IBM Application Program

GH20-0594-1

Problem Language Analyzer (PLAN)

Program Description Manual

Program Nos.1130-CX-25X, 360A-CX-26X, 360A-CX-27X

This system provides users with an efficient means of implementing and using meaningful user-oriented (problem-oriented) languages. This manual is intended to provide rules for use of the system and technical specifications defining the scope of applicability. It is intended to serve as a user's and an implementer's reference.

PREFACE

This manual is divided according to a three-level dependency. The subdivisions are noted in a three-part ascending decimal number (Example: 1.2.3). A change in the first decimal portion represents a change in major topic; a change in the second decimal portion represents the intermediate level; a change in the third decimal portion represents the minor level.

In addition to prefixing the headings, the decimal numbers are found within the table of contents, on the quick-reference line on the bottom of each page, in the index to show usage of terms, and within the body of the manual to show cross-references. The quick-reference line depicts the last major heading at the top of a page and does not necessarily reflect the last minor or intermediate level change.

Insight into the use of this manual may be gained by first reading Problem Language Analyzer (PLAN) Users' Introduction (H20-0626).

This manual includes:

Introduction to Basic PLAN Features
Use of PLAN for Problem Solving
PLAN System Support for Application Designers
Programming Support in PLAN

Second Edition (November 1969)

This edition is a major revision obsoleting H20-0594-0.

This edition applies to Version 1, Modification Level 1, of the program product Problem Language Analyzer (PLAN) (1130-CX-25X, 360A-CX-26X, 360A-CX-27X) and to all subsequent versions and modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are continually made to the information herein. Therefore, before using this publication, consult the latest 1130 and System/360 SRL Newsletters (N20-1130, N20-0360) for the editions that are applicable and current.

Copies of this and other IBM publications can be obtained through IBM branch offices.

A form has been provided at the back of this publication for readers' comments. If this form has been removed, address comments to: IBM Corporation, Technical Publications Department, 112 East Post Road, White Plains, N. Y. 10601.

CONTENTS

FIGURES

1.0.0 INTRODUCTION

The Problem Language Analyzer (PLAN) is designed to allow implementation of desirable user-oriented (problem-oriented) languages by providing a common language processor. Previously, problem-oriented languages have required independent language processors that were in themselves major implementation tasks. Even though highly desirable, problem-oriented languages were implemented only for major applications. Reimplementation on new equipment has made long-term costs even higher.

The PLAN system through a common language processor allows input to a job to be composed of several dissimilar problem-oriented languages, all operating in a homogeneous environment. It also allows easy modification and expansion of existing applications and problem-oriented languages. The PLAN concept of implementation makes complete machine independence of logic modules more easily attainable.

Logic module loading is accomplished dynamically at execution time as defined by the current job description. Logic modules are loaded only as required and existing logic modules do not require modification to incorporate new processing capabilities. Multiple implementation of the PLAN system for the IBM 1130 using Disk Monitor, Version II and the IBM System/360 using DOS/360 or OS/360, allow logic modules written in machine-independent ASA BASIC FORTRAN IV to be executed on either computer system. A job is described in problem-oriented terms in a language compatible with all systems.

In general, implementation of a problem-solving system operating within a PLAN environment involves several tasks as defined below:

1. Definition of the problem-oriented language. This definition is processed by PLAN to create a language dictionary.

2. Programming of logic modules (if existent logic modules do not suffice) to support the problem solution functions (note that this does not encompass problems of language processing; these are handled by PLAN).

3. Generation of problem-oriented language statements to describe the particular problem to be solved.

2.0.0 SYSTEM OVERVIEW

2.10.0 THE PLAN FUNCTION

The Problem Language Analyzer (PLAN) is an application development tool that is designed to assist application developers in the implementation of problem solving systems by reducing development cost and reducing the time and effort of the implementation and maintenance cycle.

An introduction to the objectives of the PLAN system may be found in the Problem Language Analyzer (PLAN) Application Description Manual (H20-0490). It is recommended that it be read before continuing.

PLAN, itself, does not provide the solution to a user's application problem. Its prime function is to assist the user in solving his problems by providing the support to meet the following objectives.

- PLAN is designed to provide for continuity of application effort across (1) applications, (2) machine systems, (3) operating systems, and (4) user boundaries. The concept of implementing a particular application system for a particular user utilizing a particular operating system on a particular computing system is not considered to be a valid constraint in the PLAN environment.
- PLAN is designed to allow paced, incremental, orderly growth of problem solving systems by providing open-ended growth ability, thereby reducing the cost of application development.
- PLAN is designed to provide a generalized, interactive user-oriented language facility in which the vocabulary of the language is user definable. In the PLAN environment, batch processing is treated as a special case of interactive processing in which a nonresponse by the user is predetermined. The user may switch between batch and interactive as conditions dictate and systems allow with no loss of operating efficiency as long as the available configuration provides a supported interactive device.
- PLAN is designed to reduce the regimentation required of a user in communicating the description of a problem to a computer. This is accomplished by providing a problem defining (note carefully that we did not say problem solving) language facility that is comprised of only those terms and data that the user chooses to use for the problem description. A mode of problem description is thus provided that does not require the learning of new conventions in transferring to new applications on new systems.

2.20.0 PERSONNEL REQUIREMENTS

Each installation can have an open-ended vocabulary of commands, descriptive phrases, data symbols and implied standard data values. This allows the users of the system to submit problems, using a simplified version (subset) of the vocabulary of the department or industry.

PLAN includes the programs required to establish a local language dictionary and to interpret input statements using local dictionary entries. No additional programming is required to define or interpret PLAN user-oriented language statements.

Individuals representing three functional groups contribute to application development under PLAN.

The System Analyst provides for input definition, data standards, input editing and validation, and program module sequencing through language definition. The function of the system analyst in his role as system designer is significantly more important with the PLAN concept of problem solving if modularity and reuse of modules is to be assured, since many times reusability cannot be appropriately measured at the programmer and user level.

The Programmer provides functional logic modules that should be segmented for maximum flexibility. The PLAN conventions support this flexibility by eliminating the need for data definition statements, program names, and data constants in the source code. The programming of algorithms is considerably simplified.

The User combines the efforts of systems analysts and programmers at execution time by describing a real problem with its data. These statements and the language definition determine the sequence of logic modules to be executed in each case.

2.25.0 PLAN OPERATING ENVIRONMENT

Application programming can grow in a planned, orderly manner without reworking previously completed segments each time a new capability is added. To add new capabilities to any system simply requires (1) that command(s) be added to a language dictionary that define the new capability and (2) that the logic modules (if new ones are required) be added to a program library. Thus, the well-defined portions of any application can be made functional while the less precisely defined portions are defined and developed.

The PLAN processor provides for processing of intermixed user-oriented language statements that logically define a problem to be solved. The free-form statements are executed immediately as each is read. Therefore, subsequent statements may be entered on the basis of results from the current statement, if the statements are entered on a console device. In concept, the PLAN processor is the only mainline program executed. It has the potential for controlling processing by the loading and interpretation of commands and the loading of logic modules defined to be executed as a result of processing the commands.

The conventional approach, when given a problem for which there is no available program, is to write a new mainline program linking available subroutines to suit the new problem.

PLAN allows existing logic modules to be linked without new source code, simply following the execution sequence implied by the user's input statements (problem description). There is no compilation of the problem description. The logic modules implied by the problem description are linked dynamically during execution. Only those program modules actually required for execution are loaded. The logic modules to be executed under PLAN should be single-functioned; they must obey certain coding conventions; and they must be stored in a system library.

The application logic modules are made machine independent by using ASA BASIC FORTRAN IV language statements that include CALLs to standardized (PLAN) subroutines for linkage, direct access file processing, sequential input/output, utility functions, and to provide compatibility for functions that vary between IBM systems. Any programming language can be used, with potential loss of machine-independence, if the FORTRAN coding and linkage conventions are maintained.

2.30.0 IMPLEMENTATION PROCEDURES

An illustration of the general concepts involved in operating the PLAN system is found in Figure 1. Before this can be considered, the following facilities must be present:

1. An input device from which PLAN commands can be accepted.
2. An output device through which PLAN may communicate with the user.
3. A phrase dictionary that contains PLAN job definitions.
4. A library of executable programs.

To provide these facilities to PLAN, a three-step process, illustrated by Figure 1, must be executed. These steps are:

1. Generate the required programs for the job by compiling/assembling the appropriate source language.
2. Define the job requirements by adding phrases to a PLAN phrase dictionary. The PLAN phrase is a definition of a PLAN job. It consists of a list of problem programs to be executed and a list of input parameters and/or constants.
3. Execute the necessary PLAN commands to run the job. A PLAN command is a statement that causes the PLAN system to invoke or execute a certain phrase description.

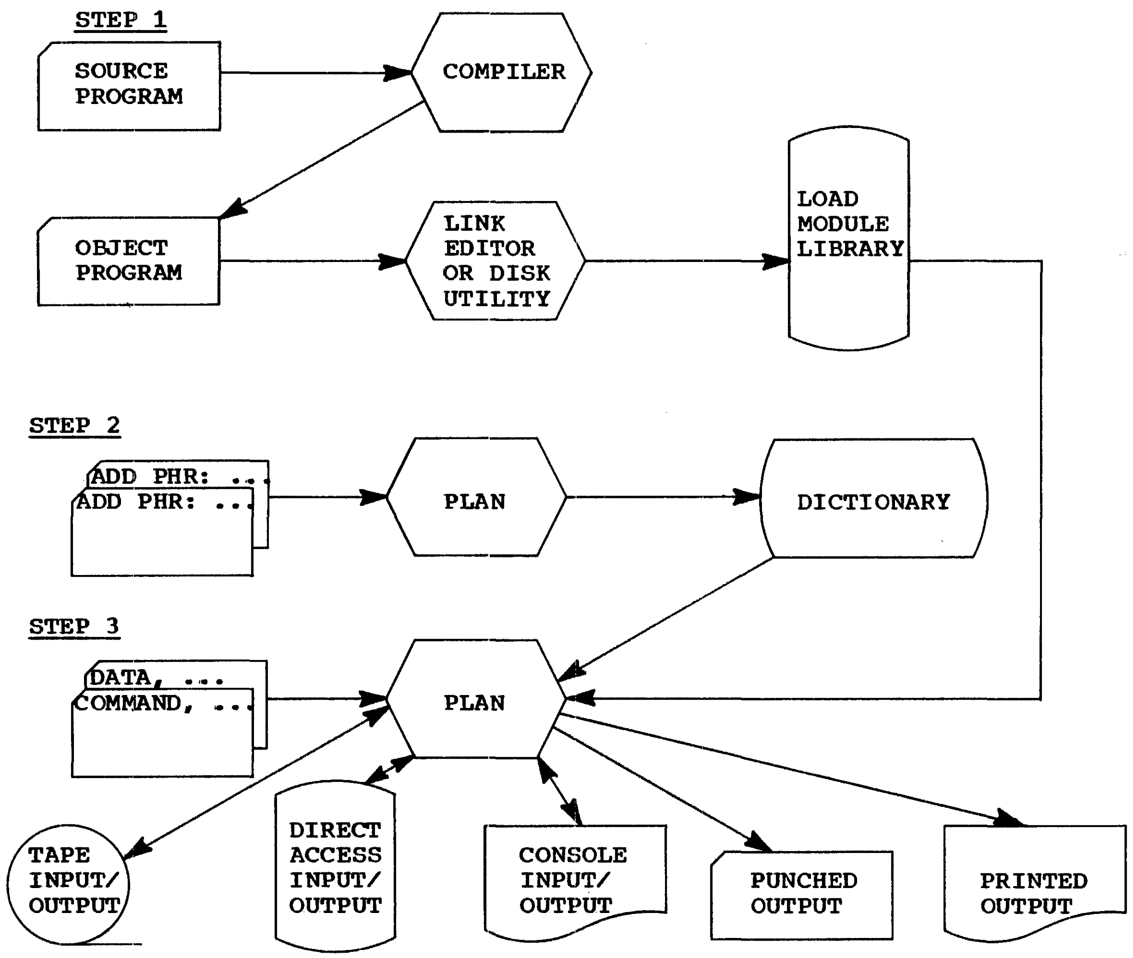


Figure 1. Necessary steps for PLAN execution

2.40.0 MEMORY ORGANIZATION

During PLAN processing, that is, throughout the full cycle in which PLAN is in control, the computer memory is divided into eight distinct areas. These areas and the functions of the areas are defined below:

1. System area. This area is that portion of memory required for hardware use (in-core registers, I/O areas, etc.) and for operating system/monitor use. The size of the area is variable for each system.

2. PLAN loader area. This area is that portion of memory permanently occupied by the PLAN loader. The PLAN loader controls program loading and command (phrase) processing. This array, the first one in blank COMMON, is 625 32-bit words in length.

3. System Switch Words. This is an area of switch words used for control of PLAN and for communication between PLAN and problem logic modules. This area is 15 32-bit words in length.

4. Managed data array. This is an array protected from overlay by PLAN modules, residing in blank COMMON, managed by PLAN according to a user-defined level structure. The managed array may be defined to a size (maximum of 32,767) desired by the user and within the capacity of the computer system.

5. Nonmanaged data array. The nonmanaged array is defined as that portion of blank COMMON not included in the PLAN loader area, the system switch area, or the managed data array that is protected from overlay by the PLAN system modules. The combined size of the managed and nonmanaged array is limited to a maximum that is variable on each computer system configuration. The nonmanaged array can be used for any working storage requirement or transmittal of data between logic modules where a level structure is not required.

Note that the name "communication array" is used to describe the combined managed and nonmanaged data array.

6. PLAN system area. In some implementations of PLAN, additional space is required for residence of code to support various functions. The size of this area is variable. This area is required by System/360 OS and DOS PLAN. Specification of the core size required may be found in the respective Operations Manual.

7. Application program area. This area is that portion of memory remaining, into which PLAN loads the application logic modules. The size of this area is variable; it is a function of the other five areas and the computer/partition size.

8. FORTRAN I/O work area. This area is required on DOS PLAN only and only if FORTRAN I/O is utilized. Its size is defined by the user at PLAN execution time but must be a minimum of 512 bytes if FORTRAN I/O is used.

```
+--------------------------------------+-+-+
|PLAN LOADER AREA                      |C|8|
| (625 32-BIT WORDS)                   |O| |
|--------------------------------------|M|K|
|PLAN SWITCH WORDS                     |M| |
| (15 32-BIT WORDS)                    |O|M|
|--------------------------------------|N|I|
|MANAGED                               | |N|
| (VARIABLE)                           | |I|
|-----------     COMMUNICATION         | |M|
|NONMANAGED      ARRAY                 | |U|
| (VARIABLE)                           | |M|
|--------------------------------------+-| |
|                                        | |
|APPLICATION PROGRAM  AREA               | |
|(PLAN MODULE LOAD AREA)                 | |
|(VARIABLE)                              | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|----------------------------------------| |
|RESIDENT 1130 MONITOR                   | |
|(510 MACHINE WORDS)                     | |
+----------------------------------------+-+
```

Figure 2. IBM 1130 organization under PLAN

```
+----------------------------------------+-+
|FORTRAN I/O WORK AREA                   | |
|----------------------------------------| |
|PLAN SYSTEM AREA                        | |
|----------------------------------------| |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|APPLICATION PROGRAM AREA                | |
|                                        | |
|                                        |2|
|                                        |4|
|                                        | |
|                                        |K|
|--------------------------------------+-| |
|NONMANAGED                            | | |
| (VARIABLE)                           | |M|
|-----------     COMMUNICATION         |C|I|
|MANAGED         ARRAY                 |O|N|
| (VARIABLE)                           |M|I|
|--------------------------------------|M|M|
|PLAN SWITCH WORDS                     |O|U|
| (15 32-BIT WORDS)                    |N|M|
|--------------------------------------| | |
|PLAN LOADER AREA                      | | |
| (625 32-bit WORDS)                   | | |
|--------------------------------------+-+-|
|DOS SUPERVISOR                            |
+------------------------------------------+
```

Figure 3. IBM DOS/360 partition organization under PLAN

```
+----------------------------------------+-+
|                                        |3|
|PLAN SYSTEM AREA                        |2|
|----------------------------------------| |
|FREE STORAGE FOR GETMAIN                |K|
|----------------------------------------| |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|                                        | |
|APPLICATION PROGRAM AREA                |M|
|                                        |I|
|                                        |N|
|--------------------------------------+-|I|
|NONMANAGED                            |C|M|
| (VARIABLE)                           |O|U|
|-----------     COMMUNICATION         |M|M|
|MANAGED         ARRAY                 |M| |
| (VARIABLE)                           |O| |
|--------------------------------------|N| |
|PLAN SWITCH WORDS                     | | |
| (15 32-BIT WORDS)                    | | |
|--------------------------------------| | |
|PLAN LOADER AREA                      | | |
| (625 32-BIT WORDS)                   | | |
+--------------------------------------+-+-+
```

Figure 4. IBM OS/360 partition organization under PLAN

2.50.0 GENERAL FUNCTIONAL PLAN ORGANIZATION

The PLAN system can be described in terms of five functional areas. These five areas are discussed below in terms of the general requirement for the function and the capability provided by PLAN in the area. The areas are:

- Dynamic Job Supervision
- User-oriented Language Processing
- Diagnostic Supervision
- Input/Output Control
- Utility Function Control

The reader is reminded, while reading the following sections, to keep in mind that PLAN is a system for problem solving made up of many pieces. A small percentage of the features discussed can be split out of the PLAN environment and used independently of PLAN. On the other hand, most of the facilities of PLAN need be used only to the degree required by the user. This system overview should allow the user to determine which features of PLAN he is interested in and therefore serve as a guide in directing his reading of the remainder of this manual.

2.50.10 DYNAMIC JOB SUPERVISION

The supposition for years has been that a programmer or system analyst could predetermine all desired capabilities at the outset of planning an application system and therefore preplan all the required system logic paths. Unfortunately, this rarely proves to be the case.

How many times, following tedious months of planning and implementation, during the celebration of a system finally working has the question "What would it take to ..." been asked? All too frequently, the answer is "restructure". Would it not then be desirable to have a less rigorous structure for application systems?

As application areas expand and new problems are tackled, it becomes apparent that total and complete problem solving structures cannot be provided. In a bank, for example; we don't know what type of transaction will be next, nor do we know that a new type of transaction will not be invented tomorrow.

In information management, we don't know the next question to be asked (nor, for that matter, can we dare assume that we have presupposed all the questions that may be postulated). In engineering, we would be great indeed if we could predetermine and plan for all combinations of problem parameters and design criteria.

Hence, the PLAN DYNAMIC loader and job supervisor. When a user decides to operate under the PLAN system, the PLAN loader is scheduled as a normal monitor/operating system job and control is passed to it.

Control must be returned to it at the termination of each user's program module as long as execution in the PLAN environment is desired. This is the first requirement for running in the PLAN environment. Providing an exit for user's modules to the PLAN loader is one of two requirements that a user must fulfill to run under PLAN. Additional information on this function is provided at a later point in this overview.

The PLAN loader is made up of two major components, (1) the program loader, and (2) the execution sequencer (hereafter called the pop-up list). The PLAN loader must remain resident throughout the entire PLAN run and is therefore placed in the first 2560 bytes of BLANK COMMON. Protection of the PLAN loader by definition of the required COMMON is the second of the two requirements a user must satisfy to run under PLAN.

The pop-up list is a programmed mechanism for processing program names that is quite analogous to the tray unloader at a company cafeteria. When a name is removed from the list, a new one pops up until the list is empty. When a name is added to the list, the existing names are pushed down until the list is full. Names can also be inserted at the bottom of the list.

What does all of this have to do with job supervision? The pop-up list is used to indicate the sequence of programs to execute. The top name in the list is the next program to load. The user may at any time add names to the list or delete names from the list. Thus, an exchange of control exists in the PLAN environment. The PLAN loader picks the top name from the pop-up list, the program loader places it in memory and transfers control to the program. The program executes (and modifies the pop-up list if required) and returns control to the PLAN loader. The cycle is repeated until the list is empty.

The user is given the option of interfacing with the loader from his modules in several ways. In the following list, "modify" should be interpreted to mean "is given the facility of adding to or extracting from". The options are:

- Terminate the module and return to the PLAN loader
- Modify the pop-up list
- Modify the pop-up list, terminate the module, and return to the PLAN loader

- Modify the pop-up list, save the status of the module currently in execution for future restart, terminate the module, and return to the PLAN loader
- Modify the pop-up list and return to the PLAN loader to invoke a coexistent, dependent module

How do the names get into the pop-up list originally? What happens when the pop-up list is empty? The PLAN loader interprets an empty list as a special signal to load the language interpreter to read a new user-oriented language (UOL) statement from the PLAN input stream. (A discussion of language processing is contained in the next section.) Thus, one user-oriented language statement is required to initiate PLAN processing.

