Defining and Using a Data Dictionary with Complex I/O Intensive Programs

Walter A. Lounsbery, 6-4-90

Table of Contents

	Table of Contents	1
	Abstract	2
	Preface	2
	Sources	3
1.0	Introduction	4
2.0	Data Dictionary Definitions	6
2.1	Language	6
2.2	User Interface	6
2.3	Support of Engineering Units	7
2.4	Overall Programming Scheme	7
3.0	Detailed Basic Data Dictionary Design	8
3.1	Main Program Requirements	8
3.1.1	COMMON Block Definitions	9
3.2	Program Routine Requirements	9
3.2.1	Variable Access in the Program	9
3.2.2	Programmed Variable Access for the User	10
3.3	Basic File Variable, Array, and Table Input	11
3.4	Basic Interactive Variable and Table Editing	13
4.0	Advanced Data Dictionary Design	14
4.1	Variable Length String Storage in FORTRAN	14
4.1.1	Variable Aliasing and Wildcards	15
4.1.2	Interactive Variable Allocation	16
4.2	Text Description of Variables	17
4.3	Journaling	17
4.4	Input Error Detection	18
4.5	Group References	19
4.5.1	Interactive Initialization by Reference	19
4.5.2	Interactive Output Definition	20
4.6	Program Data Tracing	21

Abstract

This report discusses desired features for data dictionaries in computer programs from the standpoint of engineering applications. A data dictionary provides the application program with information about program data variable names, types, and units. Enhancements possible with various data dictionary schemes are also introduced. Although this report does not present actual computer program code to implement a data dictionary, some skeleton examples are shown.

The data dictionary scheme is used to provide batch and interactive named input and output of variables used by the program. This also provides the foundation for a number of features. Some basic features supported by the data dictionary would be a generalized data table input, such as the AlphaZulu Format, interactive examination and editing of variables, and group referencing for initialization. Advanced features include variable aliasing and wildcards, journaling, and interactive output formatting.

Preface

This report is a product of the Aerodynamic Stability and Control Engineering Modernization (EnMod) effort for 1990. It is an effort to collect several ideas about engineering computer programs into a reasonably cohesive document. Many of these ideas will be incorporated in this year's EnMod developments.

The ideas presented here have been extensively refined in the two months since this report was started. I feel that the methods and skeleton examples presented here are still very useful, although they are somewhat primitive compared to current practice. Specific reports will be released to detail the advanced concepts and techniques of this year's data dictionary support library and development tools.

Sources

1. "Advanced Continuous Simulation Language (ACSL) Reference Manual", Mitchell and Gauthier Associates, Concord, Mass., 1986

2. "Development and Use of Numerical and Factual Data Bases", AGARD Lecture Series No. 130, 1983

3. Gonnet, G. H., "Handbook of Algorithms and Data Structures", Addison-Wesley Publishing Company, 1984

4. Hudak, Paul, "Exploring Parafunctional Programming: Separating the What from the How", IEEE Software, January 1988

5. Landingham, George M., "Aerodynamic Data Base User's Guide", U.S. Army Missile Research and Development Command, 5 June 1979

6. Radcliffe, Robert A., and Raab, Thomas J., "Data Handling Utilities in C", Sybex Inc., 1986

7. Smith, L. J., and Matthews, N. O., "Aircraft Flight Test Data Processing, A Review of the State of the Art", AGARD Flight Test Instrumentation Series, Vol. 12, 1980

1.0 Introduction

One of the most demanding and complex areas of computer programs involves engineering methods and analysis. Although computers are well suited to handling large quantities of numbers and complex computations, these are often handled in a very abstract manner. The formulas are often coded in computer languages and algorithms that bear little resemblence to the actual methods. Data is often spewed out in lines of unlabeled numbers, sometimes to binary files that are not readable without special help.

In this environment critical errors can easily occur in the transfer of data and interpretation of reams of computer analysis. It is common to speak of the computer program as 'code' even though its results may be more ciphered than the program. In fact, it may be so difficult for the engineer to review the analysis of the data that errors in the analysis method itself may go unnoticed.

