                                       C
1996          % FIE_SS = 'FIELD_SHIFT_SIZE';                               0
1997          % FIR_LD = 'FIRST_LINE_DROP';                                C
1998          % FIP_LSS = 'FIRST_LINE_SHIFT_SIZE';                         0
1999          % FUN_EN = 'FONT_ENTRY_NUMBER';                              0
2000          % HEI = 'HEIGHT';                                            C
2001          % LIN_D = 'LINE_DROP';                                       0
2002          % MAX_NOL = 'MAXIMUM_NUMBER_OF_LINES';                       0
2003          % OTH_LSS = 'OTHER_LINE_SHIFT_SIZE';                         0
2004          % SHI_B = 'SHIFT_BASE';                                      0
2005    % NEW_SI = 'NEW_SEGMENT_INDICATOR';                                0
2006          % UNB_VI = 'UNBREAKABLE_VALUE_INDICATOR';                    0
```

Figure 3.—Preprocessor statements.

physically inserted into the source code prior to compilation. All data declarations exist in only one place, the source library. All programmers use the same declaration from the library, and, therefore, changes in the data declaration will be automatically and immediately available to all programmers because the latest version appears in their program listing. In this way, data names and table identification are consistent.

Using the full 31 characters for naming a variable is awkward for a programmer and can lead to keypunching problems. The PL/I compiler helps here as well. In the preprocessor pass, before the actual compilation, it is possible to replace items in the source code. This feature is used to convert the programmer's abbreviation of a data variable name to the full name. A standard algorithm is used for abbreviating a name in a table: The first three characters of the first word, an underline character, and then the first character of each succeeding word. For example, the programmer writes line width as LIN_W, adjustment