The system analyst or language definer may include a program list with the definition of any statement. The program list is the definition of those programs to be executed whenever the particular statement is encountered. The language interpreter upon encountering the statement, retrieves the program list and places it in the pop-up list. Thus, the UOL statements may define the sequence of execution. Since a new statement is not read until the pop-up list is empty, the system allows the user to examine the output resulting from one statement before entering the next. Thus, we find that PLAN provides for interactive processing. If the current PLAN input device is batch oriented (for example, a card reader), batch processing takes place.

To illustrate the UOL statement - pop-up list relationship, assume that the program list defined for the statement FUNCTION is 'PROGA, PROGB, PROGC'. As a result of the language interpreter reading the FUNCTION statement, the following sequence of executions would result:

- PROGA
- PLAN Loader
- PROGB
- PLAN Loader
- PROGC
- PLAN Loader
- PLAN Language Interpreter (to read new UOL statement)

This was an elementary discussion of dynamic program loading and sequencing and does not cover many of the powerful options open to the user. It should be carefully noted that in the above example, PROGA, PROGB, and PROGC have full facility for modifying the pop-up list and thereby altering the execution sequence. For example, if PROGB were a graphic application module, a light pen detect in PROGB could have caused it to cancel PROGC or to schedule PROGD. The PLAN supervisor gives the user and his program the control at all times to modify the sequence of execution to meed the ever changing problem solving environment.

Figure 5 illustrates the logic of program control through the PLAN pop-up list. It consists of six parts. Part 1 illustrates the system status at the time PLAN is invoked. PLAN is loaded by the monitor or operating system from the system program library. The pop-up list is initialized to a zero (marked as empty). As a function of initialization, PLAN determines if the language definition dictionary is initialized. This dictionary is a direct access file defined as PFILE on 1130 PLAN and DFJPFIL on System/360 DOS and OS PLAN. It is referred to in this document as "PFILE". Then, any time the pop-up list is found to be empty, PSCAN (the command reader and scanner) is placed in the pop-up list and is subsequently loaded into the program area. This sequence, as shown in Part 2, illustrates that the PLAN loader accesses PSCAN from the system program library.

Part 3 illustrates processing with PSCAN in control. A new command is read by PSCAN from the current PLAN input device. The meaning of the command is retrieved from the dictionary (PFILE). The definition may result in initialization values and data values being placed in the communication array. Program names may be placed into the pop-up list as a result of command definition. PSCAN returns control to the PLAN loader. When the pop-up list is not empty, processing continues as in Part 4.

Part 4 illustrates processing by PLAN when the pop-up list has an entry (in this case program name A). PLAN takes the top entry from the pop-up list and loads the designated program from the program library into the program area. Once loaded into the program area, the module name is deleted from the top of the pop-up list. For example purposes only, to display the flow of events, A has been allowed to appear at the top of the list even though it has already been loaded into the program area. This is important to note, because in reality, A would have been deleted from the top of the pop-up list. Control transfers from PLAN to the loaded program, and processing continues as defined in Part 6.

Part 5 illustrates processing after PLAN has transferred control to the program defined at the top of the pop-up list and now in the program area. Program A during execution may access information stored in the communication array and store information into the array for future use. It may, through the use of the PLAN loader subroutines, modify the module names in the pop-up list. When it has completed execution, it must return control to the PLAN loader. Processing then continues as defined in Part 2 or Part 4.

The new command processor PHRAS is in control in Part 6. The command image (if an ADD PHRASE) is processed and converted to internal text and placed into the system dictionary (PFILE). The DELETE or ALTER PHRASE results in the deletion of the coded old definition of the command. At the termination of processing, control is again transferred to the PLAN loader and the procedure is repeated as defined in Parts 2 and 3.

Commands need be defined to the system only once. They may be altered (deleted and re-added) or deleted at any time. After a command has been added to a language definition dictionary (maintained on disk), it may be executed in any sequence that is logically acceptable at any time within a PLAN command stack.

The principles of dynamic program loading through use of the PLAN loader and pop-up list, as defined in Figure 5, should be clearly understood. Following the figure step-by-step again at this time may prove to be a worthwhile investment in terms of understanding material that follows in this manual.

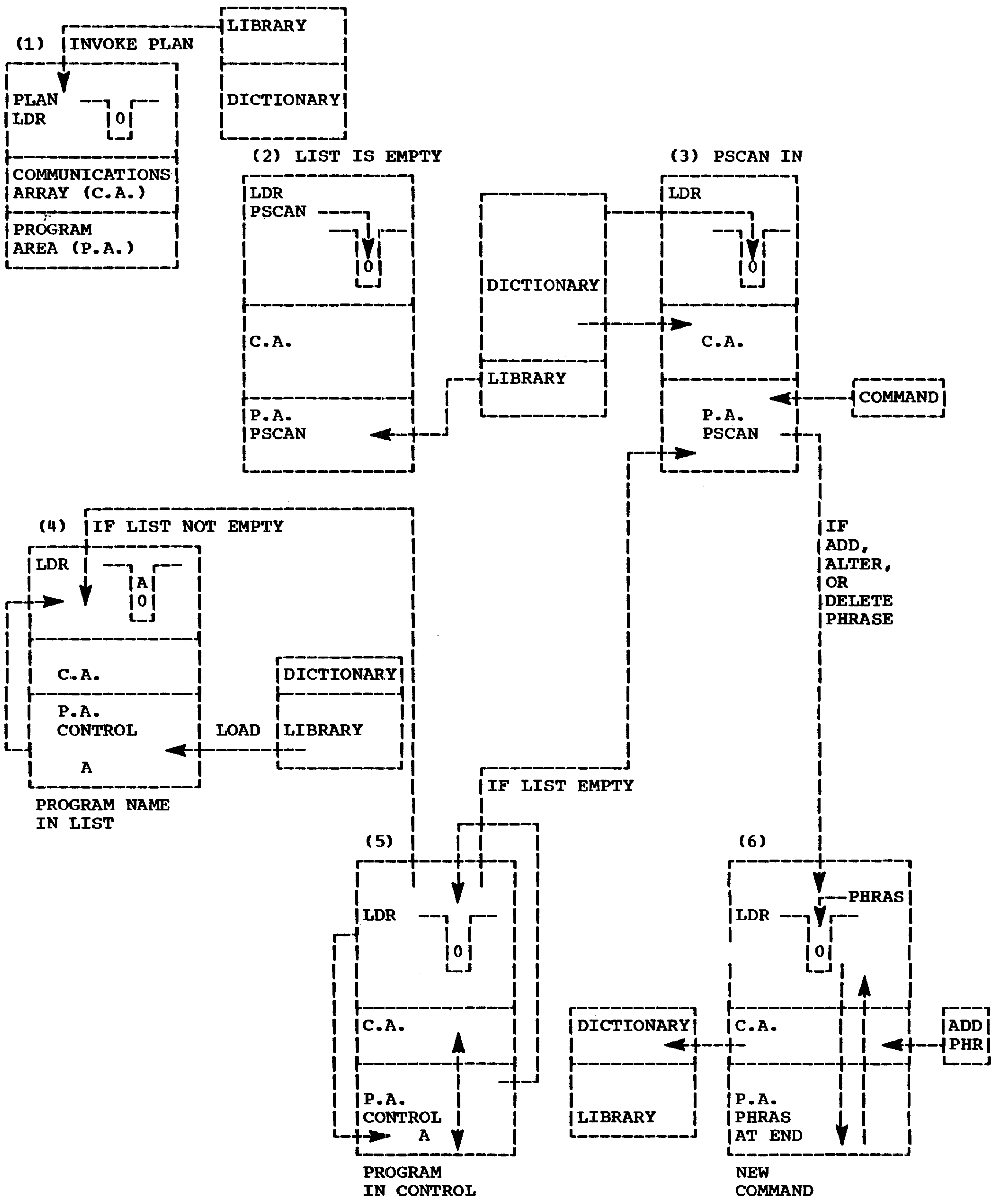


Figure 5. Logic of PLAN control

2.50.20 PROBLEM-ORIENTED LANGUAGE PROCESSING

Problem-oriented languages have been available since the time of our earliest computing systems. The meaning of problem-oriented languages is that the user communicated a problem's descritpion to a computer in terms normally associated with the problem and built into the input interpreter by the program designer. The terms and data are not necessarily those normally employed by the particular user. In PLAN, we will talk about user-oriented languages instead of problem-oriented languages because at all times the language definition is fully accessible to the user. The terms, symbolic names, and data values may be modified to terms familiar and acceptable to the user. With problem-oriented languages as defined above, changing the vocabulary required programming changes in the language processor.

The PLAN language processor contains only one programmed language statement. The statement, ADD PHRASE, is a bootstrap that allows the user to define a language that is meaningful to him. Just as importantly the language definition is always fully accessible to him for modification as the problem definition requirements change. The concept of a subsystem with a fixed language vocabulary is erased in favor of a system in which the language is made up of statements of a user's choosing and can be dynamically expanded and modified. The language can thus be truly user-oriented.

The users communication of a problem description to the computer with PLAN statements is in the following form:

Statement Name, Data Area;

The user's UOL is comprised of several statements; each added to the PLAN system by means of the ADD PHRASE statement.

The ADD PHRASE may include definitions of:

- Statement name
- Associated program list
- Data dependency level
- Allowable data names
- Default data values
- Data mode indicators
- Data scaling information
- Definition of special conversion processing
- Arithmetic expressions defining values
- Logical expressions defining values
- Logical and arithmetic tests
- Actions if the tests fail
- Logical and arithmetic formulas

Let's now make another pass through the preceding list in another form. The following narrative represents information that a user might provide to the system analyst in defining a user-oriented language statement. Bullets to the left of the narrative are meant to correspond to the sequence of items presented above. The statement definition as it would be developed as a result of the narrative is included in boxes under each narrative section. New entries resulting at each step are underlined.

- I would like to define a language statement "DESIGN SOMETHING"

```
ADD PHRASE: DESIGN SOMETHING,
```

- This statement must result in the computing of ...

```
ADD PHRASE: DESIGN SOMETHING, PROGRAM 'INPUT,CALC',
```

- Data to be defined with the statement will be identified with the names MINIMUM, MAXIMUM, DELTA, XVAR, MODULUS, ANGLE, YVAR, and TEST.

```
ADD PHRASE: DESIGN SOMETHING, PROGRAM 'INPUT,CALC',
MINIMUM, MAXIMUM, DELTA, XVAR, MODULUS, ANGLE, YVAR, TEST,
```

- The most frequently encountered values for the variables are 0, 10E6, 100, 0, 1, 90, 0, and NONE respectively. Test has no predefined value.

```
ADD PHRASE: DESIGN SOMETHING, PROGRAM 'INPUT, CALC', MINIMUM 0,
MAXIMUM 10E6, DELTA 100, XVAR 0, MODULUS 1, ANGLE 90, YVAR 0, TEST,
```

- All variables listed above may take on decimal values

```
No change to above definition
```

- MODULUS should be scaled by a value of 10^6

```
... P+6 MOD 1,...
```

- ANGLE will be a value in degrees and should be converted to radians.

```
... ANGLE 90=ANGLE*.0174532965,...
```

- TEST should be set to LOGICAL TRUE if ANGLE is between zero and 90 degrees; otherwise to LOGICAL FALSE.

```
... TEST:(ANGLE>0) & (ANGLE<1.5707965),
```

- If TEST is FALSE, an alternate program is to be loaded to process the nonfirst quadrant angle. If the minimum value is greater than the maximum value, an error message should be issued and processing terminated.

```
... TEST *T'APROG':(ANGLE>0)&(ANGLE<1.5707965),
*FA'MINIMUM GREATER THAN MAXIMUM':(MINIMUM>MAXIMUM)
```

It should be noted in conclusion of this section that the system analyst would provide additional data beyond that shown as a direct result of the narrative. This material has been added below to make the example complete.

```
ADD PHRASE:DESIGN SOMETHING, PROGRAM
'INPUT, CALC', (M)MINIMUM 0, (M+1)
MAXIMUM 10E6, (M+2)DELTA100, (M+3)XVAR
0, P+6(M+4)MODULUS 1,(M+5)ANGLE90=
ANGLE*.017452965, I(M+6)N=N+1, (M+7)
YVAR 0, (M+8)TEST*TA'APROG':(ANGLE>0) &
(ANGLE<1.5707965), (M+9) X*FA'MINIMUM
GREATER THAN MAXIMUM':
(MINIMUM>MAXIMUM);
```

The user would not be involved the complexities of the command as shown above. His use of the command would resemble that shown below.

```
DESIGN SOMETHING, MAXIMUM1000, XVAR 50,
ANGLE 75;
```

2.50.30 DIAGNOSTIC SUPERVISION

Success of any system is highly dependent upon facilities provided by the system for isolating and indicating user's errors explicitly enough to allow the source of the errors to be corrected. In this sense, an efficient diagnostic supervisor is one of the most important attributes of a system.

The PLAN system is highly diagnostic. In addition to fixed diagnostic text that defines the reason for an error, a diagnostic modifier is provided that gives variable data to assist in pinpointing the error.

The user is allowed several degrees of flexibility in processing diagnostics. The following modes of diagnostic output may be selected:

- IMMEDIATE The diagnostic is printed when detected. The program detecting the error is check-pointed (saved for future restart) to allow the PLAN diagnostic supervisor to generate the message.

- STACKED The diagnostic(s) are printed whenever the PLAN loader determines that the PLAN diagnostic supervisor can be scheduled without a requirement for check-pointing an existing module.

- QUEUED The diagnostic is written in a diagnostic file as indicated by the immediate/stacked selection. The file is subsequently written on the diagnostic device at the appropriate PLAN job break or when the user requests such action. Thus, the user has full facility for preventing the intermingling of diagnostics and normal program output.

The user is also given an interface at modest cost to allow him to interface into the diagnostic supervisor. This interface provides a user the full benefit of IMMEDIATE, STACKED, or QUEUED processing for generating his diagnostics. On the 1130 system, the cost of the IOCS to produce a printed diagnostic if no other printed output is required by the module is approximately 1200-1500 words depending on format and devices. The core required to interface to the PLAN diagnostic supervisor is approximately 500 words. Corresponding benefits are obtained under DOS/360 and OS/360.

The third aspect of diagnostic processing of importance to the user is found in the error output processing. Many users have special output processing requirements. Format changes may be desired or a device not supported as the PLAN diagnostic device (for example, a 2260) may be the most appropriate output unit. Therefore, the user is provided an error option and data interface that allows him to execute his own error output processing module.

In conclusion, all of the options we have outlined for processing system and user diagnostics are available to the user for dynamic selection. He is not forced to made a decision on mode (except where he must provide his own code for output processing) at compile time or even at job schedule time if he does not want to. Such decisions may be made dynamically.

2.50.40 INPUT/OUTPUT CONTROL

Input/output is a crucial element of virtually every data processing job, even those requiring complex mathematical solutions. It is in this area that the universal languages such as FORTRAN tend to differ among operating systems and machines and where widely varying degrees of flexibility and dynamic availability are found between different programming systems.

The language processing function of PLAN significantly reduces the input processing associated with problem definition. The diagnostic supervisor normally handles the output processing required for diagnostics. Even then, bulk data must be read, results must be transmitted to appropriate output devices, and intermediate data must be read from and written in files. To fulfill these functions, the following three subroutine sets have been provided in PLAN:

- Sequential I/O Control
- DYNAMIC File Control
- PERMANENT File Control

The sequential I/O routines process the unit record (readers, printers, punches, tapes, disks, etc.) in a continuous manner without the ability of random access. The primary objectives of this package are:

- To provide cross-system I/O compatibility
- To provide dynamic device selection
- To provide dynamic formatting selection
- To provide buffered and overlapped I/O
- To provide record reread
- To provide modular I/O programming

The above points can be better understood by examination of the following description of requirements and of present modes of operation.

- It is essential that device codes and unit control information be required in a form that does not require reprogramming, relinkediting, or recore-imaging when shifting configurations.

- The need for dynamic device selection can be understood by examining a problem commonly encountered in the IBM 1130 FORTRAN environment. Assume a user wishes to be able to switch printed output between the 1132 Printer and 1403 Printer and in cases of system malfunction switch the output to the console typewriter. There exists three alternatives, none of which are totally acceptable. (1) Programs can be recompiled with a new *IOCS statement each time a device change is required. (2) The device codes can be set in COMMON at execution initialization to select the appropriate device routine from the three that were compiled. This is obviously wasteful of core that typically is an already scarce commodity whenever I/O is required. (3) The *EQUAT function can be used at core-image time to substitute devices. Severe short comings are soon noted here, especially when trying to substitute a 1442-5 Punch and 2501 Reader for a 1442-6 Reader/Punch. A punch cannot be substituted for a printer even though only 80 characters of output are required. The programmer should not make decisions relative to the assignment of devices to functions. This function should be left to the discretion of the user or operator to allow for DYNAMIC adjustment to conditions.

- This is required in order to read a card in FORTRAN and format it as a master record if there is an x-punch in column 80 and as a detail record if there is no x-punch.
- The buffering, overlapping, and treatment of such special indicators as logical end-of-file should be treated uniformly on all systems and should be available to all users.
- The case of the x-punch as defined above is a special case of required (partial) reread.
- Only I/O and conversion routines required should be included with the user modules.

The DYNAMIC file I/O routines communicate only with direct access I/O devices. It allows a user to define a logical drive (working storage on an 1130 drive or a Data Set on System/360) which is then dynamically allocated and deallocated to as many as 255 LOGICAL files. There may be up to 5 LOGICAL drives on the 1130 and up to 8 on System/360.

Space for a LOGICAL file is allocated only when the program logic defines a requirement. Space is incrementally allocated. As a result, the disk space used by an application is only slightly greater than the sum of the data requirements as contrasted to the sum of the maximum potentially required by each file under conventional systems. In addition, great flexibility is provided to applications since no arbitrary constraint need be applied at compile time by a programmer estimating the maximum file size required.

Access to the file is totally random. Files are not addressed by any normal form of disk addressing. Rather, they are addressed by logical file number, by a relative displacement within the file, and by the size of the block to be transferred to/from the file. Thus, treatment of data is much like asking for a block of words from an out of core array. An attempt to read from outside the true file size causes the file to be closed. A write outside the current true file size causes an additional allocation of space. It should be noted that in addition to removing a compile time requirement for file space allocation, the requirement for record size definition is eliminated. The sequence and block size used for writing a file has no bearing on what may or may not be read.

It's worthwhile noting at this point that the 1130 Version of this package is implemented in such a way that access to the disk is made only when destruction of the buffered data is imminent instead of each time the user issues a write. This can eliminate several disk accesses and therefore substantially improve performance.

An efficient logical, fixed-point, floating-point, or alphameric (in any combination) sort/merge in mixed ascending/descending sequence is privided for the DYNAMIC files defined above. The sort/merge is entered by a simple CALL from a user's module. It is basically an in-place sort (totally in-place on the 1130) so the disk requirement is minimized. Since the sort/merge is callable, a file can conveniently be sorted into numerous required sequences all within a single user's module. In System/360, this precludes several entries to a job scheduler.

The PERMANENT file I/O routines are analogous to the DYNAMIC file I/O routines except that no allocation is provided. The allocations and files are defined outside of PLAN but are as fully discretely addressable as are DYNAMIC files.

All of the I/O support discussed in this section has the single basic objective of providing the user with efficient, compatible, I/O processing where options inefficient or impossible to define at compile time are given at execution time (dynamically).

2.50.50 UTILITY FUNCTION CONTROL

Every system has a block called "everything else" when an attempt is made to develop a system organization chart. PLAN is called "The Application Programmer's Tool Bag". The "everything else" box in PLAN is an assortment of tools that can be generally defined as utility functions. Many functions required when developing an application in the PLAN environment are common to many applications. These have been included in PLAN because they decrease the time and cost and increase the ease of producing application solutions.

These functions fall into the five general categories of:

- Development Aids
- Sub-word Manipulation

- Array and Table Maintenance
- Data Conversion
- Logical Value Testing and Assignment

The ability to examine data is a requirement in the testing and execution of any system. PLAN provides the facility for dumping from internal data arrays and DYNAMIC or PERMANENT files and for dumping his current user-oriented language statements.

Significant alphameric processing can be accomplished with high level languages if they have the ability to extract, mask in, or test at the character level. Similarly, program enhancement, performance improvements, and core savings can result from an ability to test, set, and extract bits and to execute test under mask operations. PLAN provides these functions to the PLAN user.

PLAN provides an efficient means for transferring strings or arrays of in-core data. Often, there is a large amount of alphameric and tabular data required in many applications. For example, many systems will have an array of diagnostic messages that may be given. Maintenance of this data in core memory requires an unjustifiable overhead. Thus, there develops a requirement for a maintenance and retrieval system for such data. The PLAN support in this area provides purely random processing of such data. This support utilizes the PERMANENT file routines for disk access. The user, by following a few simple conventions, can use these subroutines to maintain any data that must be accessed by arrays.

The PLAN system in its user-oriented language facility provides extensive numeric and alphameric data control. In addition, PLAN provides extensive LOGICAL value testing and evaluation. Because support for LOGICAL values is not universally available to users with all programming systems, PLAN provides the necessary LOGICAL processing subroutines.