Data comprehension is improved if the data can be handled by the engineer in a natural, easily understood fashion. It is important that the engineer deal with named variables with associated units instead of data codes of unknown or dubious origin. In fact, all engineers are trained that problem solving begins with identification of the relevant factors (variables) which are immediately associated with units before a single equation is written. Solving a problem any other way is an invitation to disaster.

Unfortunately, computer programs are not usually written for good data comprehension. The engineer has historically solved the problem on paper using familiar methods. The equations were handed to a programmer along with an input format and output format for the data. The engineer was not required to program the machine and deal with computer languages that don't work in unit systems, or name factors with proper mathematical symbols. The programs were treated as a 'black box' by the engineer, to be tested for correctness, but not to be altered except by the programmer.

The advent of improved computer access, interactive workstations, and increased engineer involvement in application programming has changed that situation. Now engineers often create extremely complex computer programs, after taking little or no formal training in programming. Engineers are required to modify or rewrite

programs that have no programming documents or in-line comments. It is rare to leave any of the programming to a computer programming specialist. If the responsible engineer knows the correspondence of the program code with the analysis, program input and output often looks like a collection of random numbers with occasional cryptic labels. While the programs are created faster than before, it is widely recognized that they are more error prone.

The error rate is also driven by communication between the engineer that wrote the program, and other engineers that must use it. Any computer program of significance should be of use to everyone. Yet only the engineer that wrote the program may fully understand its coded representation of the fundamental equations of the problem. At the very least the user should be able to understand the variables in the program and how they are used.

This presents a difficult problem in the current computer environment. Although there is an ongoing debate about the best computer language for engineering use, one that promotes correct analysis and communication, for the most part the engineering community is entrenched with a simple computer language called FORTRAN. FORTRAN lacks the capability of describing measurement units, and until recently, could not name a variable with more than six alphanumeric characters.

One way to improve variable definition and communication in a FORTRAN framework is to incorporate a "data dictionary" of important variables into the computer program. Just as the computer program equations are the analogue of the mathematical equations that solve the program, the data dictionary is the analogue of the factor identification that every engineer performs as the first step in solving a problem. The data dictionary is not a black box that can be attached to an arbitrary program, however. It must incorporate some simple programming techniques, which are supported by a few generic subroutines that handle common data input/output chores. The techniques are simple and add very little to the programming effort. The payoff is higher quality data analysis, better programming, and a higher degree of self-documentation in the program.

The data dictionary techniques described here provide the capability to go beyond better programming habits. Once the program has a description of the variables, it is possible to interactively inspect or edit variable contents, to halt and restart calculations, to

interactively format output files or printout, and input function tables that are dependent on any variable in the dictionary.

The last capability is a very important one. Often engineers write equations that include undetermined functions. For example, aircraft lift coefficient may depend on the lift increments due to flap deflection, which may be determined by analysis and adjusted by data from wind tunnel tests. If it is discovered in test that the lift increment due to flap is dependent on gear position, or stores configuration, those factors can be incorporated in the table and the program with a data dictionary would read the appropriate value of the table. A program without a data dictionary would require a rewrite of the program to read the new table, an expensive process that contributes to the possibility of errors in the program.

2.0 Data Dictionary Definitions

2.1 Language

This section describes a scheme for implementing a data dictionary in a FORTRAN programming environment. FORTRAN imposes rather severe limitations on construction and use of a data dictionary. Perhaps the worst limitation is the lack of memory allocation capability. It is assumed here that FORTRAN 77 or later compilers will be used, so that the character data type is available. This discussion assumes a familiarity with FORTRAN 77 and simple programming techniques.

2.2 User Interface

The addition of a data dictionary simplifies program use by the user. The user can view or alter data by name. Input files can set variable values with simple, namelist-like syntax. Functions such as tables can use any variable in the data dictionary without rewriting the program. Good user interface design requires the capability to inspect the data dictionary. Both interactive and 'batch' execution interfaces are described.