type as ADJ_T, and case as CAS. When he receives the computer listing, all the abbreviated names will have been replaced, and his listing will include the full descriptive name that was used in the declaration of the tables. The preprocessor code required to accomplish the conversion from abbreviated name to full name has been included as an addition to the member in the source library that contains the declaration of the table (fig. 3). Therefore, when the programmer includes the table declaration in his program, he is also including the preprocessor code that will accomplish all the abbreviation conversions for the data elements in the table. As a result, the preprocessor code is only prepared once, and the conversion is automatically performed any time the table is used. This does not mean, however, that the programmer cannot use the full version of the name. Either version can be used in this code.

IMPLEMENTATION

One man-week was required to code and catalog the declarations for the system library. This figure does not include the design of the structures or the keypunch time. Currently, table maintenance requires about 1/2 man-week for each calendar week. When system implementation began, 15 man-weeks were expended in coding and testing utility routines that output the data structures during program debugging.[1] This coding exercise was useful for training non-PL/I programmers and as an introduction to the rather complex data hierarchy.

PROBLEMS ENCOUNTERED

The basic concepts set forth in this paper have been validated by experience, and programmers find the structures easy to use. Design inconsistencies and program specification shortcomings appear to surface early in implementation, as soon as the program data involvement is available. However, if the reader intends to use this approach he should be aware of the following potential problems:

(1) The name abbreviations in the preprocessor statements must be unique. If they are not, the program using the declaration will not pass the preprocessor phase of compilation. This problem has caused several days of table lockout in our implementation.

(2) The printout material used by programmers must be updated with every non-comment change to the declarations. Failure to do this will cause the programmer to think that he has an error when his program is actually correct.

(3) A change in a data structure requires that all programs using that structure be recompiled.

(4) Program testing must be suspended while the data structures and related print programs are being updated in the library.

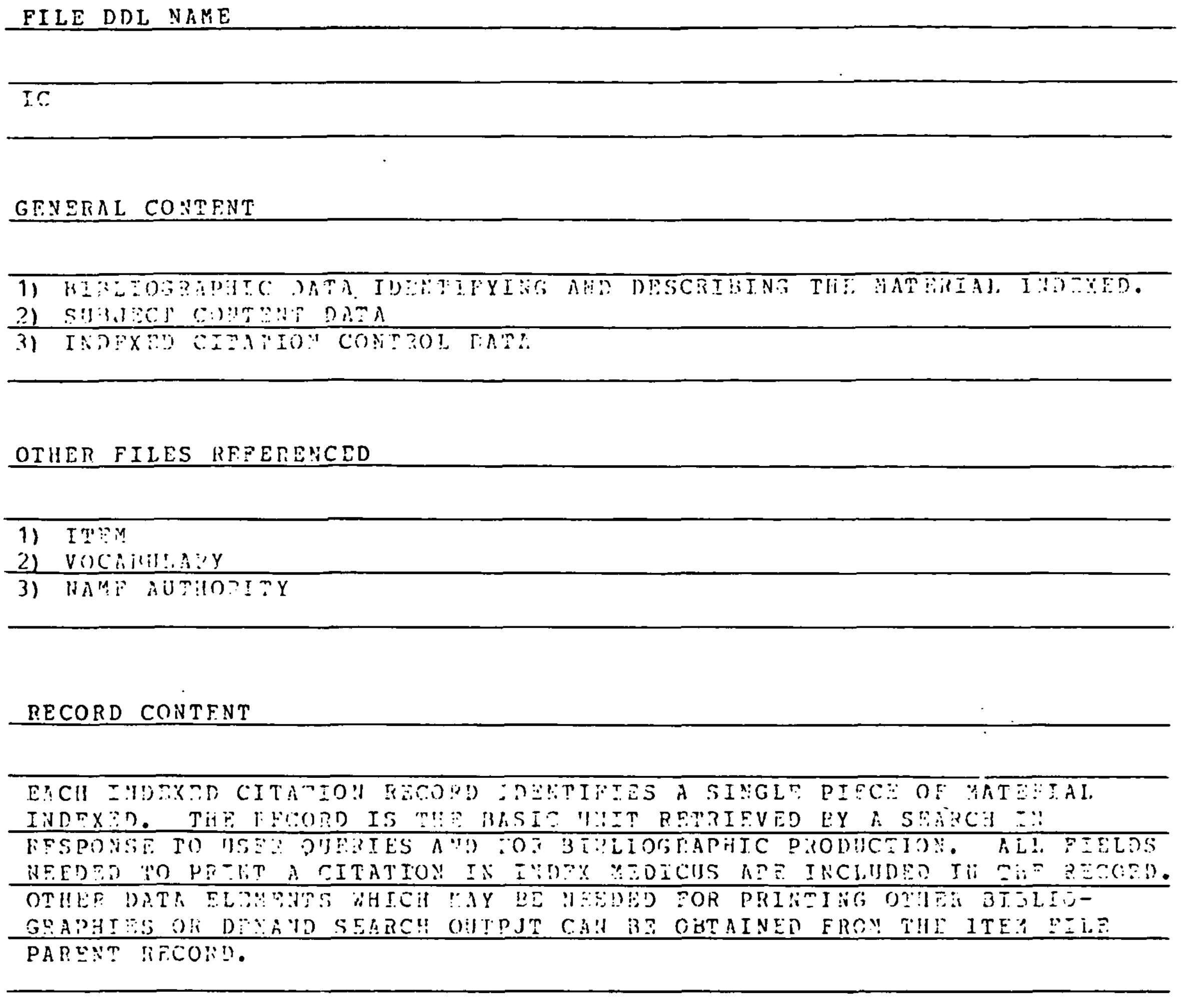
We found it best to update the tables no more frequently than once a month, to use extreme care, and to anticipate the sacrifice of at least one computer run by every programmer.

DATA FILE DOCUMENTATION

The remainder of this report briefly describes the manner in which user-oriented data files are documented. Approximately 30 individual files make up the MEDLARS II data base. A consistent method is needed for the definition of information carried in these files, a method that could be understood by non-data-processing personnel in the user's facility who are interested in the content, but not the structure, of the file and need a great deal of information about the files.

Nine descriptors were devised for every data element in a file, and these files were documented in machine-readable form so that they would be easy to update while the client thought about problems and requested changes. The files have been evolving, and the documentation has been able to keep abreast. Generally, it takes only a few days to document very sweeping changes in the design of the National Library of Medicine's bibliographic files.

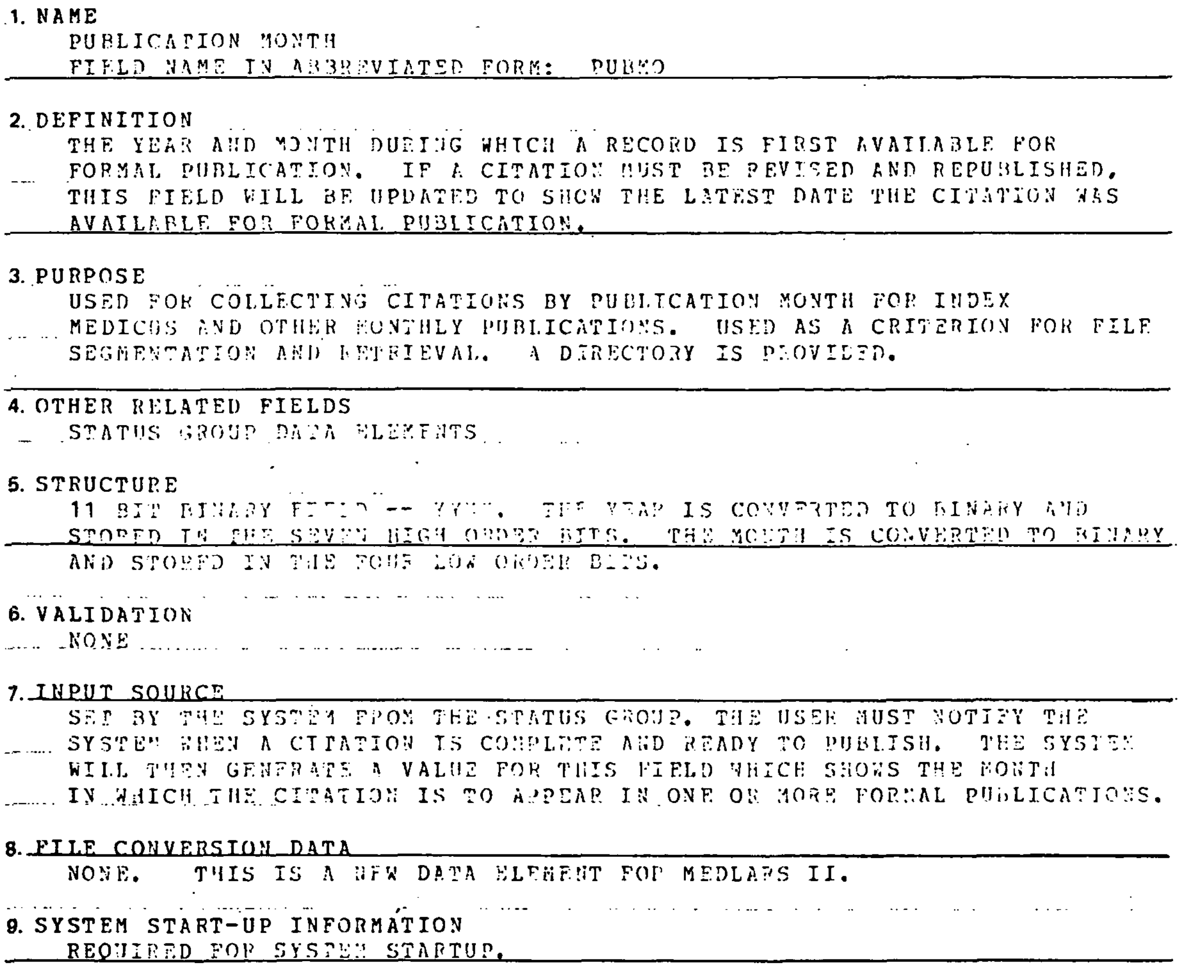
[1] The structures are based on, and therefore cannot be output by, the PUT DATA instruction nor can the variables be traced with the CHECK function.