This system overview has not attempted to delve into the intracacies of PLAN use and function. Neither has it covered all of the facilities provided in PLAN. Careful understanding of the conceptual why, what, and how presented here will allow intelligent decisions to be made relative to what additional segments of this manual must be read and what portion of PLAN will be used.

3.0.0 GENERAL DESCRIPTION OF PLAN MODULES

This section defines PLAN program components. It provides a general description of the functions and purposes of the program modules that are required to make the PLAN function operate.

No PLAN module, other than PLAN, should have execution initiated in a manner other than by appearing as a program name in the PLAN pop-up list. Any attempt to do so will result in program failure. All of these module names are prefixed with DFJ on DOS and OS PLAN.

3.1.0 RESIDENT PLAN SYSTEM

PLAN PLAN is the one "mainline" program for an entire series of modules executed to solve a problem or series of problems. PLAN executes all program loading functions and is therefore referred to as the "loader". It is the program that handles initialization for a PLAN execution and remains in control, either directly or indirectly, until control is returned to the monitor or the operating system for a non-PLAN job.

It sets up the PLAN system Switch Words and collects the information necessary to communicate with other PLAN systems modules. If the language dictionary (PFILE) has not been initialized, it creates the necessary tables and calls in PHRAS to add the command ADD PHRASE.

If the language definition file is defined and has been initialized, PLAN execution is started. If the command analyzer (PSCAN) or the error-processing module (PERRS) is not in the library, PLAN execution is inhibited and control returns to monitor or the operating system.

3.2.0 COMMAND ANALYZER

PSCAN This module of PLAN is the command collector and analyzer. The command is read from the command input device. PSCAN saves and restores the managed communication array in the managed array save file as required by the phrase level definition. It initializes the MANAGED communication array to logical FALSE each time a level 1 command is processed.

PSCAN collects input data values and stores them in the communication array. The programs defined by the command definition are added to the pop-up list. Command-defined expressions are evaluated. Checking of any required values is performed and if there are no errors, control is returned to PLAN (loader) to load and execute the first program named in the pop-up list. If errors are detected, the system error routine (PERRS) is called to generate the appropriate diagnostics.

PSCAN is not directly specified by the user for loading. It is loaded by PLAN, whenever no program names appear in the pop-up list.

3.3.0 LANGUAGE DEFINITION ANALYZER

PHRAS This module deletes from or adds to a language dictionary (PFILE), problem language command definitions. Standard values are converted to the appropriate mode (floating-point or fixed-point) and program name lists are stored. Extensive logical and syntax verification is performed before each phrase is stored. The system error module (PERRS) is called to log any required diagnostics. In cases where core size limitations prevail (as on the 8K 1130 System), PHUDT, representing the second half of PHRAS, is loaded as an overlay. PHRAS is loaded by PLAN in an identical manner with any user module. It is specified in the program list of the ADD PHRASE, ALTER PHRASE, and DELETE PHRASE commands.

3.4.0 SYSTEM ERROR PROCESSOR

PERRS This module is the system error processing module and is loaded to produce an appropriate diagnostic whenever control is returned to the loader and errors have been detected or as required by user definition of error processing.

PERRS is loaded by PLAN as a result of any call to the error subroutines (ERROR, ERREX, ERRET, ERRAT, ERLST).

3.5.0 PLAN UTILITIES

GMRGA, GMRGB	(OS/DOS only) These modules perform the merging of PLAN PERMANENT files. Their names are placed in the pop-up list as a result of a call to GMERG.
GSRTA, GSRTB	(OS/DOS only) These modules perform the sorting of PLAN PERMANENT files. Their names are placed in the pop-up list as a result of a call to GSORT.
PCDMP	This module provides a dump of the PLAN Switch Words and communication array as specified by the system Switch Words. This module is executed as a result of processing a DUMP COMMON, DUMP SWITCHES, DUMP MANAGED, or DUMP NONMANAGED command.
PDIAG	This module maintains user-specified literal strings in a disk file. This module is executed as a result of processing a SET LITERAL command.
PEDMP	This module produces a dump of the error messages recorded in the error message queue file. This module is executed as a result of processing a DUMP ERRORS command.
PFDMP	This module provides a dump of PLAN DYNAMIC files and PLAN PERMANENT files. It is executed as a result of processing a DUMP DYNAMIC command or the DUMP PERMANENT command.
PFIND	(1130 only) This module is the initialization processor for the DYNAMIC file find, release, and automatic release functions. Pack initialization and drive verification functions are performed where required.
PIDMP	This module provides a dump of the phrase currently being executed. This module is executed only if user action causes its name to appear in the pop-up list.
PIOCS	This module uses the PLAN subroutine IOCS to change PLAN system parameters through the use of commands. It allows the user to switch command input and PLAN output to new devices in the middle of a job stream.
PLENG	(OS/DOS only) This module varies the number of printed lines per page on an output device. The module is executed as a result of processing a SET PAGE LENGTH command.
PLITL	This module produces a listing of all literals maintained in a PLAN literal file by the module PDIAG. The LIST LITERAL command utilizes this module.
PMRGA	This module performs the merging of PLAN DYNAMIC files. Its name is placed in the pop-up list as a result of a call to PMERG.
PSRTA, PSRTB	These modules perform the sorting of PLAN DYNAMIC files. Their names are placed in the pop-up list as a result of a call to PSORT.
PSTSV	This module saves statements when required. It is called in by PSCAN or PLAN (the loader) whenever a statement is to be saved for subsequent execution or the SAVE or EXECUTE command is processed.
PTDMP	This module produces a listing in a tabulated format of the phrases defined (added by PHRAS) in the PLAN language definition file (PFILE). It is executed as a result of processing a DUMP PHRASES command.

4.0.0 DETAIL DESCRIPTION OF PLAN INPUT

The following sections define in detail the features, options, and restrictions associated with the use of the PLAN user-oriented languages. The section is broken into the nine segments listed below:

1. PLAN Language Terminology
   This section describes the terminology that is used throughout this manual in description of the PLAN system. It should be read and understood before attempting to study subsequent sections. The Glossary of this manual may also be helpful in attaining this understanding.

2. PLAN Language Use
   This section describes the use of a language defined under PLAN.

3. PLAN Language Definition
   This section defines the rules for writing a problem-oriented language under PLAN and describes the attainable capabilities.

4. Language Definition Tutorial
   This section provides assistance to the system analyst-designer in the form of a question-and-answer tutorial on the generation of language definition statements.

5. Standard PLAN Commands
   This section describes the commands that are released with every PLAN system because of their general utility.

6. PLAN Subroutine Support
   This section provides a brief description of subroutines available within PLAN.

7. PLAN Subroutine Use
   This section provides a description of the PLAN loader subroutines, PLAN dynamic file support, PLAN error processing subroutines, PLAN permanent file support, PLAN sequential file support, and general utility routines. Calling sequences and examples are provided.

8. Programming Conventions
   This section describes the conventions that a program must respect in order to run as a module under PLAN.

9. PLAN System Case Study
   This section takes the statement of a simple problem and carries the logic of solution under PLAN through two levels of sophistication.

The sections above describe the details of the PLAN system as generally applicable to all implementations of PLAN. Details relating to restrictions of a specific implementation or a specific option are listed in the appendices of this manual as listed below:

1. Appendix A: 1130 PLAN Specifications
2. Appendix B: System/360 DOS PLAN Specifications
3. Appendix C: System/360 OS PLAN Specifications
4. Appendix D: Syntactical Organization of the PLAN Language
5. Appendix E: PLAN Systems File Layout
6. Appendix F: PLAN System Diagnostic Messages
7. Appendix G: Compatibility Considerations
8. Appendix H: Summary of System Limits
9. Appendix I: PLAN Character Set
10. Appendix J: System Requirements
11. Appendix K: Functional Analysis Diagrams
12. Appendix L: Communication Array Layout Chart
13. Glossary

4.1.0 PLAN LANGUAGE TERMINOLOGY

This section provides a general introduction to the terminology associated with a PLAN user-oriented language.

Defining a user-oriented language statement (command) is a one-time operation (except, of course, when the command requires modification). This command definition is cataloged into a language definition dictionary (PFILE) that is maintained on a direct access device. This definition may then be immediately used to effect a problem's solution.

Control of the problem solution and communication between PLAN (command analysis) and the application logic module is provided by the pop-up list and the communication array. The pop-up list is used to stack the list of logic module names for execution. The communication array allows for storage of arithmetic, logical, and literal data for transmission between PLAN and the system programmer's logic modules.

In the discussions that follow, definitions of terms will be presented for the terminology associated with PLAN. A thorough understanding of these elements is required in the implementation of valid and effective PLAN statements.

4.1.1 WORD

A word is a sequence of one or more alphabetic characters, without embedded blanks. Only the first three characters of the word are considered significant. Words of less than three characters are considered to be padded with blanks. Examples of valid forms of words are given below:

A, ABLE, ARROW, ARRAY

Effective form (b indicates blank padding) of these words is:

Abb, ABL, ARR, ARR

Note that PLAN treats the last two words as being identical.

4.1.2 PHRASE

A phrase is a fixed sequence of one to five words separated by blanks. The user-oriented problem description language is built from phrases, data, and procedural information associated with each phrase. The following are examples of valid phrases:

```
DESIGN TORSION SPRING
GRAPHIC REPORT GENERATOR
EVALUATE STEEL FRAME BUILDING
GRAPH
```

4.1.3 OBJECT PHRASE AND VERB PHRASE

An OBJECT phrase is an independent phrase defined at ADD PHRASE time. A VERB phrase is a dependent phrase (defined at ADD PHRASE time) that is used as a prefix modifier to an OBJECT phrase and may not be used alone as a phrase. The first part of any phrase may not be a VERB phrase. If "LITTLE RED WAGON" is defined as an OBJECT phrase, it prohibits "LITTLE", or "LITTLE RED" from being defined as VERB phrases. "LITTLE RED WAGON" may, however, be both a VERB and an OBJECT (nonverb) phrase. Its syntax determines its use. If it stood alone, "LITTLE RED WAGON" would be interrogated as an OBJECT (nonverb) phrase. However, "LITTLE RED WAGON TRAIN" would result in "LITTLE RED WAGON" being interpreted as a verb phrase. A more detailed explanation of VERB phrases and their use can be found in section 4.3.5. The following are examples of VERB phrases:

```
DEFINE
REPEAT EXECUTION
ADD
EXPLAIN
ALTER
DELETE
```

4.1.4 COMMAND

A command is a sequence of one or more phrases that implies a task. All but the last (rightmost) phrase of a command must be VERB phrases. A command always contains an OBJECT (nonverb) phrase and may contain up to eight VERB phrases. The first of the command examples shown below is an OBJECT phrase; the second contains a VERB and OBJECT phrase using phrases given in the above examples:

```
DESIGN TORSION SPRING
EXPLAIN DESIGN TORSION SPRING
```

4.1.5 STATEMENT

A statement is a command, optionally followed by data. It may contain a maximum of 450 characters. It must be terminated with a semicolon (;). A PLAN input record is 80 characters in length. The statement is entered in record positions 1-75. Continuation of text is automatic from position 75 of one record to position one of the following record. A new statement may start immediately following the terminating semicolon of the previous command. A record containing 80 blank characters is ignored.

PLAN, upon processing a statement, loads logic modules that are associated with the phrases making up the statement. Thus, the sequence of statements implies the series of module executions that must be executed to complete a problem solution process.

If a statement has a blank command the preceding command is implied. Note carefully that only the object portion (non-verb) of the phrase is repeated. Example:

```
SCALE, SN1, LOS5;, SN2, LOS6; LABEL,
LTX'ABC';
```

The command SCALE utilizing the numeric data values 1 for data name SN and 5 for data name LOS is executed first, followed by the second execution of SCALE (implied) which will use the values of 2 and 6 for SN and LOS respectively. LABEL will then be executed last using the literal data value 'ABC' for data name LTX.

The comma is always required to separate the command and the data even when the command is blank. If there is no data, the only punctuation required is the statement-terminating semicolon. Example:

```
SCALE;
```

4.1.6 DATA

Data is the set of symbols and values following the command in a statement. Data may be identified by name, and is logical, arithmetic, or literal. If data is unnamed, the value is stored in the position immediately following the previously stored data value. Data values follow data names and are not separated from the data names by delimiters (except optionally by blanks).

A data element definition is organized as follows:

F	S	N	V

F	contains the format control, (user-exit control, mode control, and scale factor)
S	contains the communication array subscript (CAP)
N	contains the data name
V	contains the initialization values, check entries, and phrase-defined expressions

Examples of the three data types are:

DEF 7, SN3	Arithmetic
WORD'X', LTX'ABC'	Literal
HAVE-, SWITCH+	Logical

Data values are normally floating-point binary, but fixed-point binary or EBCDIC literals may be specified. Two values are reserved for logical value representation. They are:

Logical FALSE = 7FFFFFFF (hexadecimal)

Logical TRUE = 80000000 (hexadecimal)

4.1.7 DATA NAME (DAN)

A data name is a word that symbolizes the value of a particular storage location. If defined at phrase-definition time, it may be singly subscripted at phrase-execution time to reference a logically related value. All data names are assumed to refer to the first position of an array. Therefore, using data name "XYZ" is the equivalent of saying XYZ(1), where the subscript 1 is assumed. In the example SCALE, (20)SN 2,5,7,9, the value 9 may be referenced as SN(4).

The single character E may not be used as a data name because of possible conflict with E notation in numbers.

Care should be exercised to avoid the selection of data names containing the first three characters identical to those in other data names.

4.1.8 CONSTANT (NUV)

A constant is a signed or unsigned fixed-point (integer) or floating-point (real) decimal number. Constants may contain exponential modifiers but may not contain embedded blanks. Examples of valid constants are:

1, 1., -1, +2.5, 3.14E-1

Two special logical constants are recognized:

+ (logical TRUE)
- (logical FALSE)

4.1.9 LITERALS (SLV)

Literals consist of any alphameric data (except a semicolon) bounded in the input stream by single quotation marks (or the @ sign) or by double quotation marks. The double quote symbol is a unique EBCDIC character with no implication of two single quotes. Examples:

```
'ABC'
"1234"
@69LL@
```

The number of words in storage occupied by a particular literal is determined by applying the following formula, where M is the number of literal characters and J is the maximum number of literal characters that may be stored in one ASA floating-point word:

```
1 + (M+J-1) / J     (for single quotes)

(M+J-1) / J         (for double quotes)
```

There are two ways of storing literal data. The particular method employed is determined by the type of boundaries set around the data. If single quote marks (or the @ sign) are used, the total of the number of characters making up the literal is stored in the first word of the array (position indicated by the data name or CAP index, see 4.3.6). When double quotes bound the data, the character count is omitted. In the following examples, assume a computer system in which four characters can be stored per 32-bit word.

```
'ABCDEFGHI'
```

The above literal would be stored in four 32-bit words:

```
|9|ABCD|EFGH|Ibbb|
```

(bbb represents blank padding)

Should the example be written "ABCDEFGHI" the literal would be stored in three 32-bit words:

```
|ABCD|EFGH|Ibbb|
```

Literals bounded with single quotes (or @ signs) are hereafter called PLAN literals. The quote mark ending a literal must be identical to the quote mark beginning a literal. Any other quote mark is assumed to be a literal text character.

4.1.10 ARITHMETIC OPERANDS (AOP)

An arithmetic operand consists of terms and operators. The terms may be data names or constants. The operators are +, -, *, /, in their usual sense. Parentheses are used to show the order of evaluation. The hierarchy of evaluation is * and / followed by + and -. Processing order is left-to-right. Mixed mode terms are allowed in an arithmetic operand. Evaluation of an arithmetic operand is done in floating-point mode (and rounded before storing if required because of truncation).

```
ARITHMETIC OPERAND   (aop)
-----------------------------------------
{←dan} {←nuv}
{ } indicates that enclosed items may
    be entered more than once
  ← arithmetic operator
dan data name
nuv constant
-----------------------------------------

  B+3-(C+2*D)
```

In special cases an arithmetic operand may be a literal or a logical constant. In these cases the operand may contain only the literal or logical constant.

```
SPECIAL ARITHMETIC OPERAND   (saop)
-----------------------------------------
  "SLV"
  'SLV'
  @SLV@
   +
   -
    SLV literal
    +   designation for TRUE
    -   designation for FALSE
-----------------------------------------
  "LITERAL DATA"
  'LITERAL PLUS CHAR. CT'
```

4.1.11 ARITHMETIC EXPRESSIONS (AEX)

An arithmetic expression is introduced with an equal sign (=) and consists of an arithmetic operand. An arithmetic expression implies the storing of data. Arithmetic expressions are evaluated in floating-point mode. The result is stored in the mode indicated for the data name associated with the CAP in which the result is to be stored. Results are rounded if the storage mode is fixed-point. If any operand of an arithmetic expression has the value of logical TRUE or logical FALSE, the result of the expression evaluation is FALSE.

ARITHMETIC EXPRESSION (aex)
= aop {←aop} ← arithmetic operator {} indicates that enclosed items may be entered more than once
= 5*A-3*(B-12)

The following example illustrates use of a phrase definition to convert input data values from degrees (A) to radians (B). If no value is given at execution time for A, B will be set to a logical FALSE because of the standard value of FALSE in A. Example:

```
...A-, ...  B=A*.017453296,...
```

If a special arithmetic operand (literal or logical constant) is used in an arithmetic expression, it may be the only operand in the expression. The following example shows the use of literals and logical constants in arithmetic expressions:

```
A=+ (A is True)
A=- (A is False)
A="B" (A is Bbbb)
A=@B@ (A is 1,Bbbb)
A='B' (A is 1,Bbbb)
```

SPECIAL ARITHMETIC EXPRESSIONS (saex)
= saop
= "STANDARD DATA" = 'STANDARD DATA' = @STANDARD DATA@ = + = -

4.1.12 LOGICAL OPERAND (LOP)

A logical operand may be a data name or a relational operation.

A relational operation consists of one of the relational operators "greater than" (>), "less than" (<), or "equal to" (=) preceded and followed by an arithmetic operand. A logical value of TRUE or FALSE in either arithmetic operand forces the result of the evaluation of a relational operation to be FALSE. A relational operand must be enclosed in parenthesis.

LOGICAL OPERAND (lop)	
dan (aop ↑ aop) (dan="slv") (dan=+) (dan=-)	
dan	data name
aop	arithmetic operand
↑	relational operator
slv	literal value
+	designation for logical TRUE
-	designation for logical FALSE
A (B+5>A-D) (C="ABC") (D=+) (F=-)	

A special relational logical operand is provided for testing literals. The relational operator is an equal sign, preceded by a single data name that may be subscripted, followed by a test mask. The double quote marks (card code 7-8) enclose the test mask. The underline character (card code 0-8-5) is used to indicate characters which are not to be tested. The following example illustrates a logical operand to test the first character of the literal stored at DATA for a P in the first position, an N in the third character, and an L in the fifth character. The literal mask may contain any number of characters.

```
(DATA = "P_N_L"),
```

Note that testing includes only the number of characters in the mask, so "P_N_" is equivalent to "P_N".

A logical operand may also test for a logical TRUE (DATA=+) or a logical FALSE (DATA=-).

All characters in a logical operand must be entered in the EBCDIC code. Logical operands are tested for FALSE (7FFFFFFF Hexadecimal). If they are not FALSE, they are treated as TRUE. Therefore, any numeric value occurring in a logical value is treated as TRUE.

4.1.13 LOGICAL EXPRESSION (LEX)

A logical expression contains logical operands and operators. Logical operands have a value of either TRUE or FALSE, while logical operators are defined as "AND" (&), "OR" (|), and "NOT" (¬). All of the characters must be of the EBCDIC character set. The TRUE or FALSE result of the evaluation of a logical expression is obtained from evaluation in the order "NOT", "OR", "AND". Parentheses are employed to indicate the sequence of evaluation and also to enclose subscripts. The examples below illustrate the above points:

(B>1) | (B<0) is a logical expression that produces a result of TRUE if B is greater than 1, or less than 0.

A & B & ¬ C is a logical expression that produces a result of TRUE if A and B are TRUE and C is FALSE. The following example illustrates a logical expression defined to test data for logical TRUE, logical FALSE, or a numeric value greater than 10,000. Example:

(DATA=+) | (DATA=-) | (DATA>1E4)

```
LOGICAL EXPRESSION (lex)
----------------------------------------
  :lop{•lop}
    lop logical operand
     •  logical operator
    {}  indicates that enclosed items may
        be defined more than once
----------------------------------------
  :A|B&(C>10)|(D="XYZ")&¬(X=+)
```

The following example illustrates the logical command structure, order of evaluation, and results obtained. In the example, "OPEN" represents a data item containing a logical TRUE and "CLOSE" represents a data item containing a logical FALSE:

A:¬¬OPEN & ¬OPEN|¬CLOSE & OPEN, produces the result of TRUE in A.