2.3 Support of Engineering Units

Engineering data involves numbers and the units associated with them. Although programs for engineering analysis almost always handle units by convention (the input or output number is *understood* to be in "proper" units) it is also true that most engineering analysis programs lack manuals that describe the convention.

Ideally, the program should perform automatic checking and conversion. All data should be input with its units, and those units checked for appropriate type (length, time, area, and so on). Equations in the program should be able to accept a wide variation in unit systems (British, MKS, and so on), performing automatic conversion to obtain the desired result. This ideal capability is largely unobtainable with any "engineering" programming language. In fact, optimum use of engineering units is a very significant computer science research topic.

Since engineering units are essential for good engineering practice, some capability should be considered a requirement. Simple unit checks will accomplish most of the intent expressed in the above ideal capabilities. While this requires storage of units associated with all program input and output, the additional overhead is well worth the error reduction and program documentation associated with the units. In fact, units should be stored for all the variables in the data dictionary even if some variables are not explicitly intended for input or output.

2.4 Overall Programming Scheme

Data dictionary support depends on a central memory allocation of variables through COMMON blocks of arrays. Single variables in the program are EQUIVALENCEd to elements in the appropriate array. Variable name and units are described in other arrays that have a one-to-one element correspondence and form the entire data dictionary (database). While the EQUIVALENCE statements are required for each variable used in a routine, only one routine and set of code is required to set up the variable names and units. A sophisticated library of support routines manages named input and output from the data dictionary arrays.

3.0 Detailed Basic Data Dictionary Design

This section covers requirements for a basic data dictionary implementation. Only the application programmer and user details are discussed; this is not a detailed description of the support routines. Information in this section contains all of the information necessary to use the support library for data dictionaries in a program.

3.1 Main Program Requirements

In this scheme all program variables that are accessed through the data dictionary are declared as arrays in various COMMON blocks. Individual variables are located in these arrays by EQUIVALENCE statements. The EQUIVALENCE statements can be entered once as program code and re-used as necessary in the subroutines. In a program dealing with any large amount of numbers, it is convenient to allocate the variable memory as several COMMON blocks. Here is a example of COMMON block allocation:

```
PARAMETER (MREALP=100,MINTP=30,MLOGP=8,CLENGTH=20,MCHARP=40
LOGICAL LOGS(MLOGP)
CHARACTER*CLENGTH CHARS(MCHARP)
COMMON /CREAL/ARNUMS(MREALP),MREALS
COMMON /CINT/INUMS(MINTP),MINTS
COMMON /CLOG/LOGS,MLOGS
COMMON /CCHAR/CHARS,MCHARS
DATA MREALS/MREALP/,MINTS/MINTP/,MLOGS/MLOGP/,MCHARS/MCHARP/
```

Notice that variables are also separated by type. FORTRAN 77 does not define COMMON blocks for character variables mixed with other types. This makes segregation of variable types a good practice.

Character variable allocation here is shown in an extremely simplified form. Normally, character variables are of variable length, which requires a more sophisticated psuedo-record and indexing arrangement for the data dictionary. Usually, character variables do not need to be part of the variable dictionary, although they are required for variable names, descriptions, and units. Techniques for handling variable-length character string storage will be discussed in the Advanced Concepts section (4.1).

3.1.1 COMMON Block Definitions

This is a list of COMMON block definitions for variable storage in the data dictionary. If some variable types are not needed in a particular program, their COMMON blocks should be defined with array sizes of unity.

```
COMMON /CREAL/SRNUMS(MREALP),MREALS    ! SINGLE PRECISION REAL
COMMON /CREALU/SRUNS,SRLAB             !   UNITS AND LABELS
COMMON /CDOUB/DRNUMS(MDOUBP),MDOUB     ! DOUBLE PRECISION REAL
COMMON /CDOUBU/DRUNS,DRLAB             !   UNITS AND LABELS
COMMON /CCOMP/CNUMS(MCOMPP),MCOMP      ! COMPLEX
COMMON /CCOMPU/COMPUNS,COMPLAB         !   UNITS AND LABELS
COMMON /CINT/INUMS(MINTP),MINTS        ! SINGLE PRECISION INT.
COMMON /CINTU/IUNS,ILAB                !   UNITS AND LABELS
COMMON /CLOG/LOGS,MLOGS                ! LOGICAL
COMMON /CLOGU/LUNS,LLAB                !   UNITS AND LABELS
COMMON /CCHAR/CHARS,MCHARS             ! CHARACTER STRINGS
COMMON /CCHAR/CHARUNS,CHARLAB          !   UNITS AND LABELS
```