```
FILE DDL NAME

IC

GENERAL CONTENT

1)  BIBLIOGRAPHIC DATA IDENTIFYING AND DESCRIBING THE MATERIAL INDEXED.
2)  SUBJECT CONTENT DATA
3)  INDEXED CITATION CONTROL DATA

OTHER FILES REFERENCED

1)  ITEM
2)  VOCABULARY
3)  NAME AUTHORITY

RECORD CONTENT

EACH INDEXED CITATION RECORD IDENTIFIES A SINGLE PIECE OF MATERIAL
INDEXED.  THE RECORD IS THE BASIC UNIT RETRIEVED BY A SEARCH IN
RESPONSE TO USER QUERIES AND FOR BIBLIOGRAPHIC PRODUCTION.  ALL FIELDS
NEEDED TO PRINT A CITATION IN INDEX MEDICUS ARE INCLUDED IN THE RECORD.
OTHER DATA ELEMENTS WHICH MAY BE NEEDED FOR PRINTING OTHER BIBLIO-
GRAPHIES OR DEMAND SEARCH OUTPUT CAN BE OBTAINED FROM THE ITEM FILE
PARENT RECORD.
```

Figure 4.–Overview of indexed citation file.

Figure 4 shows the overview of the indexed citation file. The overview describes the general content of the file, identifies other files in the system that are referenced by this file in some way, and then generally describes the record content. The record content is just an overview that is used for a quick introduction to the file. It is followed by a description of the data within the file. Not shown is a pictorial representation of the file structure identifying all the data elements in the file. Each data element in a record is documented by punched cards that are numbered to identify the type of information carried. Figure 5 describes the numbered data-element listing. There is also a type of card for comments. Several small PL/I programs for formatting and editing the data have been written. A great deal of supporting keypunch work is required for implementation of the file, but once the file has been established, the data are easy to update.



```
1. NAME
    PUBLICATION MONTH
    FIELD NAME IN ABBREVIATED FORM:  PUBMO

2. DEFINITION
    THE YEAR AND MONTH DURING WHICH A RECORD IS FIRST AVAILABLE FOR
    FORMAL PUBLICATION.  IF A CITATION MUST BE REVISED AND REPUBLISHED,
    THIS FIELD WILL BE UPDATED TO SHOW THE LATEST DATE THE CITATION WAS
    AVAILABLE FOR FORMAL PUBLICATION.

3. PURPOSE
    USED FOR COLLECTING CITATIONS BY PUBLICATION MONTH FOR INDEX
    MEDICUS AND OTHER MONTHLY PUBLICATIONS.  USED AS A CRITERION FOR FILE
    SEGMENTATION AND RETRIEVAL.  A DIRECTORY IS PROVIDED.

4. OTHER RELATED FIELDS
    STATUS GROUP DATA ELEMENTS

5. STRUCTURE
    11 BIT BINARY FIELD -- YYMM.  THE YEAR IS CONVERTED TO BINARY AND
    STORED IN THE SEVEN HIGH ORDER BITS.  THE MONTH IS CONVERTED TO BINARY
    AND STORED IN THE FOUR LOW ORDER BITS.

6. VALIDATION
    NONE

7. INPUT SOURCE
    SET BY THE SYSTEM FROM THE STATUS GROUP. THE USER MUST NOTIFY THE
    SYSTEM WHEN A CITATION IS COMPLETE AND READY TO PUBLISH.  THE SYSTEM
    WILL THEN GENERATE A VALUE FOR THIS FIELD WHICH SHOWS THE MONTH
    IN WHICH THE CITATION IS TO APPEAR IN ONE OR MORE FORMAL PUBLICATIONS.