LOGICAL COMMAND STRUCTURE	A: $\underline{\text{¬¬TRUE}}$ & $\underline{\text{¬TRUE}}$ | $\underline{\text{¬FALSE}}$ & TRUE
	A: $\underline{\text{¬FALSE}}$ & FALSE | TRUE & TRUE
NOTS (¬) EVALUATED	A: $\underline{\text{TRUE \& FALSE}}$ | $\underline{\text{TRUE \& TRUE}}$
ANDS (&) EVALUATED	A: $\underline{\text{FALSE | TRUE}}$
OR (|) EVALUATED	A: TRUE

4.2.0 PLAN LANGUAGE USE

This section explains the use of a user-oriented PLAN language.

Languages, defined under PLAN, may be used immediately after the statements have been added to the language dictionary. The following descriptions and examples illustrate the requirements for using a language. The general format of a PLAN-defined language statement is:

```
    COMMAND, DATA;
```

The command is an object phrase (previously defined by ADD PHRASE) and from zero to eight prefixed verb phrases. The command is always terminated with a comma, except when the DATA section is not present. In this case, the semicolon is the only terminator. Examples of valid commands are given below:

```
    THIS VERB PHRASE MODIFIES THIS OBJECT
    PHRASE,...;

    GRAPHIC REPORT GENERATOR;

    MACRO,...;

    FORTRAN PROGRAM,...;

    DESIGN TORSION SPRING,...;
```

Omission of the command signifies that the previous command is to be used. However, the terminator must be present. Note carefully that the verb portion of the phrase is not repeated. In the first example above, if THIS VERB PHRASE MODIFIES is assumed to be defined as a verb phrase, processing THIS VERB PHRASE MODIFIES THIS OBJECT PHRASE;,; would effectively result in the following execution:

```
    THIS VERB PHRASE MODIFIES THIS OBJECT
    PHRASE;

    THIS OBJECT PHRASE;
```

The following example specifies three executions of the SCALE command:

```
    SCALE, SN1;, SN2;, SN3;
```

A feature is provided to allow a user to substitute a ditto mark for a word in a command and thus eliminate redundant entry of words in a command.

Use of ditto marks in a command causes PSCAN to pick up a word of the communication array at phrase execution time and place it in the area occupied by the dittos. This can be useful as a shortcut in saving writing time if a series of phrases has a particular word within the phrase that distinguishes that group from any similar but different group.

The first word of the communication array will contain the first three letters of the word in the phrase for which the ditto mark option is to be exercised. The three-character EBCDIC representation of the words to be substituted will be left-justified in the communication array word. The remaining fourth character in the word will always contain a blank. Example:

```
ADD PHRASE:  GEAR, (1)"GEA",...;
GEAR;
```

At phrase execution time CAP1 will look like:

G	E	A	b

Note: The b represents a blank. The nth word of the communication array is assumed to contain the left-justified EBCDIC representation of the word to be substituted for by the nth ditto mark in the command.

In the following example, the input phrase can be shortened to one word, followed by a ditto mark if the letters GEA are in the first communication array position. The ADD PHRASE statement in the example is included solely to illustrate how use of this feature may be tied into language definition. For example:

```
  ADD PHRASE: GEAR, (1)"GEA",...
  GEAR;
  DEFINE ",...;     (Equivalent to DEF GEA,)
  DESIGN ",...;     (Equivalent to DES GEA,)
  ANALYZE ",...;    (Equivalent to ANA GEA,)
  PLOT ",...;       (Equivalent to PLO GEA,)
```

The data section of a PLAN statement is free-form and requires commas and/or blanks only where required for clarity. The semicolon (;) terminates the data section and the statement.

The data section describes and defines data elements that are to be initialized, converted, scaled, evaluated, and stored in the communication array for access by logic modules associated with the command. Commas must not separate information about a single data item, for example, data name and data value. They may be used to separate different data items.

4.2.1 DATA NAME (DAN)

A data name within a command may be any data name defined by the ADD PHRASE for this command or for any preceding command that has been executed and upon which this command is dependent (higher level). Example:

```
    ADD PHRASE:  ..., (30)NO,...
```

The data name N is associated with communication array position 30.

4.2.2 SYMBOL TABLES

A symbol table of data names associated with a phrase is maintained for each of thefour possible levels (statement dependencies) of PLAN phrases (see "Level of Phrase", 4.3.3 for further definition). The symbol table for level 1 is cleared as each level 1 phrase is encountered. As a lower-level phrase is encountered, the symbol table for that level is initialized from the symbol table of the next-higher level. The level 2 symbol table is initialized to the contents of the level 1 symbol table; level 3 to the contents of level 2; and level 4 to the contents of level 3. Each symbol table may contain up to 126 symbols. These symbol tables are constructed at command execution time. The maximum number of symbols that may be contained in any ADD PHRASE is governed by the 255 16-bit words that may be contained in Table 3 of the phrase entry table (see Appendix E, 12.0.0).

Data names from the dictionary entry for the current phrase are added left-to-right to the initialized symbol table. The symbol table is constructed in a wraparound fashion so that if the symbol table overflows (over 126 symbols accumulate), the first symbols in the current symbol table are the first symbols to be destroyed. In constructing the symbol table, identical data names (created by higher-level phrases) are deleted from the table, leaving only the most currently defined value for each symbol. Overflow of the symbol table most commonly happens when processing many blank-level commands. An undefined symbol initiates a search for the symbol in all higher-level symbol tables. If the phrase level is documented to the user, he may use this information to eliminate redundant entry of data.

Data names defined in a higher-level (lower-numbered) phrase upon which a lower-level phrase is dependent may be used in the lower-level phrase. In the following example, the data name A defined in the phrase ABC is used at execution time in the dependent phrase DEF.

```
    ADD PHRASE:  ABC, (5)A, LEVEL1;
    ADD PHRASE:  DEF, LEVEL 2;
    ABC;
    DEF, A10;
```

4.2.3 DATA VALUE (SDV,SLV)

There are three types of data values: numeric, literal, and logical. When they are specifically assigned to a data name, they are positioned to the right of that name and may be separated by blanks from the name, but not by commas or any other punctuation, except as shown below for literals.

Numeric data values are fields (constants), with or without a decimal point, that may be preceded with a sign and/or followed by an exponential indicator. They may not contain embedded blanks. The data name associated with the value has no implications as to the type of mode (real or integer) in which the data value is entered. Examples:

```
    LOOK 717
    SEEK 8.65E-3
    BIG 2
    GEAR +33
```

A literal data value is made up of a literal positioned to the right of a data name. Like numeric data values, they are stored in the word associated with the data name in a left-justified manner. In the examples below assume that four characters can be stored per 32-bit word (b represents a blank character).

```
                      ┌────┬────┐
    MAIN 'AAA'        │   3│AAAb│
                      ├────┼────┤
    TOT "MESSAGE"     │MESS│AGEb│
                      └────┴────┘
```

A logical data value is a value of logical TRUE (+) or logical FALSE (-) that is associated with a particular storage position or data name.

The examples SWITCH+ and TEST- represent logical data values assigned to data names SWITCH and TEST.

A data value in a statement overrides any phrase-defined (ADD PHRASE) initialization value (see "Default Values", 4.3.12).

Use of a data name without including a data value at command execution time sets the location associated with the data name to TRUE, and should be avoided unless provided for by specified language and program module option. The following example would set the location associated with XGS to TRUE:

```
    GRAPH, XGS;
```

Several rules may be followed in reducing the amount of information required to define data. These rules are:

If no data name is given for a data value, succeeding data values are stored in adjacently higher communication array positions (see "Communication Array Position", 4.3.6) in the same mode, converted by the same user exit, and with the same scale factors as defined for the last given data name. In the following example, if XGS is assumed to be defined for communication array position 6, and is to be stored in floating-point, the results of the example would leave a floating-point 11 in array position 6 and a floating-point 8.5 in array position 7.

```
    XGS11,8.5,
```

If the data value given, for which no data name is provided, is the first value following the phrase name, the data name and storage mode assumed is that of the first data name defined for the current phrase and the data value will be stored in the associated CAP. If the first-defined reference is in the PLAN switch words, then the next definition is assumed. If there is no definition, the first position of the communication array is assumed. (Note that full understanding of the following example requires study of the section "PLAN Language Definition", 4.3.0.) Example:

```
    ADD PHRASE:  ONE, (20)A, (30)B...;
      ONE,10;
    Stores 10 in location 20

    ADD PHRASE:  TWO, (-8)ECO, (40)B...;
      TWO, 10;
    Stores 10 in location 40

    ADD PHRASE:  ABC,  (5)XYZ5,  (23)PQR10,
    I(7)2;
    ABC, 5, 6, PQR3;
```

Execution of this example would leave a floating-point 5 in array position 5, a floating-point 6 in array position 6, a floating-point 3 in array position 23 and a fixed-point 2 in array position 7.

4.2.4 EXPRESSIONS

PLAN statements may include expressions at execution time as well as at phrase definition (ADD PHRASE) time (see "PLAN Language Definition" 4.3.0). Expressions are either arithmetic or logical. Arithmetic expressions are introduced with an equal sign (=); logical expressions are introduced with a colon. An expression must be terminated with a comma or the phrase-terminating semicolon. Operands that are not constants must be in the current symbol table at the time the expression is evaluated. Examples:

```
    A5,A(2) = A*.017452965
    B6, C=B*39.37+F
```

The above arithmetic expressions represent entries in execution-time statements. They are valid only if the variable operands, (A,B,F), are in the symbol table when the expression is evaluated.

```
    D:  (A&B) | (C&D)
    K:  (A + 3> F)
    R:  (DATA=+)
```

The above three logical expressions (they are introduced by a colon) may be found in an execution-time statement. In the first example, D will contain a logical value of TRUE (+) if either (A&B) are both TRUE or (C&D) are both TRUE. Otherwise, it will be set to logical FALSE (-). K will be logically TRUE if the value of A+3 is greater than the value of F. However, K will be set to logical FALSE (-) if the value of either F or A is logical (TRUE or FALSE), or if the value of F is equal to or greater than the value of A+3. R will receive a value of logical TRUE if DATA contains a logical TRUE; otherwise, R will be set to logical FALSE (-). The following examples further illustrate the use of the previously mentioned rules:

1. ADD PHRASE: SITE, 13, (6)ABC7, (23)PQR;
   SITE;
   At execution time CAP 1 contains a value of 13.

2. ADD PHRASE: SITE;
   SITE, 13;
   Execution of this example results in a floating-point 13 being placed into CAP 1.

3. ADD PHRASE: SITE, 13, (-6)ABC, (23)PQR6;
   SITE, 5;
   Execution of this example results in a floating-point value of 13 being placed into CAP 1 and a floating-point 5 placed into CAP 23.

4. ADD PHRASE: SITE, 13, (-6)ABC;
   SITE, 5;
   Execution of this example results in a floating-point 5 being placed into CAP 1, overriding the phrase defined 13, because the switch word (-6) is not eligible to receive the value of 5. The encountering of the 5 at execution time generated a search of the symbol table created by the ADD PHRASE statement. If no symbols are present (switch words are excluded), the 5 is placed in CAP 1.

4.2.5 SUBSCRIPTS

Each data name definition is assumed to be the name corresponding with the first position of an array. Thus, using data name "ABC" is the equivalent of saying ABC(1), where the subscript one is assumed.

Any relative communication array position may be referenced by using the appropriate subscript. In the following example, XGS is a data name assigned to communication array position 5, and YGS is a data name assigned to communication array position 6. Each line of the example illustrates the correct means of entering values of XGS and YGS. The storage mode is assumed to be identical for both XGS and YGS. Example:

```
    XGS50,YGS60
    XGS 50,YGS 60
    XGS 50 YGS 60
    XGS 50 XGS(2)60
    XGS 50,60
    XGS 50 60
```

Arrays may be initialized to a single numeric data value at execution time by use of a three-parameter subscript.

The general format of this subscript is:

```
    DAN(I,J,K)V
```

where:

I	is the initial subscript
J	is the last subscript
K	is the increment to the subscript
V	is the initial numeric value to be used

In the following example, assume A has been defined as equivalent to communication array position 72. The example indicates that every other position between communication array 75 and 81 is set to zero.

```
    A(4,10,2)0
```

In the above example, a particular area of storage is to be initialized with a value of zero (0), which is the number outside and to the right of the parentheses. This area is defined using two displacement values (limits) from the reference point A. These specific boundaries are indicated by the first two numbers within the parentheses (4 and 10), with the first representing the lower limit and the second the upper. Since A was assigned to position 72, the positions 75 to 81 (or A(4) to A(10)) signify the designated area. The third and last number within the parentheses (2), is the interval at which the initialization value (0) is to be assigned. Hence, after the example is executed, the positions 75, 77, 79, and 81 will contain the value 0. Care must be taken to ensure that the difference between the middle number and the first number (10-4) is evenly divisible by the third number or a diagnostic will result.

4.2.6 FORMULA NUMBERS

Formula numbers may be assigned to data items defined at execution time (expressions, data assignment, formulas). (See also "Formula Area", 4.3.20.)

Formula numbers are introduced with a dollar sign ($) and are a numeric field in the range of 1 to 32,767. Formula numbers may precede any data item. Formula numbers allow branching and looping within the data section of a statement input. Formula numbers may be assigned to any allowable data item. In addition, the following special type data item entries are associated with formula numbers.

Syntax :$n,
Meaning Formula number n is to be executed next (functions like a FORTRAN GO TO).

Syntax $n;
Meaning Formula number is a dummy formula number to allow transfer to the end of the command; no execution is implied.

Syntax :(expression) ?$n !$m,
Meaning Either the TRUE (?) or FALSE (!) leg (or both) of a conditional expression may be replaced with an expression number. The formula number (n or m) becomes a "branch to" indicator.

The following illustrates use of formula numbers in the control of execution-time data definitions. Example:

```
    TEST, B0A1$1B=B+1, A2:(B=2)?$2, A3:
    $1,$2;
```

would be executed with the following steps:

B0	assignment statement B=0
A1	assignment statement A=1
$1	assignment of formula #1 to the expression B=B+1
B=B+1	after the first execution of formula #1, B will have a value of 1 (1=0+1)
A2	assignment statement A=2
:(B=2)?$2,	this conditional expression will cause a branch to for-

	mula #2 ($2) if B has a value of 2, however; now B has a value of 1.
A3	assignment statement A=3
:$1	go to formula #1 (B=B+1)
B=B+1	formula #1 is executed giving B a value of 2 (2=1+1)
A2	assignment statement A=2
:(B=2)?$2	a branch to formula # 2 will be executed as the condition being tested (B=2) is TRUE.
$2;	formula #2 is a dummy end of this command.

4.3.0 PLAN LANGUAGE DEFINITION

This section explains the procedures for defining a PLAN language statement.

Language definition provides the phrase name, phrase-defined data, and other control information required to direct the execution of a logical segment of a task. A language, once defined (in PFILE by PHRAS) remains permanently defined until altered or deleted by processing an ALTER PHRASE or DELETE PHRASE command. The following discussion treats each possible element of a language statement and its implied effect on the use of and generation of problem solution logic modules. A PLAN problem-oriented language is defined by phrases; the newly loaded system has the capability only of adding phrases through the use of the basic system command ADD PHRASE. As soon as new commands are added, they may be included in the PLAN input stream for execution, as defined in the preceding section. The general format for adding phrases is:

```
    ADD PHRASE:   PHRASE   NAME,   PHRASE-
    DEFINED DATA;
```

A defined phrase may be deleted from the language dictionary by the use of the DELETE PHRASE command. If the phrase to be deleted is a VERB phrase, the specification word VERB must be used. Format of the command is as defined for ADD PHRASE, except that there is no phrase-defined data. Example:

```
    DELETE PHRASE:  THIS PHRASE;
    DELETE PHRASE:  THAT PHRASE, VERB;
```

The ALTER PHRASE command is a combination of a DELETE PHRASE followed by an ADD PHRASE. Partial modification of a phrase in PFILE is not implied. Therefore, the ALTER PHRASE must follow the exact format of the ADD PHRASE. Example:

```
ALTER PHRASE:   PHRASE  NAME,  NEW PHRASE-
DEFINED DATA;
```

In the above example PHRASE NAME specifies both the name of the phrase to be deleted and the name of the new phrase to be added.

If the ALTER PHRASE is used for a nonexistent phrase, a diagnostic will be produced indicating that the phrase to be deleted cannot be found but the phrase to be added will be successfully entered into the language dictionary.

Note that phrase definition commands are always followed with a colon and the phrase is terminated with a semicolon. Any statement encountered by PSCAN that has the command followed by a colon is not processed beyond the colon. If the phrase is interpreted to be ADD PHRASE, ALTER PHRASE, or DELETE PHRASE, the program name PHRAS is placed in the pop-up loader to be executed next. BCD or EBCDIC character coding may be used where multiple coding conventions exist, except where otherwise specified.

4.3.1 PHRASE NAME

Phrase names follow the previously listed rules under "PLAN Language Terminology", 4.1.0. Important points for naming phrases are:

1. They may contain from one to five words.
2. Words are truncated after three alphabetic characters. Care should be exercised in using words of more than three characters to avoid creation of undesired synonyms with other words.
3. Words must contain only alphabetic characters.
4. The phrase name is terminated with a comma (if phrase-defined data does not exist, the comma may be omitted).
5. Words of less than three characters are assumed to contain blank padding to three characters.
6. The same sequence of words used in a VERB phrase name may not be used as the first part of any other phrase name.

4.3.2 PHRASE-DEFINED DATA

The following general categories are specified as phrase-defined data:

1. Level of phrase
2. Program list
3. Verb designation and program list
4. Data elements
   a. Data name
   b. Default values
   c. Scale factors
   d. Mode
   e. Communication array position
   f. Checking rules
   g. Expressions to evaluate
5. Statement-saving specifications
6. User-exit programs
7. Exit
8. Formula area

4.3.3 LEVEL OF PHRASE

The input interpreter has the ability to work with up to four levels of statement dependence. This permits convenient, concise input because data entered at a higher level is made automatically available at a lower level. PLAN also effects correct data context during error recovery and input validation. Execution errors need result only in an abort of that portion of the job affected by the invalid data. Definition of a problem solution may often be defined in logical, indented outline form. Example:

```
I. JOB NAME

   A. Major Activity

      1. Intermediate Activity 1

         a. Detail

         b. Detail

      2. Intermediate Activity 2

         a. Detail
            .
            .
            .

   B. Major Activity
      .
      .
      .

II. JOB NAME
```

The PLAN commands may have a level assignment corresponding to this outline. Statements at level 1 (I, II, etc.) are completely independent of all other statements. Level 2 (A, B, etc.), level 3 (1, 2), and level 4 (a,b,etc.) statements are dependent upon the accumulation of information provided by the preceding statement of each higher (lower-numbered) level.

The PLAN level structure processor saves and forwards data (managed array) from each statement to its logical successors. The logical sanctity of the managed array is preserved by PLAN through saving, and restoring, the proper data context.

Commands, that have no assigned level, are executed at the level of the previously executed command.

An error in a command, at any level, of the severity to demand cessation of processing when operating with a level concept, need cause only skipping of affected commands. Thus, an error at any level results in skipping of commands only until a command of equal or higher level is encountered. The managed data array is then initialized to represent the proper level of data. All blank-level commands following a command in error are skipped except when the error command is blank-level. In that case, only the blank-level command in error is aborted.

A level 0 command must be the first command processed when entering the PLAN system. ADD PHRASE, ALTER PHRASE, DELETE PHRASE, and PLAN JOB are system level (level 0) commands.

A level 0 command causes all system default options to be set. System options (see "Switch Words", 4.3.21) may also be set to the user's specifications. The standard PLAN command "PLAN JOB;" is recommended as a level 0 command. A level 0 command must always be followed by a level 1 command or another level 0 command. A level 0 command may be introduced at any position within a PLAN job stack to reset the system standard options.

The managed array, the size of which is indicated in Switch Word 10, is set to the value of logical FALSE (7FFFFFFF) whenever a level 1 phrase is encountered.

If a value is specified for the keyword LEVEL, it must be 0, 1, 2, 3, or 4. Example:

```
ADD PHRASE: SOME PHRAS, LEVEL1, ...;
```

Level 2, 3, and 4 commands are each dependent on the preceding higher-level (lower-numbered) command.

The level indicator is placed in the phrase dictionary (PFILE) to indicate the logical position of this phrase in a sequence of dependent statements. If no level is assigned, the level is considered to be blank. Blank-level phrases are processed as continuations of the last phrase without forcing a level test. A level test, following the rules indicated below, is forced whenever a phrase is processed that has a level designation. Two basic functions are fulfilled by the level test. If an error occurs in a phrase, recovery can be made at a point in processing where dependent data is not in error, that is, at a phrase of equal or higher level. Secondly, the managed communication array can be saved and restored so that it always includes data of only those phrases upon which this phrase depends. If the managed array length (Switch Word 10) is set to zero, the data management function is bypassed, but error control processing will still be effected.