3.2 Program Routine Requirements

The data stored in the COMMON blocks is referenced in two separate ways: in the program itself through EQUIVALENCE statements, and by the user through the data dictionary routines. Strictly speaking, the variable name that the program uses does not have to be the same name that the user deals with. However, it is important that the names be consistent. If it is necessary to deal with alternate names (for example, an old database input that uses different variable names for a different program), an aliasing or substitution capability can be implemented. That is discussed in Section 4.

3.2.1 Variable Access in the Program

The program routines use variables in the data dictionary by defining EQUIVALENCE statements. For consistency, the routine variable name should be the same as the data dictionary name. Here is an example of a layout for setting up the real variables:

```
SUBROUTINE TESTEM
PARAMETER(MREALS=100)
COMMON /CREAL/RNUMS(MREALS)
EQUIVALENCE (ARNUM(1)  ,START)
EQUIVALENCE (ARNUM(2)  ,STOP)
EQUIVALENCE (ARNUM(3)  ,CNBETA)
EQUIVALENCE (ARNUM(4)  ,CLIFT)
```

```
EQUIVALENCE (ARNUM(99) ,DOINIT)
STOP=START+100.
CNBETA=DOINIT
CLIFT=DOINIT
RETURN
END
```

Notice that, while the COMMON blocks describe the entire memory allocation, only those variables that are used from the data dictionary need to have EQUIVALENCE statements. The corollary is that, if the data dictionary is altered, EQUIVALENCE statements in affected routines should be updated. This is an excellent application for a computer-aided software engineering tool.

Although many EQUIVALENCE statements are required to get access to the data dictionary, modern text editors (such as TPU or even EDT on VAX computers) can automate much of the chores in creating that part of the code. This part of the data dictionary scheme is actually very easy, since there is a one-to-one correspondence to a defined memory location (array element) for every variable. Each program should have a central set of code that includes all of the EQUIVALENCE statements and acts as a control for program updates. That routine is described in the next section. However, it is worth noting here that, even if the EQUIVALENCE statements grow to a large number of lines, text copying and differencing utilities provide a quick way to incorporate the appropriate changes in all the routines in a program. The use of advanced aliasing and substitution techniques can also alleviate maintenance effort in a program with a data dictionary of this type.

3.2.2 Programmed Variable Access for the User

In this scheme, a pivotal subroutine is used by both the main application and the data dictionary routine library to set up the variable name correspondence for input and output. This routine is called 'DORF', which stands for Dictionary Organization Reference Function. DORF contains all the variable EQUIVALENCE statements for the COMMON blocks. It also contains the DATA statements that set corresponding character string arrays to the variable names used in the program, as well as storing the variable units. This one-on-one correspondence is best shown by simple example:

```
SUBROUTINE DORF
PARAMETER(MREALS=4)
CHARACTER*24 SRLABS(MREALS)
CHARACTER*10 SRUNS(MREALS)
```

```
COMMON /CREAL/SRNUMS(MREALS)
COMMON /CRUNS/SRUNS,RLABS
EQUIVALENCE (SRNUM(1)  ,START)
EQUIVALENCE (SRNUM(2)  ,STOP)
EQUIVALENCE (SRNUM(3)  ,CNBETA)
EQUIVALENCE (SRNUM(4)  ,CLIFT)
DATA SRLABS(1)  /'START'/,  SRUNS(1)  /'SEC'/
DATA SRLABS(2)  /'STOP'/,   SRUNS(2)  /'SEC'/
DATA SRLABS(3)  /'CNBETA'/, SRUNS(3)  /'1/DEG'/
DATA SRLABS(4)  /'CLIFT'/,  SRUNS(4)  /'NONE'/
RETURN
END
```

This routine constitutes the actual data dictionary. Support routines read this database to read or write data to the proper elements of the various arrays.