8. FILE CONVERSION DATA
    NONE.   THIS IS A NEW DATA ELEMENT FOR MEDLARS II.

9. SYSTEM START-UP INFORMATION
    REQUIRED FOR SYSTEM STARTUP.
```

Figure 5.–Data-element listing.

DISCUSSION

MEMBER OF THE AUDIENCE: Do you feel that it is a worthwhile effort to attempt to document large programs that are continually changing?

LUEBKE: I think that it is a good idea and should be helpful, as part of the source documentation, when the system is finally operational. Having the data within the system catalog will permit programmers who did not participate in the development of the system to identify the relationships between the programs and the data so that they can perform maintenance.

MEMBER OF THE AUDIENCE: How many programmers are involved in your project?

LUEBKE: At the present time, we have about 27 programmers working in this area.

MEMBER OF THE AUDIENCE: For how many years has your project been in existence?

LUEBKE: We began programming in August 1967 and should be finished with that phase in the spring of 1971.

PANEL DISCUSSION

MEMBER OF THE AUDIENCE: I feel one of the most pertinent things that Dr. Swift said was that using automatic techniques to describe data is one of the most feasible and valuable things that can be done. In this last paper, there seems to be a suggestion of how to do that. Does the panel agree or disagree, and what are the relative merits of this approach?

PANEL MEMBER: I agree fully that the independent description of data is something that can now be done.

PANEL MEMBER: Not only description of data but also typing the description of the data to the system itself so that the data description entering into the system are basically the same ones the programmer works from.

PANEL MEMBER: I would like to add that besides describing the data, there is a problem of analyzing it. It becomes important to know where the data are throughout the programs and subprograms and how the data interact with each other after being changed.

PANEL MEMBER: My feeling is that as serious a problem as the traditional after-the-fact documentation is, the most serious part of the problem comes at the beginning of a project, what I call the "upstream documentation." This is the attempt to record information that everybody can understand, work with, and write code from. A considerable amount of documentation is brought into existence by going through the various stages of system design, from the requirements at the start of the project to the various elements and levels of the design approach to, finally, the stage that can be solved by writing code rather than by developing a further level of design.

I think the data processing business has perhaps been somewhat deficient in not putting enough emphasis on the total process of supporting and facilitating development of this upstream documentation. If properly done, it should form a body of material that can be coupled to the kinds of tools that are capable of being developed now for automatic documentation. This upstream documentation, rather than the problem of building from something that is already computer processable, is the basic problem.

PANEL MEMBER: I agree. Until program documentation flows naturally out of the designing process, it is always going to be a problem. You have to start at the beginning and generate most of the documentation in the process of designing a system. Until proper procedures are followed, good documentation is never going to happen.

PANEL MEMBER: Let me add that managers must also face the fact that certain costs are going to be incurred and that their project reports are going to reflect cost increases before any code will be written.

PANEL MEMBER: That comment compares the cost of getting the deck as it comes out of the computer without any comments to the cost of getting it with BELLFLOW. I think the manager should compare what it costs to put BELLFLOW comments in against what it costs to put in the comments that should be put in the deck anyway. Often if you insist on doing the job correctly in the first place, some of these other additions do not add much of a burden.

MEMBER OF THE AUDIENCE: The comment that programmers are undisciplined has been made several times. I wonder what the panel thinks about this.

PANEL MEMBER: I have a few random thoughts on the subject. One of the roots of the problem lies in programmer education. When programmers are trained, documentation very often is not stressed. No organized way to write a program is taught. Programs consequently reflect the idiosyncrasies of the programmer. But the organization of a program, given the same kind of problem, should be standard. Unfortunately, it is not. The solution to this part of the problem lies in better training.

Second, I agree that the only way to get reasonably good documentation is to have the documentation developed with the programming. For one thing, it is the only way it will get done because programmers are generally working on another project soon after the completion of one and have neither the time nor the inclination to document once the program is finished. In addition, they may forget certain aspects of the program. So, the only way to document is to integrate documentation with normal programming activity.

MEMBER OF THE AUDIENCE: I think you have to recognize that a computer program is a very difficult thing to describe in the first place. No system tells how to read documentation, there is nothing like a flowchart of how to proceed through a particular piece of documentation, which may be in narrative form.

The programmer works directly with a program and has no way of viewing it as a reader. I think one of the basic problems in program documentation is that programmers are not trained to think of how to make their programs readable to others. Most scientists, let alone most programmers, are not trained to write, and this fact must be recognized as we attempt to develop tools for automatic documentation.