In all discussions, references to the level of the preceding phrase should be interpreted to mean "the preceding phrase with an explicit level indication". Thus, blank-level phrases are significant only to the degree that they are implicitly assigned the level of the preceding phrase. The following rules govern the saving and restoring of the managed communication array:

1. The communication array is copied onto disk if a level 1, 2, or 3 phrase has just been executed and a phrase of lower level has been accepted from the current input device. The communication array resulting from the execution of a level 4 command is never saved because there are no lower-level commands. In other words, an error in a level 4 command causes the communication array to be restored at the level 3 status. Note that there must be a user-defined file (PDATA on 1130 PLAN and PLMANFIL on System/360 PLAN) if the communication array is to be saved. The appropriate Operations Manual contains the necessary instructions for defining this file.

2. Upon reading a phrase of equal or higher-level as compared with the level of the phrase just executed, the managed communication array is restored from the disk save area which is associated with the level that is one greater than the level just read. Assume we have just finished processing a level 3 command and have just read a new level 3 command. The communication array would be restored to the status as of the last level 2 command. The last level 2 command was the one that had generated data results used by the old level 3 command. It is now possible for the new level 3 command to use the same level 2 data. The reader should be able to see how the PLAN level structure facilitates interactive processing.

3. The data from previous equal or lower-level (higher-numbered) phrases can never be seen in the managed communication array.

4. If the new level is 1 there is no restore from level 0, nor is there ever a save of level 0.

The following example illustrates the rules stated above. Five phrase definitions were entered into PFILE by the ADD PHRASE commands below:

```
ADD PHRASE:  A, LEV 1, I(1)S0,T0,U0,V0,W0;
ADD PHRASE:  B, LEVEL 2;
ADD PHRASE:  C, LEVEL 3;
ADD PHRASE:  D, LEVEL 4;
ADD PHRASE:  E;
```

The status of the communication array is shown as it would exist following the execution of the commands listed in the table. These commands are issued in the order listed. Note: The hyphens shown in the table indicate that when an error occurs, the user cannot rely on the fact that previously stored data is still available. Execution of the first command (A;) causes the first five words of the communication array to be set to zero.

	CONTENTS OF				
COMMAND	S	T	U	V	W
A; (level 1)	0	0	0	0	0
B,S1; (level 2 after level 1)	1	0	0	0	0
C,T2; (level 3 after level 2)	1	2	0	0	0
D,U3; (level 4 after level 3)	1	2	3	0	0
E,V4; (blank level after level 4)	1	2	3	4	0
C,W5; (level 3 after blank (4)	1	0	0	0	5
D,U6; (level 4 after level 3)	1	0	6	0	5
A,W7; (level 1)	0	0	0	0	7
B,U8; (level 2 after level 1)	0	0	8	0	7
C,S3,X5; (X is error)	-	-	-	-	-
D,S5; (in error recovery)	-	-	-	-	-
E,S7; (blank level in error recovery)	-	-	-	-	-
B,S10; (level 2 recovers)	10	0	0	0	7
A; (level 1)	0	0	0	0	0
D,S5; (level 4 after 1)	5	0	0	0	0
B,U2; (level 2 after 4)	0	0	2	0	0

When errors that result in a phrase abort are encountered, error indicators are set to inhibit execution of any phrase until a phrase of equal or higher level is encountered. All phrases are, however, checked for proper syntax, and if errors are found, PLAN diagnostics are generated.

Data names defined in a phrase are available to phrases of lower level. Thus, a data name defined in level 1 may be used by a dependent level 2, level 3, level 4, or blank-level phrase without redefinition. Any new definition in a phrase supersedes an identical data name defined in a higher-level phrase. Therefore, the same data name may be used in every phrase. A new data name table is initialized only when a level 1 phrase is encountered. For a discussion of symbol tables under level control see 4.3.25 and Figure 9.

4.3.4 PROGRAM LIST

A program list may be associated with a particular phrase. This list is added to the pop-up list before the program list associated with check entries. The keyword PROGRAMS introduces a PLAN literal containing the program names to be placed in the pop-up list. The pop-up list is loaded so that the first program named will be executed first. If a program list is not specified for a phrase, the pop-up list is not changed when the phrase is encountered. A new command will be processed immediately and PSCAN will not require reloading.

In the following example, programs entitled "M0730", "M0745", and "M0737" are to be executed in that order when the phrase "NAME" is processed.

```
    ADD   PHR:    NAME,  PRO'M0730,  M0745,
    M0737' ...;
```

Three EBCDIC special characters are also recognized as valid entries in the program list. These EBCDIC characters must be left-justified in a PLAN double-word (64 bits) if they are to be added to the pop-up load list through user programming. If, however, they are to be added to the pop-up load list through use of the specification word PROGRAMS'...', PLAN will ensure that these special characters are inserted correctly. These entries are:

- * indicates a return to a checkpointed program (see "LCHEX" under "Program Linkage Routines", 5.11.1).
- (indicates the start of coexistent program grouping
-) indicates the end of coexistent program grouping

See Appendix B (DOS), "Core Management", 9.7.0 and Appendix C (OS), 10.6.0 for more information on coexistent program grouping. Note that this feature is not functional on 1130 PLAN.

The following example illustrates a portion of a program a user might write to establish the first of the above special characters (*) as a program name in the pop-up load list.

```
      DIMENSION A(2)
      DATA A/'*     ','bbbb'/
      •
      •
      •
      CALL LIST(2,A)
```

"LIST" is called to move the contents of the double-word array to the pop-up list.

The following is a summary of the requirements for program list construction:

1. A program name must begin with an alphabetic character. Subsequent characters in the name may be either alphabetic or numeric. No special characters are allowed within a program name. A program name may be as long as eight characters for System/360 PLAN users and as long as five characters for 1130 PLAN users.

2. The three EBCDIC characters -- asterisk, left parenthesis, and right parenthesis -- are exceptions to the rules stated above. These three characters are considered valid program names.

3. The asterisks, left parenthesis, and right parenthesis need not be delineated with commas.

4. Unmatched right parentheses may be included.

5. Consecutive commas indicate that the program items to the left in this list are to replace all items currently in the list. For example "PRO 'A,B,,'" will eliminate all list entries and place A and B into the list.

The following example illustrates an ADD PHRASE with a valid program list and shows the contents of the pop-up list after the command NAME is executed.

ADD PHRASE: NAME, PRO'A,B* (D,E,F),,'...;

```
---┐     ┌---
   |  A  |
   |  B  |
   |  *  |
   |  (  |
   |  D  |
   |  E  |
   |  F  |
   |  )  |
   |  0  |
   └-----┘
```

The consecutive commas in the program list cause the program names contained within the quote marks to completely replace existing program names in the pop-up list. In the example above, program A is executed first, and its name is removed from the pop-up list. After A is executed, B is loaded into core and executed. Suppose that during B's execution a checkpoint is taken, (B is saved on disk) and two new programs, X and Y, are placed at the top of the pop-up load list.

```
---┐     ┌---
   |  X  |
   |  Y  |          ┌-----┐
   |  *  |          |  B  |
   |  (  |          └-----┘
   |  D  |
   |  E  |
   |  F  |
   |  )  |
   |  0  |
   └-----┘
```

Since X is at the top of the pop-up list, it is loaded into core and executed, and its name is removed from the pop-up list. Y is then similarly treated. The next program name in the pop-up list is *. This special program name causes a return to a checkpointed program. B is reloaded into core and B's execution continues at the next executable instruction after the checkpoint instructions. The * is removed from the pop-up list. When B is completed, the left parenthesis "(" is encountered in the pop-up load list. This special program name signals PLAN that the subsequent program names, until a right parenthesis ")" is encountered in the pop-up list, are to be simultaneously core-resident. PLAN will load D, E, and F into core and remove their names and the left parenthesis "(" and right parenthesis ")" from the pop-up list. When D, E, and F have concluded their processing, a zero will be encountered in the pop-up list. This indicates that the pop-up list is empty and that PLAN must load PSCAN to read the next command and analyze it.

4.3.5 VERB DESIGNATION AND PROGRAM LIST

The specification word VERB is used to define a phrase that is not a complete command. The VERB phrase is used to change or modify the meaning of an OBJECT (non-verb) phrase.

A VERB phrase may have two types of program lists. One is associated with the keyword VERB and the other with the keyword PROGRAM. They will subsequently be referred to as VERB lists and PROGRAM lists.

A command may consist of only one OBJECT phrase but may contain up to eight VERB phrases as prefixes to the OBJECT phrase providing a maximum of 45 words in a command (a phrase is 1-5 words). (See "PLAN Language Terminology", 4.1.0.) A DELETE PHRASE of a VERB phrase must contain the specification word VERB.

The pop-up list at the end of processing a command containing VERB phrases will contain (listed in top-to-bottom order) program lists defined in the phrases in the order listed below:

Check entry programs from leftmost VERB phrase (*C'LIST', 4.3.15)
Program list from leftmost VERB phrase (PROGRAM'A,B,C',4.3.4)
.
.
.
Check entry programs from rightmost VERB phrase (*C'LIST', 4.3.15)
Program list from rightmost VERB phrase (PROGRAM'A,B,C',4.3.4)
Check entry programs from OBJECT phrase (*C'LIST',4.3.15)
Program list from OBJECT phrase (PROGRAM'A,B,C',4.3.4)
Verb designated program list from rightmost verb phrase (VERB'A,B,C', 4.3.5)
.
.
.
Verb program list from leftmost VERB phrase (VERB'A,B,C',4.3.5)
Existing program list (in the case of a CALL LREPT)

The following example illustrates definition of an OBJECT phrase and a VERB phrase. The phrases are then used as a command and the resultant program list is shown:

(OBJECT PHRASE)

```
    ADD PHRASE:  DATA, (5)A-*R'PROGA', +,
    PROGRAMS 'PROGB';
```

(VERB PHRASE)

```
    ADD PHRASE: EXPLAIN, (5)A, B, (255)*
    T'PROGC', VERB'PROGD', PROGRAMS'PROGE';
```

(COMMAND)

```
    EXPLAIN DATA, A, B5;
```

The resultant pop-up list contains the following programs:

PROGC	(in list only if communication array (255) is not TRUE)
PROGE	
PROGA	(in list because data was not given for A)
PROGB	
PROGD	

Additional information on the sequence of execution is given under "PSCAN Execution Sequence", 4.3.25.

The following example illustrates use of a VERB phrase to modify standard data for an OBJECT phrase. Assume an OBJECT phrase (MOTOR VEHICLE DATA) is designed to introduce data for a series of motor vehicle calculations. One of the data items is the minimum driving age (MDA). The country-wide standard is set at 16. Assume the standard for New York to be 18. A VERB phrase (NEW YORK) is defined to modify MOTOR VEHICLE DATA to a minimum driving age of 18. Obviously, this simple example could be extended across many data items with many modification options. Example:

```
    ADD PHRASE:  MOTOR VEHICLE DATA, I(1)
    MDA16, PROG'MVCAL',...;

    ADD PHRASE:     NEW     YORK,     VERB,
    I(1)MDA18,...;
```

If the command "NEW YORK MOTOR VEHICLE DATA;" is executed, the minimum driving age (18) for New York would override the country-wide standard (16) and be used in the motor vehicle calculations by the program 'MVCAL'.

Phrase defined data as defined in the following sections is written in the following format:

U	I	P	S	N	V	E	X

U is the user-exit specification (4.3.18)

I is the mode indicator (4.3.14)

P is the scale factor (4.3.13)

S is the communication array position (4.3.6 - 4.3.10)

N is the data name (4.3.11)

V is the default value (4.3.12)

E is the checking (check entry) specification (4.3.15)

X are the expressions to evaluate (4.3.16 - 4.3.17)

4.3.6 COMMUNICATION ARRAY POSITION

Definition of a data item is not complete unless it includes a definition of where the data item is to be stored. Data items are stored in a COMMON array known as the "communication array" (see "General Description" 2.0.0). A single 32-bit word within this array is referred to as a "communication array position" (CAP). The definition of a CAP is required to provide communication between the input processor (PSCAN) and the system analyst (and programmer). Since the CAP definition represents a displacement relative to the beginning of the communication array, the term subscript is sometimes used interchangeablywith the term CAP. However, in the strictest sense, these terms are different from each other.

The CAP may be defined in any of four different ways. It may be defined as a CONSTANT, an IMPLIED DO, an ARITHMETIC OPERAND or the combination of an ARITHMETIC OPERAND and an IMPLIED DO.

4.3.7 CAP DEFINED AS A CONSTANT

A CAP defined as a constant takes the form (n), where n is an integer constant in the range 1 to 16,368. The CAP has a limit of 8176 if a data name is associated with the CAP. For example, (33) specifies that the 33rd word in the communication array is desired, for some purpose, for an associated data item. The reader will recall that the System Switch words immediately precede the communication array. It is possible, by defining a CAP as a negative integer constant in the range -1 to -15, to reference those COMMON switch indicators (see "Switch Words", 4.3.22).

```
32-BIT
WORDS

┌─────┬─────┐   ┌─────┬─────┬─────┐   ┌─────────┐
│     │     │•• │     │     │     │•• │         │
│     │     │   │     │     │     │   │         │
└─────┴─────┘   └─────┴─────┴─────┘   └─────────┘
 (-1) (-2)      (-15) (1)  (2)         (16,368)
 System Switch Words Communication Array
```

4.3.8 CAP DEFINED AS AN IMPLIED DO

The definition of a CAP may designate a range within the communication array. A range plus an increment within the range is indicated in a manner similar to the FORTRAN DO LOOP. The following example illustrates a CAP defined as an Implied Do. Reference to the communication array starts at position 17, with subsequent references at every third word (20, 23, 26...), ending at position 38.

(17,38,3)

The general form of the CAP defined as an implied DO is (I_1, I_2, I_3), where:

I_1 is the first referenced communication array position,
I_2 is the last referenced communication array position, and
I_3 is the increment used to reference subsequent CAPs within the range specified by I_1 and I_2.

Three rules must not be violated when defining a CAP as an Implied Do:

(1) I_2 must not be negative.
(2) The range divided by the increment must equal a whole number.
(3) A single-valued logical or numeric constant (nuv,+,-) must follow definition of a CAP defined as an implied DO.

Failure to comply with these three rules will cause a PLAN system diagnostic to be issued, and the phrase in question will not be added to the dictionary (PFILE). The reader may wish to verify that our example (17, 38, 3) does not violate our three rules, and as such is acceptable as a CAP defined as an Implied Do. Since I_1 and I_2 represent displacements relative to the same core location (the beginning of the communication array), the range can be determined by merely subtracting I_1 from I_2. Hence, the range in the above example is 21. The range (21) divided by the increment (3) results in a whole number (7). Thus, rule (2) is satisfied. Rule (1) is satisfied by inspection.

Although I_2 may not be negative, a negative integer is allowed for I_1. Defining I_1 as a negative integer gives the user the ability to reference the System Switch Words as part of the range of the Implied Do. An example of a valid CAP definition which references the System Switch Words and functions as an Implied Do is (-1,22, 3). However, since I_1 and I_2 are not relative to the same core location, a slight problem arises in determining the range of the Do. This problem is resolved by adding 15 to I_2, thus making I_2 relative to the beginning of the System Switch Words. I_1 is then treated as a positive integer. Since (-1) really means Switch Word 1, testing I_1 as a positive integer causes I_1 to become relative to the beginning of the System Switch Words. The reader should verify that the range in the example (-1,22,3) is 36.

One last point worth noting about defining a CAP as an Implied Do is that I_3 (the increment) may be omitted. If such is the case, the increment is assumed to be 1. In the example (3,27) the first referenced CAP is 3, subsequent references are 4, 5, 6, 7, etc., through the last referenced CAP, 27.

4.3.9 CAP DEFINED AS AN ARITHMETIC OPERAND

A CAP may be defined as an Arithmetic Operand. Arithmetic Operands, as described under "PLAN Language Terminology", are composed of data names and constants connected by the operators *, /, + and -. An example of a CAP defined as an Arithmetic Operand is (M + 2 - N). The discussion on "Data Names", 4.3.11, states that a CAP defined as an Arithmetic Operand (symbolic CAP) requires an associated data name. Since the effective value (the actual displacement from the beginning of the communication array) of a symbolic CAP is not determined until execution of the phrase which contains the symbolic CAP, and since all data items require an associated core storage location, the data name becomes the "handle" by which the data item is known. Thus, our example as given is not complete and must be rewritten to read, for example, (M + 2 - N)ABC. It is important to note that M and N are previously defined data names with associated positions in the communication array.

Suppose the two commands listed below were executed in the order shown.

```
    (1) ADD PHRASE: SYMBOLIC CAP EXAMPLE,
        I(5)M, I(6)N, (M+2-N)ABC-;

    (2) SYMBOLIC CAP EXAMPLE, M30, N25;
```

Execution of command (1) places the phrase "SYMBOLIC CAP EXAMPLE" into the language dictionary (PFILE). This phrase specifies that CAPs 5 and 6, known as M and N respectively, should contain integer values and that the symbolic CAP (M+2-N), known as ABC, should contain logical FALSE (if no override is specified) as the initialization value. Execution of command (2) first causes the values 30 and 25 to be stored as integers in CAPs 5 and 6. Then the symbolic CAP (M+2-N) is evaluated. The expression is evaluated using the current contents of the CAPs specified by the symbols within the expression. Thus, the symbolic CAP in this case is evaluated as (30+2-25). Since no override was specified in command (2) for ABC, CAP 7 will be set to logical FALSE.

It is possible to direct the PLAN system to evaluate a symbolic CAP by using the actual displacements of the data names in the expression rather than their associated contents. This method of evaluation is specified by prefixing the data names involved with S'. If in command (1) above we had written (S'M+2-S'N)ABC- instead of (M+2-N)ABC-, regardless of the values specified for M and N by command (2), CAP 1, known as ABC, would be set to logical FALSE. The symbolic CAP (S'M+2-S'N) is evaluated using the CAPs (or displacements) of M and N, which are, respectively, 5 and 6. Thus, the symbolic CAP in this case is evaluated as (5+2-6). It is important to note that as a phrase is being processed, data names are added to the symbol table (see "PLAN Language Use", 4.2.3) from left-to-right and the execution-defined symbolic CAPs are evaluated in sequence. This means that the identity of a data name may change during symbol table creation. Example:

```
    ADD PHRASE: TEST, (1)A, (S'A+2)A,
    (S'A)C, (14)A;
```

would yield the following symbol table:

```
    (3)C, (14)A
```

CAPs defined as Arithmetic Operands at language definition time (ADD PHRASE) must result in an effective value of less than 512 if a scale factor is defined; otherwise, less than 8,176.

Note: All defined limits (such as less than 512, less than 8176) will be more clearly understood by study of the organization of the Phrase Entry Table in Appendix E, 12.1.4.

4.3.10 CAP DEFINED AS AN ARITHMETIC OPERAND AND AN IMPLIED DO

An example of a CAP defined as the combination of an Arithmetic Operand and an Implied Do is (M+2,10,2)NAME1. As in the case of a symbolic CAP, a data name is required. In this example, the data name is NAME. The above example is evaluated in the following manner:

1. The first parameter, I_1, is evaluated at execution time by obtaining the current contents of the data name M and adding the constant 2 to that value. The result of the evaluation specifies a CAP which will be known as NAME and will be the first referenced CAP of the Implied Do.

2. The second parameter, I_2, is a displacement from the result obtained by evaluating I_1. When this displacement is added to that result, the last CAP to be referenced by the Implied Do becomes known.

3. I_3 is the increment.

Note that I_2 and I_3 must be integer constants.

Assume that the current content of the CAP known as M is 30. Then I_1 would be evaluated as 32 and I_2 would become 42. Since in our example, I_3 is 2 and a default value of 1 is defined for the Implied Do, CAPs 32, 34, 36, 38, 40, and 42 would be initialized with the value 1. Since a CAP defined as the combination of an Arithmetic Operand and an Implied Do reduces to an Implied Do (at execution time), it is not possible for PLAN to determine the range at ADD PHRASE time. To ensure that the rule, range divided by increment must equal a whole number, is not violated, the user must define I_2 and I_3 as constants, where I_2 is an integer multiple of I_3.

4.3.11 DATA NAME

The data name allows the definition of data to be processed with the phrase in terms familiar to the problem definer. For example, suppose the problem definer were interested in calculating the electromotive force or voltage across a wire of given resistance at varying currents, he could define the data names as V, I, and R, where V = I x R (Ohm's Law). However, if the data names chosen were VOLTS, CURRENT, and RESISTANCE, the data items these names represent might perhaps become more meaningful to the problem definer. A single data name may be associated with <u>only</u> a communication array position (hereafter identified as CAP) or a Switch Word. Remember, the term "communication array" refers to both the managed and nonmanaged data arrays, and the term "Switch Word" refers to one of 15 32-bit words that comprise the System Switch Words. A data name must be present if a symbolic CAP is used (see "Communication Array Position", 4.3.6). A data name may be changed at any time by the user to one more meaningful to him without causing a change to the problem-oriented logic modules. If a data name change is desired, the user must consult the system analyst so that the system analyst may reflect the data name change in a dictionary (PFILE). Data names as defined in the preceding definitions follow the restrictions listed below:

1. A data name must be one WORD (a sequence of one or more alphabetic characters with no embedded blanks).

2. A data name must not be the single letter E.

3. A data name must not contain numerics or special characters.

4. Since PLAN truncates WORDS to three characters or pads out WORDS with blanks to three characters, care must be taken when choosing data names to ensure that duplication of data names does not occur within the same phrase.