This is a simplified example, so some details have been left out. For example, in scanning the dictionary, there is a different array of names for each type of variable. The support routines must scan each name array to determine the type of the variable. Also, if the scanning is to proceed quickly, the names must be presorted for fast searching operations. Additional memory is needed for the integer indexing arrays, and that should be allocated in DORF. Of course, this operation will not alter the storage order specified by the user (which may be determined by other considerations).

3.3 Basic File Variable, Array, and Table Input

Assuming that a variable name has been defined for the program, it can be used in the input stream to define the variable value. Also, data functions can be entered which describe data bases and relations associated with variables in the data dictionary. Additionally, it will be useful to define vectors and arrays within the data dictionary storage space.

Functional input must be well defined so the correct variables are associated with the input table structure in the right way. This is a complex definition that will be provided elsewhere. Function input formats are discussed in the AlphaZulu Data Format specification. As a matter of course, the input stream should flag the appearance of a function. Also, while it is practical to enter variables and simple functions interactively, it is unlikely that multi-dimensional

tables will be entered outside of a specialized function editor program.

For full flexibility, the input stream could be a mix of data dictionary references, AZDF functions, FORTRAN namelist type input, or the old-style fixed format or card image input lines. The only normal restriction on this type of input is a requirement for text data (no embedded binary data) since it is intended for direct editing. Given that, it is natural to define comment text capability. These are all handled using a scheme that has roots in the namelist input environment. Namelist input is set off by special characters and key words. Here is an example of namelist input:

```
$CONTROL
      TITLE='TESTT002AA',
      INTERVAL=1,
      RESET=.TRUE.,
      START=10.2,
      STOP=14.5
$END
```

The namelist input stream is delimited by the dollar signs. The indentation is not required. The first dollar sign is followed by a label which identifies the particular namelist declaration in the program. Either a dollar sign or an ampersand (&) may be used as a delimiter. Most compilers require the delimiters to be the first character on a line.

The AlphaZulu Data Format uses special characters as well. Function header information is denoted by a '>' as the first nonblank character on a line. Component information is denoted by a '+' as the first nonblank character on a line. Text lines which don't meet these tests are interpreted as comments associated with the preceeding header line for the most part.

Input through the data dictionary definitions will use the '<' character for a delimiter. The beginning delimiter will be '<SET', and the ending delimiter will be '<ENDSET'. Since comments are needed anywhere in the input, the '!' (exclamation point) is defined as the comment delimiter. Characters following the '!' are not interpreted, and a single line between the data dictionary inputs can be a comment if '!' is the first nonblank character. This is a sample data dictionary input:

```
<SET
      INTERVAL=1, RESET=.TRUE.
```

```
      START=10.2 SEC
      STOP=14.5 SEC
      ARRAY(2,5)=STOP
<ENDSET
```

Unlike the namelist input, a comma is only needed to separate two assignments on the same line. While data dictionary input should allow some arithmetic, only simple assignments are part of the basic definition. In other words, a mathematical parser is left as a later enhancement since it requires another level of memory allocation, and probably some of the features of advanced data dictionary design covered in section 4.0.

Note that there is another difference with the namelist example. The character variable assignment is not shown. Units are attached to some of the variables, too. Blank characters separate constants or variable names from units, which is similar to the way the numbers would be written. Units may be delimited differently when a mathematical parser is designed, especially if the FORTRAN equation syntax is used.

3.4 Basic Interactive Variable and Table Editing

Once the program has a list of defined variable names, and a means to locate the variable in the COMMON block, interactive variable editing and table editing is an extremely simple operation. This is basically equivalent to entering the assignment string and calling the assignment interpreter used for file input. However, error handling must be recoverable for the inevitable typographical errors. It is not acceptable for the program to show the error and stop. This should be handled by the input routines provided as part of the data dictionary system.