Finally, the symbols that we use in flowcharts do not fit together with meaningful ways of expressing this information. I think that something ought to be done if we are going to have large names in data and procedure names that are 31 characters long. I wonder what the panel's comments on this are.

PANEL MEMBER: The situation in data processing is that the tools of documentation and description for computer programs that have been used are reasonably appropriate for basic computer program and data processing situations. But they are not really suitable for establishing and describing things like multiple-application, multiple-user, on-line, and real-time systems of various kinds. It is in many ways rather surprising that we have been able to do as well as we have in describing some of these newer situations with tools that were developed for an earlier kind of problem. Herein lies one of the main challenges for documentation.

PANEL MEMBER: I think one important question would be determining the amount of documentation that has to be done by people and the amount that can be done by computer.

PANEL MEMBER: When a building, the hardware part of a computer system, or a communication switching network is documented, something that can be seen and either agreed or disagreed with is being documented. Documenting a program is documenting an idea rather than something physical. It is more difficult to accept the fact that we have to spend money to figure out how to document and convey an idea to somebody than it would be for something physical.

You could show an executive director of a company, who has not been involved in a program, documentation of that program, and he would find it very difficult, even if it were the best documentation possible, to decide whether it is worth doing or not.

PANEL MEMBER: I think documentation should be divided into three stages. The first stage would be documentation of the planning stage. Then there would be documentation of the implementation effort. Once the program is implemented, there would be terminal documentation. Possibly, this might be the way to approach documentation. I would say that the motivation to document is certainly very strong at the beginning of a project. By looking at documentation in terms of a kind of life cycle, we certainly would get a great deal of this documentation completed at the very beginning. It may be that a lot of our thinking is simply not recorded at the time when it would be easiest.

PANEL MEMBER: I would like to comment on documentation of the implementation design effort. We are trying to record the development of a program so that if modifications are to be made to a program, there will be some idea of how complex the procedure of changing the program will be.

MEMBER OF THE AUDIENCE: One of the problems in documentation is that poor programs are often written. Very little is known about how to write programs. If you keep a program long enough, debug it long enough, and patch it long enough, it finally does everything it is supposed to do. Then, we often try to document these programs, which should not have been written in the first place.

PANEL MEMBER: I agree with you in one sense, and in another sense I do not. Very often what happens when programs are developed is that they are developed for certain specifications. The program, however, may be used for many years. If a little more effort were put into the program and a certain amount of flexibility were built into the program, then as the program changed, there would be less of a problem in adding to the program. I know it is difficult to try to foresee what changes may be needed, but I think if, for instance, the initial investment were increased by 25 percent, more than 100 percent may be saved during the life of that system.

PANEL MEMBER: Another problem is that finiteness is gradually disappearing, especially from the larger data-driven systems. There are now so many alternatives that to decide what future alternatives are is almost impossible.

PANEL MEMBER: Documenting programs that should not have been written is a problem. The programmer is not always at fault, however. Often he receives a specification that does not reflect the final product.

One company has eliminated this particular hazard by using intermediate personnel between the engineers and the programmers. They are familiar with the engineer and his field and also understand something about programming. These intermediaries read the specifications and translate them for the programmer.

As far as specifications are concerned, the person who wants the program written should know what he wants it to do. It is not enough to know what the input is and what the output should be. Good preliminary documentation is needed to write up a correct specification for a programmer. You have to have good preliminary documentation before you can give a specification to the programmer because he does what the specification tells him to and that is all he can do.

MEMBER OF THE AUDIENCE: After a program is debugged, it is assumed to operate perfectly. That is not true about hardware. In hardware, fault indicators and other safeguards are built in so that if a system does not operate properly it stops. The equipment is often capable of indicating why the machine is failing. Documentation certainly is often used to try to track down problems, but not to the extent it could be. The typical commercial application is developed so that you can check for data errors, but the program will not check to see that it itself is performing properly.

PANEL MEMBER: Many of the things we do, of course, have been done by the systems approach but because of various factors including the undependability of Government funding, we often start and then stop programming efforts. Another problem is that badly documented programs are often inherited.

MEMBER OF THE AUDIENCE: This brings up an interesting point. I think all the panel members have operated under Government contract. What do you think of the specifications laid down for programming documentation and the followup action on the part of the Government from the start of a program to the delivery of the final product?