5. The data name must immediately follow the definition of the CAP.

The following example illustrates definition of the data element with the data name VALUE (effective PLAN data name VAL) to be associated with CAP 33. This means that when the data name VALUE is encountered in a problem description stream, any data associated with it will be stored in position 33 of the communication array.

```
(33)VALUE
```

4.3.12 DEFAULT VALUES

A default value, defined in a phrase is a value that is used to initialize an associated CAP when that phrase in PFILE is executed. The default value may be a literal, a logical constant, or a numeric constant, and must immediately follow the data name. If a data name is not defined, the default value must immediately follow the CAP definition. For a review of a literal, a logical constant, and a numeric constant, see "PLAN Language Terminology", 4.1.0. The following two commands cause the phrases A and B, which specify different default values for the same CAP, to be entered into the language dictionary (PFILE):

```
ADD PHRASE:  A,(20)"ABC";

ADD PHRASE:  B,(20)VALUE70;
```

Suppose now, after the two commands above have been executed, the three commands listed below are executed in the order shown.

```
(1)  A;
(2)  B;
(3)  B,VALUE-;
```

Execution of the command in step (1) would cause CAP 20 to be initialized with the default value "ABC" (core representation ABCb). Execution of the command in step

(2) would cause CAP 20, known as VALUE, to be initialized with the default value 70. Execution of the command in step (3) would cause CAP 20, known as VALUE, to be set to logical FALSE. Step (3) is an example of how the problem definer may override a default value. Steps (1) and (2) illustrate that PLAN supplies the standard data value (default value) whenever commands fail to specify an override. Of course, if no default value was defined for a CAP at ADD PHRASE time, and no value is specified for the CAP at phrase execution time, PLAN would have nothing to supply as the standard data value. In that case, the CAP would contain residual data stored from the previous command in the input stream. The "default value" option should prove useful to the PLAN user as a means of reducing execution-time problem definition.

4.3.13 SCALE FACTORS

Input data may be scaled by a specified power of 10 in the range of plus/minus 7. Scaling is indicated by a P±n indicator, where n is in the range of 1-7. A plus sign indicates movement of the decimal point to the right; a minus sign indicates movement to the left. The scaling indicator must immediately precede the CAP indicator. In the following example, the value associated with the identification VALUE is to be scaled from feet to hundreds of feet. The value is stored in the communication array position 33. A default (initialization) value of five defines a standard data value of 500. Scale factors may not be used with CAP indicators that reference the System Switch Words, that is, CAPs that have negative subscript indicators. Scaling will be provided to either the default value or the given data value. Example:

```
P+2(33)VALUE 5
```

4.3.14 MODE

The normal storage mode of PLAN is real (floating-point). Literal data automatically overrides this standard. The user may designate integer (fixed-point) storage mode by prefixing an I to the scale factor, or to the CAP if no scale factor exists.

In the following example, a fixed-point 5 is associated as a standard value with communication area 33, and is assigned the data name VALUE. Example:

```
I(33)VALUE5
```

The following example illustrates the sequence of operations when a scale factor and mode are defined. Example:

IP+2(20)NAME4.1 is interpreted in the following manner:

- Scale by a factor of 100 (410.←4.1)
- Integer conversion (410←410.)
- Communication array(20)=410

4.3.15 CHECKING RULES

Checking of values in the communication array may be achieved during command processing (PSCAN execution) immediately before transfer to PLAN for the loading and execution of the first problem program module. The rules to be followed in performing the checks at command processing (problem-solution) time are defined in the phrase and are known as a check entry.

A check entry contains the following two parts:

(1) Definition of a test to be performed
(2) Definition of an action to be taken if the test fails

The definition of the check entry is written in the relative location within an element's description normally used for the standard data value, or it immediately follows the standard data value. The result of the test may be used to terminate processing of the command, to alter the sequence of programs to be executed, or to generate a diagnostic message. The general form of a check entry definition is:

```
(N) TEST ACTION
```

The N is the CAP (in the range of -15 to +8176). TEST indicates the condition to be tested, and ACTION controls the action to be taken if the defined test fails. If the defined test passes, the ACTION is ignored. The conditions that can be tested and the ACTIONS that may be specified are shown in Figure 6.

	ACTION								
TEST	None	'LIST'	C'DIAG'	A'DIAG'	P'PHRAS'	(N)	C(N)	A(N)	P(N)
* (NOT FALSE)	1,4	3,6	3,5	1,5	2,3,7	3,9	3,8	1,8	2,3,10
*T (TRUE)	1,4	3,6	3,5	1,5	2,3,7	3,9	3,8	1,8	2,3,10
*F (FALSE)	1,4	3,6	3,5	1,5	2,3,7	3,9	3,8	1,8	2,3,10
*R (REAL)	1,4	3,6	3,5	1,5	2,3,7	3,9	3,8	1,8	2,3,10

*NOTE: See Figure 7 for an explanation of the numbers in this figure.

Figure 6. Summary of check entry processing

CODE	SYSTEM ACTION TAKEN
1	Abort the command. PLAN level error recovery is initiated.
2	Last pushed command is the only command executed.
3	The processing of the current command is continued.
4	PLAN diagnostic 223-226 is generated.
5	The user diagnostic "DIAG" is generated.
6	The program list defined by LIST is added to the pop-up list.
7	The command "PHRAS" is executed next. The pop-up list is not modified.
8	The user diagnostic contained in PLAN literal form at position "n" of the communication array is generated.
9	The program list located at position "n" of the communication array is added to the pop-up list.
10	The command existing in PLAN literal form at position "n" of the communication array is executed next. The pop-up list is not modified.

Figure 7. Summary of check entry actions

Actions are controlled by the phrase context text (ACTION) that follows the condition TEST. For example, check entry *TC'DIAG' specifies a test for TRUE (*T) where action is to be taken according to the ACTION (C'DIAG') if the value tested is FALSE or REAL. The system action taken is specified by the two numbers indicated by the coordinates *T and C'DIAG', in this case 3 and 5. The numerical codes in Figure 6 are defined in Figure 7. Note: Logical TRUE in hexadecimal is 80000000. Logical FALSE in hexadecimal is 7FFFFFFF. Any other value is REAL.

The ACTIONS that may be defined are described below. The codes shown in Figure 6 that correspond to these actions are indicated within parentheses following each action heading.

<u>Abort Phrase</u> (None)
If, when the tested condition is not met, this phrase and following phrases are to be skipped until a phrase of equal or higher level is encountered (see "LEVEL"), no action indicator is required. A PLAN diagnostic logs the check entry failure and shows the word that was tested when the test failed (see execution error codes 223-226 in Appendix F, 13.0.0).

<u>Modify Pop-up List</u> ('LIST' or (N))
A string of program names may be added to the pop-up list if the tested condition is not met. The program names are included as a literal string, for example, 'M1111, M2222, M3333'. This literal string corresponds to the option 'LIST'. An alternate method for adding program names to the pop-up list makes use of the option (N). The option (N) is the subscript of the communication array location that contains the count of the number of words in the adjacent list. The program list in the communication array must be in EBCDIC representation. Each program name must occupy two PLAN words. The CAP position referenced by the check entry, for example, *T(10), may have been established with the following ADD

PHRASE entry. The list must be preceded with the integer count of the number of PLAN words to be added to the list. Example:

I(10)4, "PROGA", "PROGB"...

PLAN error recovery is not initiated. Additional information on the formation of program lists is given under "Program List". If any errors are found in the command or if the command is to be skipped as a result of level error recovery procedures, the programs will not be executed.

Use of the option to modify the pop-up list is a technique often used to call an error module if the expected amount of data is not given by the user. Data tests give either the results TRUE or FALSE, based on the logical value of the data word to be tested. If the result of the test is TRUE, the action option is not performed. If the result of the test is FALSE, the action indicated by the option is executed. An example of how this technique may be implemented follows:

ADD PHRASE: DATA, (5)A-*'PROGA'... ;

The check entry *'PROGA' tests CAP 5 for NOT FALSE. The test will fail if CAP 5 contains FALSE. Since a default value of FALSE is specified for CAP 5, the test will fail for all instances where no data is submitted to override the default value. If no data is submitted (the test fails), PROGA is added to the pop-up list and executed. It is assumed that PROGA is an error module.

The program added to the pop-up list, as a result of check entry action, is placed in the list after programs in the phrase program list (see "Program List", 4.3.4), and will therefore be executed first. If no action entry is specified (abort option), PLAN generates an error diagnostic for any FALSE result and returns, after all tests have been performed, to PSCAN for processing the next phrase.

Generate Diagnostic and Abort Phrase (A'DIAG' or A(N))
PLAN level error recovery is initiated following generation of a diagnostic when the condition is not met. The action indicator is either the constant subscript (A(N) option) of a location within the communication array that contains the count for the adjacent literal, or the literal itself (A'DIAG'). PLAN error recovery procedures are initiated to skip all phrases until a phrase of equal or higher level is encountered (see "Level of Phrase", 4.3.3").

Generate Diagnostic and Continue Phrase (C'DIAG' or C(N))
The execution of the current phrase is continued following generation of a diagnostic (A PLAN literal or the constant subscript of a location within the communication array that contains the literal) when the condition is not met.

Invoke Execution of a New Command (P'PHRAS' or P(N))
The new command is scheduled for execution. The command image is given as a PLAN literal, or as the subscript of the communication array location containing the literal. The terminating semicolon is not included in the literal, but a blank must be provided to allow its insertion. This action must not be used in a command that is to be implicitly saved (see "Statement Save", 4.3.22). More than one command may be "pushed" from a check entry because the scan of the command is continued. However, only the last one "pushed" (the rightmost check entry in the leftmost verb) will be executed. If any abort message is produced by the current command or if the current command is to be skipped as a part of level error recovery processing, the "pushed" command will not be executed.

The following examples illustrate the use of check entries.

1. (2)* writes a PLAN error message and aborts the phrase if the value of (2) is FALSE. *, the test for NOT FALSE, fails on FALSE only.

2. (2)*T writes a PLAN error message and aborts the phrase if the value of (2) is NOT TRUE. *T, the test for TRUE, fails on FALSE or REAL (NOT TRUE).

3. (2)*F writes a PLAN error message and aborts the phrase if the value of (2) is NOT FALSE. *F, the test for FALSE, fails on TRUE or REAL (NOT FALSE).

4. (2)*R writes a PLAN error message and aborts the phrase if the value of (2) is not REAL. *R, the test for REAL, fails on TRUE or FALSE (NOT REAL).

5. (2)*'P222,P345,P98,P35' inserts the program names P222, P345, P98 and P35 into the pop-up list, if the value of (2) is FALSE (NOT TRUE or REAL). Consecutive commas within the list eliminate all currently existing names from the pop-up list (see "Program

List", 4.3.4). Processing of the current phrase continues.

6. (2)*A'DATA VALUE MISSING' writes a diagnostic, DATA VALUE MISSING, and aborts the phrase if the value of (2) is FALSE.

7. (2)*C'ONE INCH RADIUS ASSUMED' writes a diagnostic, "ONE INCH RADIUS ASSUMED", and continues processing if the value of (2) is FALSE.

8. (2)*P'DUMP ' invokes execution of the command DUMP if the contents of (2) are found to be FALSE. The scan of the current command is continued. Note that implicit saving of statements is inhibited by failure of this test.

9. (2)*(30) If the test fails, add the program list whose length is contained in CAP 30 to the pop-up list, and continue processing the current phrase. The data at location 30 must be in the following general form:

   ```
   n, "NAME1", "NAME2", ...
   ```

   where: n is the number of PLAN (32-bit) words that follow and must be added to the pop-up list. A value of 2 times the number of program names must be specified for n.

   NAME1,... is a program name to be added to the pop-up list and must be given in two 32-bit words and must include trailing blanks.

   Note that the format of the array to be processed is identical to that required for processing by the PLAN loader subroutines (see CALL LIST/LEX/LOCAL/LREPT under "Program Linkage Routines", 5.11.1).

10. (2)*C(30) If the test fails, write a diagnostic and continue processing. CAP 30 contains the character count of the PLAN literal that is to make up the diagnostic text.

11. (2)*A(30) If the test fails, write a diagnostic and abort the phrase. CAP 30 contains the character count of the PLAN literal that is to make up the diagnostic text.

12. (2)*P(30) If the test fails, invoke execution of the command that is found in literal form minus the semicolon at CAP 31. The character count is found at location 30. The literal count must include a blank at the end of the text to allow for insertion of the semicolon. Note that implicit saving of statements is inhibited by failure of this test.

A phrase may be continued automatically by forcing the last check entry in a command to fail and thus invoke execution of the phrase continuation. Example:

```
ADD PHRASE: NAME,...(1)-*TP'CON NAME ';

ADD PHRASE:  CON NAME, ...;
```

Using this method, the formula area (see "Formula Area", 4.3.20) may not be split between commands if backward branching is required. Neither will data generated by the CONTINUE command be tested by check entries in the first command. Pop-up list (program) entries effected by the first command remain and will be executed following entries placed in the list by the CONTINUE command.

Combinations of logical expressions (see "Logical Operand", 4.1.11) and logical tests (see "Expressions", 4.1.12) may be used to change the implication of a phrase to suit the specific requirements of input data. The following example illustrates the use of checking to modify the program list for those cases where the value of ANGLE is less than .01.

```
...(2)ANGLE*R, (3)TEST
   *F'P204,,':(ANGLE<.01),...
```

In the above example, ANGLE is the data name assigned to CAP 2. A check entry is specified to ascertain that a value for ANGLE is given (REAL, not FALSE, or not TRUE). TEST, assigned to CAP 3, is set to TRUE if the value of ANGLE is less than .01; otherwise, it is set to FALSE. If TEST is found to be FALSE, no further action follows. If TEST is not FALSE, program name P204 is added to the pop-up list. The consecutive commas indicate that existing names in the pop-up list are to be eliminated.

4.3.16 EXPRESSIONS TO EVALUATE

Arithmetic and logical expressions may be defined to generate data values as arithmetic or logical results of other data values. Arithmetic expressions are introduced with the equal sign; logical expressions are introduced with a colon. In the examples of valid expressions given below, included blanks are optional:

```
(5) A=A*.017453,
(5) =B*100,
(5) B:(A>5) & (A<10),
(5) :  B & C|¬D,
```

Warning: If a data name for a CAP, which contains logical TRUE or FALSE, appears in an arithmetic expression, the resultant evaluation of the arithmetic expression will be logical FALSE. Assume the CAPs known as B, C, and D contain the values 4, +(TRUE) and 3.3, respectively. In the example, (1) A=B*C-D, CAP (1), known as A, would be set to logical FALSE because C contains a logical value (logical TRUE).

Standard data values (default values) and arithmetic or logical expressions may both be defined for a CAP. For example, (5) A-=A*.0174532965, defines CAP 5 to be initialized as logical FALSE if no data is submitted for A at execution time. Should data be submitted for A at execution time, the result of evaluating the arithmetic expression A*.0174532965 would be placed in CAP 5. The arithmetic expression above is one which converts degrees to radians.

4.3.17 CONDITIONAL EVALUATION

A data value may be conditionally generated on the basis of the evaluation of a logical expression. The arithmetic or logical expression which generates the data value, if the TRUE condition is found for the conditional logical expression, is preceded with a question mark (?). In the following example, NAME, defined for CAP 15, is set equal to AAA if the logical value of LLL is TRUE.

Example:
```
(15)NAME:  LLL?  = AAA,
```

A second expression to generate a data value, preceded by an exclamation mark (!), may be defined for evaluation if the conditional logical expression, when evaluated, is found to be FALSE. The TRUE option must be present if the FALSE option is to be used.

In the following example, DATA, assigned to CAP 5, is set equal to A multiplied by B, if it is TRUE that C is less than 50. If, however, C is not less than 50 (C≥ 50), the expression following the FALSE indicator (!) is evaluated. This expression will set CAP 5 to logical TRUE if both D and F are TRUE. Otherwise, CAP 5 will be set to logical FALSE.

Example:
```
(5)DATA:  (C<50)?=A*B !:D&F,
```

The data names A, B, C, D, and F must be defined in the current symbol table.

4.3.18 USER-EXIT PROGRAMS

User exits from PSCAN are provided to give additional conversion facilities of user-defined data. Examples of these could be feet to inches, extended precision, or fractional constants. Up to three different user-exit routines may be defined for any given PLAN phrase. When a user exit is required in a phrase, those data items that require user-exit processing are specially indicated.

When PSCAN encounters a data name that has an associated user-exit definition, the appropriate user-exit program will be called. Special subroutines are provided for scanning the input stream and placing the converted values in the PLAN communication array. These subroutines are IUSER, NUSER, GUSER, and EUSER. They are discussed later in this section. User-exit programs may not be used to store data into the Switch Words. All user-exit programs must terminate execution by a CALL EUSER. User-exit programs on the 1130 may not call LOCAL (see "PLAN Subroutine Use", 5.11.0).

During phrase definition time, standard phrase-defined data is written as follows:

```
IP±n(CAP) data name
```

Phrase-defined data that is to be processed by a user-exit program is written as follows:

```
Um IP±n(CAP) data name
```

U indicates that user-exit processing is required. The m represents one of three possible user-exit programs, associated with this phrase, to be executed and is expressed as 1, 2, or 3. Note that if user-exit programs are to be specified, the keyword EXIT may be used to specify the names of the three user-exit programs. If the keyword EXIT is not used, the standard default names EXIT1, EXIT2 and EXIT3 are automatically invoked (see "Exits", 4.3.19).

The definers I, P, and n are not available to the user-exit routine nor do they scale or alter format of the user-collected value. They are used by PSCAN when an expression is detected after a user-exit symbol. For instance, if the value associated with a user-exit symbol indicates the mode to be integer, and the data name is subsequently encountered in an expression, the data value will be treated as integer for purposes of the evaluation.

The user-exit program is entered during command scan time (PSCAN execution) when a data name with a user-exit definition is detected and the next significant character does not indicate an expression or literals to follow.

PSCAN will not relinquish control to the user-exit routine if the first recognizable nonblank character following the data name that is associated with the user exit is found to be an equal sign (=), a quote ('), an at sign (@), a double quote ("), a pound sign (#, BCD = sign), a colon (:), a comma (,), a left parenthesis ((), or a semicolon (;).

PSCAN will not honor further calls to the character fetch routine (see "CALL GUSER") after a comma (,) or a semicolon (;) has been processed. Either of these characters results in the return of a binary zero by the fetch routine to the calling routine. The user should then return control to PSCAN with no further character fetch requests.

It is the user's responsibility to index the communication array pointer when required. The user-exit program must alter the communication array pointer (see "CALL NUSER" below) by an amount equal to the number of PLAN words (32-bit) stored.

Four subroutine calls are provided for exclusive use within user-exit programs:

CALL IUSER must be issued before any other user-exit program subroutine is called. It provides linkage to PSCAN and on 1130 PLAN sets index register 1 to the LIBF subroutine linkage block as defined in Appendix A, 8.0.0.

CALL GUSER(ICHAR) accesses the next PSCAN input stream character and places it in ICHAR as an 8-bit EBCDIC character right-justified within the integer word ICHAR. The first CALL GUSER(ICHAR) issued following each entry into a user-exit program is the first nonblank character following the data name that caused the user-exit program to be invoked. A zero is returned if a comma or semicolon is encountered. Further GUSER calls should not then be issued without relinquishing control to PSCAN (see "CALL EUSER" below).

CALL NUSER(ISUB,ISW) places the current CAP in ISUB the first time it is called from each execution of a user-exit module. ISW is set to zero if it is permissible to store data values. If ISW is positive, the user-exit program must not store values in the communication array but must complete all other user-exit functions. The value will be positive if the subscript specified is too large or if a user-exit program is processed while executing a "go to" in the formula area evaluation. The second and each succeeding CALL NUSER issued during each execution of a user-exit program causes the CAP to be incremented by one. Thus NUSER should be called n+1 times if n 32-bit values are stored by a user-exit.

CALL EUSER(N1,N2,LIT) returns control to PSCAN. User-exit programs must exit via this call. If N1 is zero, no error is indicated. If N1 is positive, the parameters of this call are used as error parameters to call ERRAT. (See "PLAN Error Processing", 5.3.0.)

The examples listed below show data that could be processed with user-exit programming. Examples:

```
ADD PHRASE:  NAME, U1(5)ABC...;

1. NAME, ABC NODE FROM TO,...;

2. NAME, ABC 7'4" 8'5", ...;

3. NAME, ABC 7-4, 2-1, ...;

4. NAME, ABC LINE/DX4 DY7, ...;
```

The data in example 3 could be degrees-minutes or anything else the user wishes. The hyphen is used merely as a delineator.