Interactive use also requires simple listing capability from the data dictionary. The user should be able to query for current variable values and for variable names. The user must be able to perform searches on names or values (or both) for those applications with large dictionaries and/or arrays. These capabilities should also be provided by the data dictionary routines. More detail than this requires design of the actual data dictionary routine library, which will not be covered here.

4.0 Advanced Data Dictionary Design

Section 3 described the capabilities and features of a simple data dictionary design. It is possible to greatly augment the data dictionary concept with a few more programming techniques. These techniques are mainly transparent to the application programmer, but require a high level of sophistication in the routines supporting the data dictionary.

4.1 Variable Length String Storage in FORTRAN

The scheme described here is very similar to a stack-oriented string allocation within an emulated disk or block memory allocation. Memory is allocated in the form of a vector of fixed-length character strings:

```
PARAMETER (MCSTOREP=1000,LSTR=100)
CHARACTER*LSTR CSTORE(MCSTOREP)
DATA MCSTORE/MCSTOREP/
COMMON /STRSTORE/ CSTORE
```

Pointers are used to access any string in the storage array.

```
INTEGER ICSTORE(MCSTOREP,3)
```

Where:
- ICSTORE(I,1) = Pointer to array element containing first character in string.
- ICSTORE(I,2) = Pointer to character position of first character of string in buffer CSTORE(ICSTORE(I,1))
- ICSTORE(I,3) = Length of the string

Storage is allocated from low to high indices. When the string index results in a reference to memory outside the fixed buffer (virtual disk), then an attempt must be made to clear out deleted strings. Deleted strings are marked with a control character, decimal 12 (formfeed) in the first character position. The remaining strings are packed into low vector addresses, a process called garbage collection. If the string still cannot be stored, an error condition has occurred.

On occasion, it is necessary to store arrays of variable-length character strings. In that case, another layer of indirection is added and the scheme operates in a similar manner. It should be mentioned

that a separate storage area, or virtual disk, is created for every category of string variable. For example, one string array would be created for data dictionary variable names, another for temporary variable names, and so on.

4.1.1 Variable Aliasing and Wildcards

There are many circumstances where it is useful to refer to the same program variable by several names. For example, large input data sets may contain outmoded variable names in the input that would be tedious to change to the new naming convention. Another case may arise where the program is to be applied to several permutations of the same input, where it is more convenient to re-route numbers by alias than to edit the data stream. Perhaps the most important application is in allowing the programmer to input variables using the same variable names as the program code, while the user is encouraged to use more descriptive input variables. Of course, an experienced user could even define shorthand input variable names and save some input effort.

Aliasing can be implemented a number of ways, depending on memory limitations. The simplest technique is to allocate a pointer array that describes lists of names for each variable in the data dictionary. The routines for accessing string lists provide most of the functions for this scheme. Alternatively, the variable names are altered to build linked lists of alias names, and the program detects the links from the context of the name itself.

The scheme for building linked lists of variable names is simple, although it requires its own set of support code. For example, a special character is appended on the first variable name, the vertical bar (|). Alias names are appended after the vertical bar. An example of three alias names on a root name would be the string "CL|LiftCoef|Cl|CoefLift".

Finally, it is extremely useful to provide a wild card capability in variable names. Wild cards are special characters that allow substitution in matching variables names. Common wild cards are the '*', which substitutes for any number of characters, and '?', which substitutes for any single character.

Some examples of wild cards follow:

Pattern	Matches			
CL*	CLMAX	CLmax	CLFlap	CL
*Flap	CLFlap	Flap	CDFlap	
A*D	AD	ABD	ABCFD	
CL?	CL1	CLD		
CL?4	CLE4	CL44		
CL??	CLE4	CLAA		
?D	CD	AD		

Wild cards should be available to the user to do searches or listings of variables in the program. On input, wildcards would give the ability to read variables with extra information such as run numbers (eg RMB134).