PANEL MEMBER: I believe that all contracts should stipulate that the contractor maintain and document his program for a specified length of time after the completion of the contract.

MEMBER OF THE AUDIENCE: How do you handle the problems of having to take over a system that is already set up and documenting for others?

PANEL MEMBER: One case that I happen to know about concerned a defense-oriented system of considerable size. It became necessary for another group to take over its maintenance. In that particular instance, the company worked backwards. They began by specifying what the requirements were when the requirements specification was developed. They then proceeded to go through the first level of developing the end-item specification, called the general design specification. These appeared to be in agreement with what was going on. Then the next levels of specifications were produced and checked until, finally, all the specifications that ought to have been produced in the process of developing the program in the first place were written. Some "fudging" brought the actual code into agreement with the specifications thus created. Only at that point did it really become possible for the company to relax and begin actual maintenance.

MEMBER OF THE AUDIENCE: One of our principal customers has a requirement that many of our programs be sent to COSMIC for further distribution. In the past, we had a great deal of difficulty in getting the programs accepted. We solved that problem by setting up a review team independent of the original programming group. The review group is composed of operations, engineering, and programming people that take a program and work through the program library, program by program, to see whether it is completely understandable and can be shipped out to someone else with the assurance that it works and can be run elsewhere on the same machine.

The original programming group has a fixed budget for each program sent through the review group. If it costs more than the allotted amount to review, the originating cost center gets tabbed for the excess cost. We have not had enough experience to see how this is going to work out, but this factor of economic accountability would seem to guarantee its success.

MEMBER OF THE AUDIENCE: The talk about developing the documentation for the life cycle of a program sounds very nice; however, for some programs that is not necessary. I think automation can really be useful by keying information on data in different ways.

I think there should be a file of information about a program that can be called upon when necessary, but there is no need for reams of information that you cannot find your way through. The trouble is not having enough documentation but having so much that you cannot begin to understand how to use it or how to be able to get into it. There are times when you have to change a program in a short period of time. Then you need a certain amount of assurance that you have documentation for and know the location of all the data of a certain kind for each program. You do not need to have something printed out every time one factor in a program is changed. Maybe we should look upon the computer as being an information retrieval system of documents and documentary information about various elements of the program and its logic and not let it become a producer of printed documents.

PANEL MEMBER: I agree that we do not need to print all the necessary documentation. We find a tape recorder very useful in documentation in two ways. General descriptions and information about the program are recorded but not printed. We keep the tape for reference only. Generally, the original designer talks about where you might change the program and certain idiosyncrasies of the program. This tape gives a very good picture of how the program was developed and why and where you might have to be very careful if changes have to be made.

We also use the cassette to record basic information that a programmer should write but never gets around to doing.

MEMBER OF THE AUDIENCE: If you had clear-cut specifications in advance, then clearly all you need is somebody to code it. That is one thing. But in advanced theoretical research, you probably cannot get clear-cut specifications in advance that will lead to efficient ways of eventually writing the program. I think that we may lose some creativity in programming if we force too many specifications on programmers.

PANEL MEMBER: I would say that 90 percent of all the programming done in the United States is not creative. I would also say that there is certainly no reason why one should not have complete flexibility in the other 10 percent of the cases.

PANEL MEMBER: If you are generating a system to be used only once, you do not care much about how it is evolved. If you are generating something that is going to be used many times, then you should not be so creative that you ignore efficiency and good design.

MEMBER OF THE AUDIENCE: You mentioned a 25-percent increase in cost to make programs easier to handle. How do you decide which programs will persist and thus need additional effort to make them meet future requirements?

PANEL MEMBER: In some situations, for example, a payroll system, it is relatively easy to see how the system might have to change in the future as the company changes. In other cases, for example, the space program, how the system will change is not so obvious but the fact that it will is. So some allowance for change has to be made and is worth the extra 25 percent. But even in these obvious cases, little is being done.

PANEL MEMBER: It seems to me that almost any program of reasonable size that is being developed is worth spending that extra 25 percent on. You may be wrong, and it may have only limited life. But if you think that it is going to have long life, you ought to put in extra effort.

Session II

15