Note that example 3 above results in two calls to the user-exit routine. The first call terminates (PSCAN returns a zero indicator and does not honor further calls to GUSER) when the first comma is encountered. The CAP points to ABC(1). When the comma is encountered, ICHAR is set to zero by CALL GUSER. CALL NUSER should then be called for the second time to increment CAP to the next value (ABC(2)). Then, since no data name is given for the next data item (2-1,), the same formatting rules (mode, user exit, scale factors) are used as for the preceding input value. (See "Data Value", 4.2.1.) A user-exit program is never entered unless the appropriate data name is encountered in the input command data stream.

The following example would process only one call to the user exit. The letter A following the comma would be interrogated as a DATA NAME:

```
NAME, ABC A1B, A1C,...
```

The command shown above has the following implications when executed by normal PLAN PSCAN processing.

NAME	VALUE
A	1
B	TRUE
A	1
C	TRUE

The statement scan may at times be required to pass over symbols and data that normally require user-exit conversion. This will happen during transfer of control over user-exit-associated data names when evaluating execution-defined expressions. An indicator ISW (see "CALL NUSER") is zero if the user-exit program is to store values, and nonzero if values are not to be stored. In either case, the user-exit routine must CALL GUSER until a zero-value is returned in ICHAR, or an error may result from PSCAN causing a phrase abort.

The following user-exit routine, written with 1130 FORTRAN control cards, illustrates a FORTRAN user-exit program to convert input in the form of feet-inches (3'11") into a value in inches. A portion of the ADD PHRASE and execution-time phrase are also shown.

```
ADD PHRASE: FOOT INCH, EXIT'EXIT1, FINCH,
EXIT3', U2I(12)FIN,...;

FOOT INCH, FIN 1'3", 12'3", 4', 9",...;

// JOB
// DUP
*DELETE     FINCH
// FOR
*LIST ALL
      COMMON L(625),LS(15),KA(510),PA(2196)

C     SEE APPENDIX A FOR SPECIFICATION OF
C     PA (PROGRAM AREA PROTECTION)
C     INITIALIZE USER EXIT LINKAGE
      CALL IUSER
C     INITIALIZE SUM OF INCHES
      NSUM = 0
      NTEMP = 0
C     SET MODE SELECTION
      MODE = 0
C     IS STORE VALID
      CALL NUSER(ISUB,ISW)
C     FETCH CHARACTER
    1 CALL GUSER(ICHAR)
C     HAS SCAN BEEN TERMINATED
      IF (ICHAR) 25,25,5
C     IS IT SINGLE QUOTE
    5 IF (ICHAR -125) 20,10,35
C     IS A SINGLE QUOTE ACCEPTABLE
   10 IF (MODE -1)15,30,30
C     CONVERT FEET TO INCHES
   15 NTEMP = NSUM * 12
      NSUM = 0
C     SET MODE SWITCH
      MODE = 2
      GO TO 1
C     INVALID CHARACTER INCREMENT SUBSCRIPT
   20 KERR = 101
   22 CALL NUSER (ISUB,ISW)
      KCHAR = ICHAR
C     SCAN OUT TO END OF FIELD
   23 CALL GUSER (ICHAR)
      IF (ICHAR)24,24,23
C     SET ERROR CODE - GIVE CHARACTER CODE
   24 CALL EUSER (KERR, KCHAR, 0)
C     INCREMENT SUBSCRIPT, STORE VALUE, AND
C     EXIT
   25 IF(ISW)27,26,27
   26 KA (ISUB) = NSUM + NTEMP
      CALL NUSER (ISUB,ISW)
   27 CALL EUSER (0,0,0)
C     INVALID FORMAT - FOOT MARK INVALID
   30 KERR = 102
      GO TO 22
C     IS CHARACTER INCH INDICATOR
   35 IF (ICHAR-127) 20,40,60
C     IS INCH MARK VALID
   40 IF (MODE-2) 50,50,55
C     ERROR 103
   45 KERR = 103
      GO TO 22
C     SET OTHER CHARACTERS INVALID
   50 MODE = 3
      GO TO 1
C     INVALID CHARACTER FOLLOWING INCH MARK
   55 KERR = 104
      GO TO 22
C     IS CHARACTER ACCEPTABLE
   60 IF (MODE -3) 65,55,45
C     ACCUMULATE SUM IF NUMERIC
   65 IF (ICHAR -240) 20,75,70
   70 IF (ICHAR -249) 75,75,20
   75 NSUM = NSUM*10 + ICHAR-240
      GO TO 1
      END
// DUP
*STORECI    WS   UA   FINCH
```

4.3.19 EXITS

The keyword EXIT introduces a three name program list specifying the names of the modules to be executed as user-exit programs 1, 2, and 3, respectively. The following example illustrates definition of PROGA for user exit 1, PROGB for user exit 2 and PROGC for user exit 3. Example:

```
    EXIT 'PROGA, PROGB, PROGC',...
```

If user-exit programs are specified for data item conversion and the keyword EXIT is not used to provide a routine name, the default program names EXIT1, EXIT2, and EXIT3 are used.

The 1130 PLAN system provides a program named EXIT1 that converts data to extended precision. If the user were to program, compile, and catalog his two most common data conversion requirements under the names EXIT2 and EXIT3, the need to express the keyword EXIT and user-exit program names would be held to a minimum.

4.3.20 FORMULA AREA

When present, the formula area follows all other phrase-defined data and keyword entries (level, data items, program lists, etc.). The formula area is a special adaptation of the function provided by formula numbers during language use time. The formula area is introduced with a dollar sign ($) and is terminated with the phrase-terminating semicolon (;). The formula area is comprised of any number of formulas within the limits of the maximum phrase length. Formulas are separated with commas. Each formula may be labeled with one or more formula numbers. A formula number is a dollar sign ($) followed by an integer number in the range of 0 to 1024. The formulas following a conditional evaluation may be formula numbers that indicate the formula number to which control is to be transferred if the condition is satisfied. They may also be expressions as defined under "Conditional Evaluation", 4.3.17. Evaluation proceeds left-to-right within the formula area. Formula number zero may not be referenced in another formula.

The differences between the ADD PHRASE formula area and execution-time formula number usage are outlined below:

1. Allowable syntax organization

   a. ADD PHRASE: The formula area must be the last area in the defined phrase, that is, data item definitions must not follow the first occurrence of a formula number.

   b. Execution time: Data assignment not requiring expression evaluation (D57.5) may be intermixed with formula numbers and expressions as shown in the following example:

      ```
      EXECUTE, $5 A=B+C, D57.5,  F=(A/C),
      $2=B*C,...;
      ```

2. Unreferenced formula numbers

   a. ADD PHRASE: Unreferenced formula numbers are indicated by diagnostics (see Appendix F, 13.0.0).
   b. Execution time: Unreferenced formula numbers are not detected.

3. Formula number limits
   a. ADD PHRASE: 1-1024 (Numbers greater than 1024 or equal to 0 are ignored but are invalid as references.)
   b. Execution time: 1-32,767

4. Valid expression number suffixes

   a. ADD PHRASE: Formula numbers must be followed by another formula number, by a data name, by an expression, or a comma or semicolon.

   b. Execution time: No restrictions.

5. Undefined formula numbers

   a. ADD PHRASE: Undefined formula numbers are indicated by diagnostics (see Appendix F, 13.0.0).

   b. Execution time: Undefined formula numbers are not detected. A branch to an undefined formula number is treated like a branch to the semicolon. A branch to formula number 0 is not executed (acts like a NOP).

The following types of statements may be in the ADD PHRASE formula area. Each may be prefixed with a formula number. The data name in each case is optional.

1. Arithmetic evaluation
   data name = arithmetic expression,
   Example: `A=B*100+C`

2. Logical evaluation
   data name: logical expression,
   Example: `A:B|C&¬D`

3. Conditional evaluation
   data name: logical expression ? = arithmetic expression or: logical expression !=arithmetic expression or: logical expression
   ```
   Example:  A:(B<100) & (B>0)?=20!=0,
             X:(CA=-)&(B=+)?:A&B!:A|B,
   ```

4. Conditional branch
   data name: logical expression ?$n!$m,
   Example: `:(A>5) ?$3!$4,`

5. Unconditional branch
   :$n,
   Example: `:$3,`

6. Mixed conditional
   data name: logical expression ? expression !$n,
   data name: logical expression ?$n!expression
   ```
   Examples: A:(B="ABCD")?=1000 !$5,
             B:  (A=+)?$1 !:B|C,
   ```

Statements of the type defined in 4, 5, or 6 above may result in transfer of control. A maximum of 1000 branches is permitted for the execution of any phrase. This prevents the programming of endless loops.

The following example illustrates use of the formula area in the addition of a phrase. Reference should be made to "Logical Operand" and "Logical Expression" in the section "PLAN Language Terminology", 4.1.0. The example produces the count of the number of literal values given at phrase execution time with the data name "NAMES". The number of literals will be accumulated at N (nonmanaged array 1), and the CAP of the word following the last literal will be given at S.

ADD PHRASE: TEST, I(-9)8192, M, I(M+1)N0, I(M+2)S1, (M+3,97)NAMES-'DEFAULT', $0:(NAMES(S)=+)?$2!$4, $1N=N+1, S=S+(NAMES(S)+7)/4-.5, $4:¬NAMES(S)?$2!$1, $2;

The following explanation of the above example is treated step-by-step as executed at phrase-execution time:

<u>TEST,</u> gives the name of the phrase to add to the language dictionary (TES)
<u>I(-9)8192,</u> sets the size of COMMON to 8192 words (Switch Word 9)
<u>M,</u> sets the label M equivalent to Switch Word 10. It is assumed that Switch Word 10 is set at execution time by a level 0 command to the size of the managed array.
<u>I(M+1)N0,</u> assigns the label N to the first position of the nonmanaged array and sets the default value for the location to zero. It is assumed that Switch Word 10, the size of the managed array, has been set by a previous phrase.
<u>I(M+2)S1,</u> assigns the label S to the second position of COMMON beyond the managed area and sets a default value of 1.
<u>(M+3,97)NAMES-'DEFAULT',</u> sets the label NAMES equivalent to the third position of the nonmanaged array. The third to the hundredth position of the nonmanaged array is initialized to FALSE. The literal DEFAULT is set to the third, fourth, and fifth position of the nonmanaged array (see section "Multiple Data Element Definitions", 4.3.26).
<u>$0:(NAMES(S)=+)?$2!$4,</u> introduces the formula area ($0) and sets the number of the first formula to zero. If NAMES(S) is TRUE, transfer is to expression 2; otherwise, transfer is to 4. This tests for the use of NAMES without a given literal. (Note that formula number zero may not be referenced.)
<u>$1N=N+1,</u> adds one to the counter (count of literals) that was initialized to zero, located in the first position of the nonmanaged area.
<u>S=S+(NAMES(S)+7)/4-.5,</u> calculates the position of the word that contains the next literal character count.
<u>$4:¬NAMES(S)?$2!$1,</u> causes a transfer to formula 2 if NAMES(S) is FALSE; otherwise, transfer is to formula 1.
<u>$2;</u> indicates the end of processing.

Check entries defined in a phrase are evaluated following execution of phrase defined formulas. Thus, check entries may be used to test for the validity of tests performed within the formula area.

4.3.21 SWITCH WORDS

A block of 15 switches is a portion of the standard PLAN permanent residence section. The 15 switches are located in COMMON between the 625 32-bit word PLAN loader area and the communication array. The format of the COMMON Switch Words is:

1 The first word of the DYNAMIC file control block (ID(1)) of the file currently in use for statement saving. The word may indicate either an open or a closed DYNAMIC file (see "Dynamic File Support", 5.5.0).

2 Contains the statement number of the saved statement that is to be executed next. If this parameter is zero, processing is in the normal manner (processing from PLAN input device).

3 Contains the statement number of the last saved statement to be executed. Note that if Switch Word 2 contains a zero, Switch Word 3 and Switch Word 1 may be used for any desired user system function.

4-7 These switches are for the exclusive use of the application modules. Recommended usage of these switches as data pointers is outlined below.

8 Contains the subscript of the first word of a block of the communication array that may be treated as "erasable COMMON". The function of "erasable COMMON" is to provide an area of command and module independent memory that may be used by utility commands/routines and by user modules with the knowledge that they are not destroying system data required for continued execution.

The length of "erasable COMMON" is assumed to be the number of words to the end of COMMON. Thus, no length specification is required.

9 Contains the maximum size (in PLAN 32-bit words) of COMMON for the phrase being processed. This allows the user to manage COMMON. It is the sum of the requirement for the loader, system switch area, managed array, and any additional nonmanaged COMMON. The minimum allowable value for this switch is 640.

10 Contains the number of PLAN words to be managed by the PLAN level

management, as the managed communication array.

Setting this value to zero allows the use of the level error recovery facility of PLAN without initiating the disk access operations involved in the communication array data management.

(Switch Words 11-12 may define one of two functions as defined below.)

11-12 Contains the name of the user-written module that is to process error conditions, instead of the normal PLAN error processing procedures. User error modules may not call PLAN error subroutines (ERROR/ ERREX/ERRAT/ERRET, 5.11.6). PERRS will exit to the user error module via the checkpoint facility on each error on the 1130 version of PLAN. On S/360 PLAN, the named module is loaded as a LOCAL to process the error condition.

An array in the following format is created in erasable COMMON.

BYTE	CONTENTS
0-7	Program name causing diagnostic call
8-11	Error number (N1 from error subroutine call)
12-15	Error code (N2 from error subroutine call)
16-20	ID from cc. 76-80
21	hexadecimal FF = system error, 0 = user error
22	hexadecimal FF = abort, 0 = continue
23	(Unused)
24-27	Sequence
28-31	Length of literal in characters
32-107	Diagnostic literal
108-111	Literal count of phrase
112-561	Phrase text

11 If Switch Word 11 is zero or a positive number, it indicates the count of the number of diagnostic messages that are to be written onto DYNAMIC file 255 of drive 0 (error message queue file) before they are written on the device defined in Switch Word 12 or as a result of CALL ERLST. This option is not available if the diagnostics are processed by a user-written module. If this word is zero or one, diagnostics are written directly to the output device specified in Switch Word 12.

12 Contains the device code (see CALL IOCS, 5.11.5) for the device on which diagnostics are to be written.

13 Contains switches governing the mode of error processing. The low-order four bits of the integer portion of this switch word govern PLAN error processing.

BINARY BIT VALUE		
1	OFF	Short form error messages
	ON	Long form error messages
2	OFF	Stacked error processing
	ON	Immediate error processing
4	OFF	Dynamic file error abort
	ON	Dynamic file error continue
8	OFF	Permanent file error abort
	ON	Permanent file error continue

(See "PLAN System Diagnostic Processing", 13.0.0)

14 (Reserved)
15 User Functions

The switch words are initialized when PLAN processes any level 0 phrase. The settings assigned at that time are:

SWITCH	INITIALIZATION VALUE	
1	0	(saved statement file)
2	0	(saved statements not being executed)
3	0	(last statement saved)
4	0	(data list pointer one)
5	0	(data list pointer two)
6	0	(data list pointer three)
7	0	(data list pointer four)
8	490	(erasable COMMON)
9	1150	(625+15+510) number of 32-bit words in COMMON
10	0	(managed array size)
11	0	(error processing control)
12	100	(standard PLAN output device)
13	0	(error processing mode)
14	0	(reserved)
15	-	(not initialized)

The switch words should be set by the first command processed when PLAN is invoked to reflect the desired operating environment for this run. A suggested command for this function is "PLAN JOB;" (see "Standard PLAN Commands", 4.5.0).

Note that any attempt to use the standard PLAN utility commands DUMP COMMON, DUMP SWITCHES, DUMP MANAGED, DUMP NONMANAGED, DUMP DYNAMIC, DUMP PERMANENT, DUMP PHRASES, and others will not be honored if Switch Word 8 is not set to a valid (positive) erasable COMMON pointer.

The switch words are referenced in the same manner as the managed communication array, except that the subscript is negative in the range of -1 to -15. The following example illustrates the FORTRAN definition of a 350-word communication array, where M is the managed array and N is the non-managed array:

```
    COMMON L(625), LS(15), M(200), N(150)
```

4.3.22 SWITCH WORDS 4-7 AS DATA POINTERS

Serious consideration should be given to the use of the switch words as data list pointers. Logic modules written to expect data strings in predefined positions of the communication array may be useless when an attempt is made to process another data set which for some reason requires a different starting CAP.

The use of the switch word data list pointer concept allows processing of data no matter where it occurs. It also allows a user to shift his data bank location more freely without impact to application programming.

The following example illustrates use of Switch Word 4 as a data list pointer:

In the phrase definition "ADD PHRASE: NAME,I(-4)M, (M,200)A0...;" M is the direct pointer to the first data item of the data list. Thus, CAP names of M, M+1, M+2, etc., define successive values within the A data list. The user logic module need not concern itself with where the data array will actually be located because of this ability to reference the data symbolically. As a matter of fact, the problem definer, at execution time, may change the location of the data list at will without requiring the programmer to change the logic module. All that is required of the problem definer is that he issue a command which gives a value to M. An example of such a command follows:

```
NAME, M1;
```

Switch Word 4, known as M, would receive the value 1. This value would be the starting CAP for the 200-word array known as A. Since no override is specified for the default value zero, communication array locations 1, 2, 3, ..., through 200, would be set to zero.

Switch Words 4-7 are available to be used as data pointers allowing a maximum of four arrays to be <u>directly</u> referenced at any one time. However, it is possible to process more than four arrays at any one time. This can be done in an <u>indirect</u> manner. If one or more of the Switch Words 4-7 point to CAPs which are not the start of the data arrays, but rather are the start of a list of CAPs which point to the data arrays, it is possible to define as many arrays as is required by the user (within the limits of core). This principle is illustrated in Figure 8.

```
┌ - - - - -┌───┬──┬──┬──┐- - ┌──┬───┬─┬─┬─┬─┬─────────────────┬────────────────┬───────┐
|CONTENTS  |25|  0|  0|  0|    |30|130|E T C|  |<-----A ARRAY---->|<---B ARRAY--->| ETC. |
└ - - - - -└───┴──┴──┴──┘- - └──┴───┴─┴─┴─┴─┴─────────────────┴────────────────┴───────┘
 CAPS       -4 -5 -6 -7       25  26 27 28 29 30                  130
```

Figure 8. Schematic of the indirect data pointer

Programs that allow for maximum interchangeability of data may be provided if a convention using Switch Word (3+N) for data list N is adopted, where N is in the range of 1-4. If a value of N greater than 4 is required, the switch words should be used as indirect pointers as defined above.

Use of the data list pointer concept (Switch Words 4-7) is made easy by the two subroutines PARGO and PARGI. PARGO provides for transfer of data lists from a user array to the communication array location pointed to by an indicated switch word. PARGI provides for transfer of data from the communication array to a user array.

The following example shows use of the PARGO routine in transferring array F, in a module, to communication array location 20. Example:

```
    COMMON L(625), LS(15), M(255)
    LS(4)=20
    CALL PARGO(4,F)
```

In the above example, if F(1) contained a 10, then communication array position 20 would be set to 10 and F(2) through F(11) would be transferred to communication array positions 21 through 30.

If the origin arrays are also in the communication array (COMMON), more efficient execution can be gained by coding the COMMON references with symbolic subscripts to avoid physical movement of data. For example, if the task of the module is to add up the data list, it can be coded:

In the module creating the array:

```
    LS(4)=INDEX
```

In the module summing the array:

```
    DIMENSION FILE(...
    COMMON L(625),LS(15),M(...
    EQUIVALENCE (FILE(1),M(1))
    K=LS(4)
    LGTH=M(K)
    IJ=K+1
    I2=K+LGTH-1
    SUM=0.
    DO 2 I=IJ,I2
  2 SUM=SUM + FILE(I)
      •
      •
      •
```

4.3.23 STATEMENT SAVE

Any PLAN statement or group of PLAN statements, that define a procedure, can be saved in a PLAN DYNAMIC file for later execution. Identification of saved statements is by statement number and through use of the PLAN switch words. Statement numbers, in the range of 1 to 32,767, are prefixed to any PLAN statement that is to be saved. Thus, the PLAN statement takes on the general format shown in the following example:

```
    N COMMAND, DATA; N COMMAND, DATA;...
```

In the example, N, when present, is the statement number. Note that it is a distinct advantage functionally to keep N as small as possible.

Statements are saved either implicitly or explicitly. The standard PLAN command SAVE starts the saving in the designated PLAN DYNAMIC file of the commands that follow. Saving operations are terminated when (1) a new SAVE command, (2) a SEND (Save End) command, or (3) an unnumbered command is encountered. Statements saved explicitly are not executed during the saving operation. Every statement to be saved must be numbered. The following example illustrates the use of the SAVE command for saving statements that describe a procedure.

```
    SAVE, FILE 45, DRIVE 1;
    5  COMMAND;
    6  COMMAND;
    7  COMMAND;
         .
         .
         .
       SEND;
```

The SEND command has no parameters.