4.1.2 Interactive Variable Allocation

Efficient string allocation methods allow allocation of 'extra' variables for performing analysis beyond a basic program's capabilities. This is useful for a number of features, such as function evaluation, group references, or command files (to be discussed later).

Error detection and quality control requires a declaration of new variables in the input stream. Declaration of vectors and arrays is not defined here. Simple variables require a declaration of type, precision, and name. In a non-interactive input stream, a special command is used to allocate a variable, as shown.

```
{define} YawCoef SREAL
{define} Title STR 42
```

The first word is the command, the second is the name, and the third is a keyword describing type and precision of the variable. String variables have a fourth word (third parameter) that declares the length of the string. This list covers the normal range of FORTRAN data types:

KEYWORD	TYPE AND PRECISION
SREAL	Single precision real
DREAL	Double precision real
COMP	Complex
SINT	Single precision integer
DINT	Double precision integer
LOG	Logical

CHAR	Single character
STR	Character string

4.2 Text Description of Variables

So far, the data dictionary encompasses definition of variables for the program and not the user of the program. The user should have the ability to obtain helpful text describing each variable. This improves the learning curve for new users and improves the quality of the work done with a program, particularly if it is used in an interactive manner.

Although facilities for character string lists discussed so far lend themselves to implementing descriptive text, experience shows that use of variables tends to change rapidly compared to the frequency of program updates. In other words, the variable usage text, if encoded in the program, would quickly become outdated, or require frequent (expensive) compiling and debugging of the program.

Since the variable names are available to the program already, it should be sufficient to provide routines to access and display descriptive text from data files. This way the program does not have to be updated every time a variable description changes. In fact, this facility should also provide usage information on the program itself. The variable names could be used by the program to perform searches on a single "help" file that contains the program help and the variable descriptions that could be displayed for the user.

The scheme favored here is to use special characters to set off the variable names in the text file. This way the descriptive text is located unambiguously. The pound sign, '#', is favored since it is rarely used in text and it is visually easy to locate (eg #CLMAX#). The end of the descriptive text should be set off by a null variable, '##'.

4.3 Journaling

Interactive programs offer several advantages to the user as long as the computer, program, or user operate correctly. Given one error, however, the results can range from crashing the computer to a simple error that may be difficult to trace back to the input or the operation of the program. One technique that helps solve these problems is called "journaling". Any action that is related to setting

input or performing computation is echoed to a special journal file. This file survives even the worst of machine crashes, enabling the user to reconstruct events leading up to a disaster. Even an ordinary machine crash (due to some other cause) does not cause the loss of hours of work.

The best journal file implementation is a "reversible" journal. In other words, the journal file can also be input to the program to reproduce the interactive work of the user. The user should have the option of picking up the work at that point. Given this capability, it is a simple matter to run an analysis interactively and then edit the journal file to do the same analysis on other data in a "batch" mode. Journal files of this nature are generally called command files, since the user will use them in a batch sense more than in the error recovery sense (hopefully).

The guidelines for data input established earlier provide the specifications for journal file input recording formats. New guidelines are only required for command recording. It is desired that commands be delimited in a manner that fully describes any tree structures present in a menu scheme. For this scheme, commands are delimited by curly brackets, with tree structure separated by the back slash character. For example:

```
{FILE\READ}
 <SET
   INFILE="GOOD.DAT"
 <ENDSET
```

Command file capability leads naturally into two useful features. First, the user should be able to interactively execute command files. Second, command files add functionality to a program with interactive function execution capability. For example, if the user can enter and execute formulas interactively (which implies run-time variable allocation), then the command files can be written as general routines and called in the same manner as native functions of the program.

4.4 Input Error Detection

As a further expansion on the ideas presented in Section 4.3, the features of run-time function definition and evaluation can be applied to improve input error detection. Simply put, each input should lie within reasonable bounds. Sometimes the bounds are very

well defined, in which case the input is compared against some constant values. It is also necessary to be able to determine if a variable is a reasonable sign, to compare reasonable magnitude and sign dependent on other variables, and to specify what error messages and actions will be taken if an error is encountered.

4.5 Group References

Group referencing is a very powerful capability that allows passing large arbitrary collections of variables to functions. This technique is commonly called data structuring in computer science. The technique promotes efficient and correct handling of data, and it provides a flexible shorthand for data manipulation, input, and output. It is most useful when it is available in an interactive or command file driven manner.

The technique requires the capability to create named lists of variables, which is essentially a simple variation of the basic string variable allocation previously discussed. It is also necessary to allocate and define names for the data groups. Here is an example of a data reference definition:

```
{define/group} AeroCoef
    LiftCoef
    DragCoef
    SideFCoef
    PitchCoef
    YawCoef
    RollCoef
{define/endgroup}
```

Notice that types and units are not required since the variables should already be defined in the usual manner.

4.5.1 Interactive Initialization by Reference

It is often helpful to be able to stop a computation, save intermediate results, or initialize a computation using a simple command. This is difficult to do in a flexible and concise manner unless the group reference feature is available. Since group references allow access to any variable in the data dictionary, it is only necessary to implement functions that save, load, and initialize variables by group reference. This is a fairly trivial option. It must be emphasized that file manipulations should preserve the command

file structure so that the user can understand and use the results of group reference manipulation.

4.5.2 Interactive Output Definition

The data dictionary schemes provide an extremely flexible means of entering data to a program, verifying data reasonableness, and manipulating data. However, since the advanced features discussed so far are required to properly do data output, the output features of a data dictionary scheme must also be classed as advanced techniques. However, this is one of the most desirable capabilities. If processed data cannot be output or stored in a flexible manner, then effort and resources must be expended to do simple output changes, and the reprogramming involved increases risk of endemic program error.

The range of output schemes is so tremendous that they are beyond the scope of this paper to deal with in detail. The type of output desired drives interactive output specification so completely that a simple example would also be a useless example. Instead, it is better to concentrate on the chief issues that would drive the interface design.

Primarily, what sort of output is to be specified? If it is necessary to produce computer files equivalent to printed output, then page layout functions are needed. Functions are required that specify the page size, margins, number of lines per inch, number of characters per inch, where and what text will appear on the page, and the output fields for data from the program. The specification of variable groups to associate with the output is a small part of this. If data files for random access are to be produced, then the output specification commands are different, as they would be for plot output.

Other questions should be asked. Should the output be device independent? Should some standard data file format be supported? Should layout be interactive, or through a mouse? How are control characters and escape sequences supported? Are graphics supported? Detailed design of the output formatter is driven by these matters.

4.6 Program Data Tracing

Most FORTRAN programming systems (compilers, linkers, libraries, and so on) are supplied with a program called a 'debugger'. This runs the FORTRAN program and allows interactive inspection of variables, single-line execution of the program, setting of breakpoints in the program, and so forth. While this is, in fact, a viable alternative for many data dictionary functions, it has three major drawbacks: the debugger cannot be extended to offer the other functions mentioned here, it does not support variable units, and it requires a high level of expertise in FORTRAN to understand and use well.

It is possible to implement many of the debugger functions through the data dictionary, and several more besides. Use of a "tracer" subroutine call in important sections of code can provide information on variables in an interactive or batch manner. Since all important variables should be available in the data dictionary, it would be possible to display any variable of interest without re-programming the tracer routine. It could be made generic and part of the data dictionary library.

The tracer call should provide many useful functions:

1. Interactive or batch reports on changed variables. The variables should be selectable.
2. Interactive or batch reports on running time between tracer routine calls.
3. Setting of breakpoints at tracer call locations.
4. Display of tracer call locations.

The last capability really requires that a location label is supplied to the tracer routine. A sample tracer routine call:

```
TRACER('GMOD','INIT','GEAR MODEL ROUTINE INITIALIZE')
```

The first string is the subroutine or function name, the second is a sub-level label, and the third is a descriptive string.