The SAVE command has the following parameters:

FILE	This data item defines the PLAN DYNAMIC file that is to be used for saving the commands. If the value is not provided, the current file number in Switch Word 1 will be used. The file does not need to be open (see "DYNAMIC File Support", 5.5.0).
DRIVE	This data value defines the PLAN DYNAMIC Drive on which the statement save file is located. If the value is not provided, the current value in Switch Word 3 divided by 2048 will be used as the dynamic drive indicator.

Numbered PLAN statements that are not explicitly saved are automatically saved (implicit) but are also executed. ADD PHRASE, ALTER PHRASE, and DELETE PHRASE commands <u>may not</u> be implicitly saved. Statements saved implicitly are stored in the file indicated by PLAN Switch Word 1 (see "Switch Words", 4.3.21). If a statement number is the same as a statement already on that file, the new statement replaces the old statement. Any error found by PSCAN while scanning a statement to be saved implicitly will inhibit the saving of the statement. If no previous file has been designated, file 254 is used on drive 0. Note that the rules stated under "DYNAMIC File Support", 5.11.0, control also the release of statement save files. Thus, a permanent statement save file may not be defined on DYNAMIC Drive 0. The failure of a check-entry with a "P" action code (*P'PHRASE') prevents implicit saving of the PLAN statement.

Saved PLAN statements may be executed at any time through entry of an EXECUTE command delineating the limits of the statements to be executed. These limits are maintained in the PLAN switch words, and may be program-altered at any time. Execution always proceeds to the next-higher-numbered statement that is in the save file regardless of the order in which the statements were added to the save file. An example of the EXECUTE command is given below:

```
    EXECUTE, FROM  5  TO 10, FILE 45, DRIVE
    1;
```

The EXECUTE command has the following parameters:

FROM This data item defines the lowest-numbered saved statement to be executed as a result of processing this command. If the number is not in the file, an error diagnostic (DFJ172) will be given.

TO This data item defines the highest-saved statement that may be executed as a result of this EXECUTE command.

FILE This data item defines the number of the DYNAMIC file that contains the saved statements to be executed. If this parameter is not specified, the current file number in Switch Word 1 will be used.

DRIVE This data defines the DYNAMIC drive number that contains the saved statement If the value is not provided, the current value in Switch Word 3 divided by 2048 will be used as the DYNAMIC drive indicator.

Execution of saved statements may also be controlled by any command or logic module. A saved statement is executed any time the PLAN loader is entered and (1) the pop-up list is found to be empty, and (2) system Switch Word 2 is not zero.

Therefore, any logic module or any command that properly sets system Switch Words 1, 2, and 3 may control (start, stop, or modify order) saved statement execution.

The following commands illustrate the use of saved statements for looping within a command string. The first commands are the ADD PHRASE commands to define the controlling statements. Note that statements 1-7 adjust the sequence of execution.

```
    ADD PHRASE:  GO, I(-2)TO;
    ADD PHRASE:    IF,   I(1)TEST,   (-2):
    TEST?=TEST(2);
    ADD PHRASE:  DO, (5)LIT, I(3)A5, ...;
    SAVE;
    1 DO, 'THIS'...;
    2 DO, 'THAT'...;
    3 GO, TO7;
    4 DO, 'SOMETHING',A0...;
    5 ...
    6 ...
    7 IF, :(A>4),4;
    EXECUTE, FROM 1 TO 7;
    NEXT PHRAS,...;
```

The EXECUTE statement in the above example results in the following statement executions.

STATEMENT			
NUMBER	CAP	DATA NAME	SET TO
1	3	A	5
	5	LIT	4,'THIS'
2	3	A	5
	5	LIT	4,'THAT'
3	-2	TO	7
7	1	TEST	+(TRUE)
	2	TEST(2)	4
	-2	TO	4
4	3	A	5
	5	LIT	9,'SOME','THIN',G'
	<3	<A	<0
5	•	•	•
6	•	•	•
7	1	TEST	-(FALSE)

Execution continues with 'NEXT PHRAS'.

In the above example, execution of statement 3 causes control to pass to statement 7 by setting the PLAN Switch Word 2 to the statement number of the next statement to be executed. Statement 7 tests the first position of the communication array for presence of a logical TRUE. If found to be TRUE, the contents of the second position of the communication array are moved to Switch Word 2 to indicate the next statement to be executed. Thus, a loop can be established and statements executed under program control.

4.3.24 IMPLIED DATA ELEMENT DEFINITION

Phrase data element definitions may be implicitly defined as successor elements to previously defined data elements. An implied data element definition may not follow a data element definition including a symbolic subscript. A data element definition is organized as follows:

```
    | F | S | N | V |
    L___L___L___L___J
```

F	contains the format control, (user-exit control, mode control, and scale factor)
S	contains the communication array subscript (CAP)
N	contains the data name
V	contains the initialization values, check entries, and phrase-defined expressions

The S and N sections may be implied as long as the sections to the left of the section to be implied are not included. A comma within the command indicates a new data element definition; therefore, any following data values are implicitly defined. The CAP of the implied data element definition is one greater than the previous CAP.

Implicit definition of S leads to a value of the next communication array position. Example:

```
    IP+2(10)A5, B7,...
```

The above example would assign the standard value of 7, scaled by 100, in the integer mode, to the data name B at CAP 11. In the following example, B is assigned to CAP 13 and is stored in floating-point mode. Example:

```
    (10)A'LITERAL', B7,...
```

In the previous examples the data name B was also optional. The location equivalent to B in the two examples could be referenced in an execution-time statement by A(2) and A(4), respectively.

Additional implicit definitions are given in the examples below:

```
    (10)A5, 'LITERAL'...
    (10)A, = A*100,...
    (10)A, :A|B...
    (10)A, *TA 'POSITION 11 BAD'...
    (10,20)A, B32...
```

If the implied value is the first item to be defined in the phrase, CAP 1 is assumed.

Implicit definition also applies to the formula area. Execution of the following commands would yield a value of 1 in CAP 10, 4 in 11, and a logical TRUE in 12. Example:

```
    ADD PHRASE:    TEST,    (10)A  $0=A+3,
    :(A=4)?=+;
    TEST,A1;
```

4.3.25 PSCAN EXECUTION SEQUENCE

This section describes the sequence of operations during interpretation and processing of an execution-time statement.

1. Phrase entry. The phrase definition is is retrieved from PFILE. The managed array is initialized, saved, and restored according to the rules defined by the level indicated for the phrase.

2. VERB phrases. Program names are added to the pop-up list as the VERB phrases are encountered in a left-to-right manner. The lists processed at this time are those associated with the keyword VERB.

3. Symbol table initialization. The symbol table is initialized according to the level of the phrase. Data names from this phrase are then added to the symbol table in a left-to-right order as defined in the ADD PHRASE (further information is described in the discussion of Table 3 of the phrase entry table in Appendix E, 12.0.0). Data names from VERB phrases follow data names from the OBJECT phrase with the data names from the leftmost VERB phrase entered last. Symbol table construction is illustrated in Figure 9.

   The symbol table construction effect is illustrated in the following example:

```
ADD PHRASE:  ONE, LEVEL 1, (2)A, (1)A,
(3)B1,0;
ADD PHRASE:  THR, (2)B, (4)C;
ADD PHRASE:  TWO, C, VERB, (3)C;
ONE;
TWO THR, C1, B2, A3;
```

   The communication array will contain the following data after the above execution. Underlined letters represent the final symbol table entries.

```
CAP  CONTENTS  SYMBOL TABLE ENTRIES
 1      3          A       C
 2      2        A    B
 3      1           B        C
 4      0                C
                --------------->
                 TIME
```

   Note that the final symbol table definition of each data name is underlined. Entries not underlined are subsequently overridden.

   The first command issued is "ONE;". Therefore, PSCAN first analyzes that PFILE phrase entry. Since ONE is a level 1 command, a new symbol table is started. The first data item encountered as PSCAN scans the phrase entry from left to right is A. A is entered into the symbol table as the data name for CAP 2. The next data item encountered is also named A. The CAP specified for A this time is 1. The previous reference of A is eliminated from the symbol table, and the new reference for A is recorded in the symbol table. The data name A now refers to CAP 1. The next data item encountered is named B. B's definition causes an entry to be made in the symbol table that specifies that CAP 3 is to have an associated default value of 1. The last data item encountered in this phrase is the value 0. Since no CAP is explicitly defined for this value, PSCAN assigns this value to the next available CAP. Therefore, CAP 4 is assigned the value 0.

The following diagram represents the results of the steps described above.

```
CAP  CONTENTS  SYMBOL TABLE ENTRIES
 1                  A
 2                A
 3      1              B
 4      0
```

If the reader uses the method of analysis outlined above in accordance with the rules for symbol table initialization, the diagram that shows the final symbol table entries should become apparent. Hint: Analyze the OBJECT phrase "THR" before the VERB phrase "TWO".

4. Data initialization. Default values defined in the ADD PHRASE are stored in the communication array. Default values from VERB phrases (right-to-left) are processed following processing of default values from the OBJECT phrase.

5. Input analysis. Execution-time data is converted and moved to the communication array. This phase includes all data defined in the input stream as analyzed in the left-to-right order.

6. Phrase-defined expressions. All expressions defined in the ADD PHRASE including the formula area, are evaluated. The formula area for each phrase is evaluated following data item expressions.

7. Program list. Program lists are added to the pop-up list.

8. Check entries. All check entry tests are performed in a left-to-right order and the appropriate specified action is executed. If there are VERB phrases (see "Verb Designation and Program List", 4.3.5), steps 6 through 8 are repeated in a right-to-left manner.

The following example illustrates the contents of the communication array following expression execution of phrase-defined and execution-defined expressions: An execution-defined expression is defined as any expression contained in the execution-time input stream.

```
ADD PHRASE:  ONE, LEVEL 1, A, B0, =0,
=6, $0 B(3)=9;
ONE, $1A5:(B=1)?$9, A4, B=1, 5, 7,:$1;
```

```
          CONTENTS ACCORDING
CAP NAME  TO PSCAN SEQUENCE
 1   A      5 4       5
 2   B    0     1
 3                5     0
 4                  7     6 9
```

PSCAN execution for the above example occurs in the sequence listed below:

1. Default values defined in the ADD PHRASE are stored in the communication array. This causes B, which is assigned CAP 2, to be set with a default value of zero.

2. Execution-time data as defined in the command "ONE" is converted and moved to the communication array. The first data name encountered, scanning left to right, is A.

   A has been given the data value 5. Hence, CAP 1 (from previous A definition) will be given the value 5. Next, PSCAN encounters the logical expression ":(B=1)?$9". Since B at this point in time contains a zero, the branch to formula number 9 is not taken. PSCAN, next, sets A(CAP 1) to 4, B(CAP 2) to 1, and CAP 3 and CAP 4 to 5 and 7, respectively. The next data item encountered, ":$1", causes an unconditional branch back to formula number 1. Once again A(CAP 1) is set to 5. The logical expression ":(B=1)?$9" this time, however, is TRUE (since the arithmetic expression B=1 was executed above), and a branch is taken to formula number 9. But, since formula number 9 has not been defined in this command, a branch is taken to the semicolon (see "Formula Area", 4.3.20), and PSCAN execution continues as described in 3.

3. Expressions defined in the ADD PHRASE, including those in the formula area, are evaluated. The evaluation of the expressions in the

formula area follows the evaluation of the data element expressions. Since CAP 2(B) was just tested, the PSCAN CAP pointer now moves over to CAP 3 and the value zero is stored. PSCAN then sets CAP 4 to 6.

Finally, the formula "$0B(3)=9" is evaluated. Evaluation causes a 9 to be placed in B(3). Since B(1) is CAP 2 and B(2) is CAP 3, then B(3) is CAP 4. Hence, a 9 is stored in CAP 4.

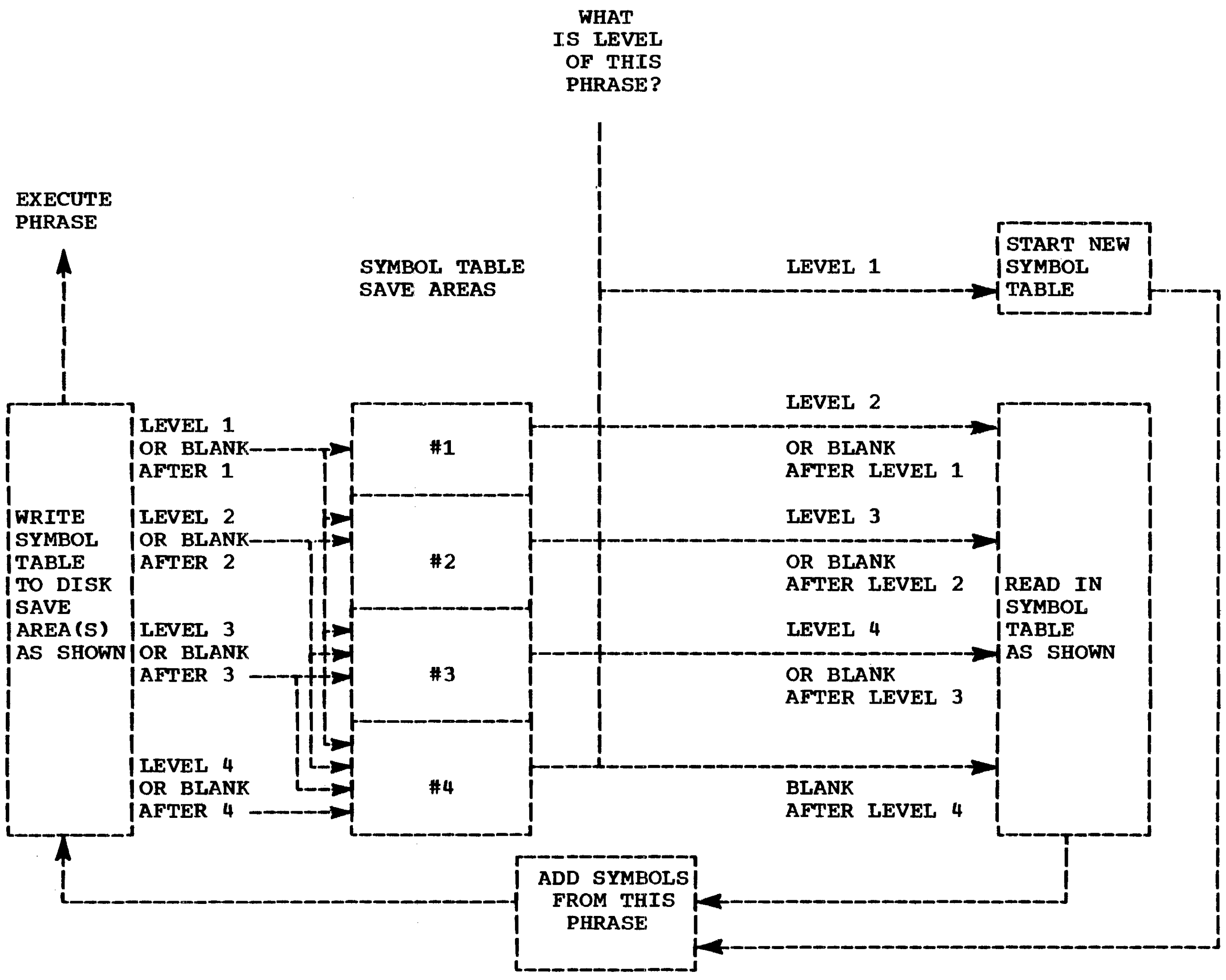


Figure 9. Symbol tables save and restore logic

4.3.26 MULTIPLE DATA ELEMENT DEFINITIONS

Multiple data elements may be defined and referenced to the same communication array definition. This is possible because the PSCAN CAP pointer does not normally increment to the next CAP until the comma that signals the end of a data element's definition is encountered. The general format of a command with the additional organization requirements that must be followed in using multiple definitions is shown in the following schematic:

```
| F | S | N | C | L | E | X |
└───┴───┴───┴───┴───┴───┴───┘
```

F contains any format indicators (user-exit control, mode control, and scale factor)

S	is the communication array subscript (CAP)
N	contains the data name
C	contains the default numeric or logical data values. It is this section only of the data element definition that may be propagated through an array by a CAP defined as an Implied Do.
L	contains the default literal data values
E	contains the check entry information
X	contains the phrase-defined expressions

The following example illustrates a sample entry defined to set an array containing 100 values to FALSE, to set a standard literal to the first five array positions, and to test for the proper use of the data name and corresponding literal. In other words, the following example has been created to ensure that if the data name ARRAY is specified in a command, its associated expected literal is also present. If the associated literal is omitted, error messages are issued. Example:

```
ADD PHRASE: SAMPLE, (25,124)ARRAY-
'STANDARD DATA' *A'NO DATA' *RA'NAME
ONLY',...
```

In the above example, the following sequence of events takes place within PSCAN:

1. A logical FALSE (7FFFFFFF) is set to communication array positions 25 to 124. Note that ARRAY is entered into the symbol table equivalent to CAP 25 and that the PSCAN CAP pointer rests at CAP 25.

2. "STANDARD DATA" is set into ARRAY(2) through ARRAY(5). The literal count is set to ARRAY(1). Since the PSCAN CAP pointer rests at CAP 25 and since a single quote literal definition requires a character count to be stored, the count is stored in CAP 25 (ARRAY(1)). The literal itself is stored in CAPs 26-29 (ARRAY(2) through ARRAY(5)) as STAN,DARD,bDAT and Abbb, respectively. The literal character count is 13, which does not include the padded blanks in CAP 29 but does include all other blanks (CAP 28) which may be part of the literal itself. Note: The PSCAN CAP pointer still sits at CAP 25 since the data item terminating comma has not been encountered.

3. A check entry is made on CAP 25 for not FALSE. If the value is found to be FALSE, the diagnostic "NO DATA" is generated and execution of this phrase is terminated. The location could be FALSE only if entered as a data value, for example, ARRAY-, since initialization places a standard literal in the location.

4. A second check entry is made on ARRAY(1) for real. Since FALSE has previously been checked, this is essentially a test for TRUE, although FALSE would also cause failure of this check. If found to be TRUE (or FALSE), the diagnostic "NAME ONLY" is generated and execution of this phrase is terminated. A TRUE value can result from a user entering the following command:

   ```
   SAMPLE, ARRAY,...;
   ```

   If a data name is defined and no associated value is specified, a logical TRUE is placed in the CAP represented by the data name. Since the sample ADD PHRASE contains check entries which test for this type of error, no harm is done. The phrase is terminated with a message and the user reenters the corrected command.

4.4.0 REVIEW OF LANGUAGE DEFINITION

This section is a self-teaching introduction to language definition under PLAN. It does not cover every possible option. Reference will be required to other parts of the manual for in-depth explanation and understanding of the questions asked.

The material is presented as a series of numbered questions. Following the question is either a "yes"(Y) or a "no"(N) entry indicating an action to be taken according to the answer selected. If only one answer follows the question and it is not the selected answer, this indicates that the next question is to be processed. Material in the answer section that is underlined indicates entries to be made in generating the ADD PHRASE. Transfers to other numbered questions are preceded with the number sign (#). A G instead of, or following, a Y or N entry indicates a transfer to the indicated question number. The following example illustrates the organization of this section. The first step is initiated with the question "Is today Tuesday?". If the answer is "yes", the "Y" option is executed. This tells the user to enter "TUESDAY" into the command. If the answer is "no", the "N" option is executed and the user enters the day of the week into the command in literal form. Processing continues at step A1. The G ("GO TO") could have been eliminated since execution of the next entry is always implicit. Example:

#27. Is today Tuesday?
Y. 'TUESDAY' G. #A1
N. 'LITERAL' (Enter day of week for literal) G. #A1

#A1. Is there a new phrase to be added to the system?
Y. ADD PHRASE: NAME, (NAME is one to five words. Only the first three characters of each word are considered significant.)
N. G. #Z1

#A2. Are there programs to be executed each time this phrase is executed?
Y. PROGRAM'NAME1
N. G. #A4

#A3. Are there additional program names to be added?
Y. ,NAMEn G. #A3
N. ',

#A4. Is this phrase one of a series of dependent phrases?
Y. LEVEL
N. G. #A6

#A5. Is this the independent phrase of the dependency group?
Y. 1,
N. n, (n=2,3,4 indicating increasing levels of dependency)

#A6. Is this phrase to be used as a modifier to another phrase?
Y. VERB
N. G. #A9

#A7. Are there programs to be executed after the programs associated with the object (nonverb) phrase?
Y. 'PROGA
N. G. #A9

#A8. Are there additional programs for the verb list?
Y. ,PROGN G.#A8.
N. ',

#A9. Are there to be data items that require special user-written programs to convert the data?
Y. EXIT
N. G. #B1

#A10. Is the program to be executed with user exit 1 named EXIT1, user exit 2 named EXIT2, and user exit 3 named EXIT3?
Y. ,
N. 'PROG1,PROG2,PROG3', ('PROG1,2,3' are the names of the programs to be associated with the respective user exit.)

#B1. Are there data items to be defined for this phrase?
Y. G. #B2
N. G. #C1

#B2. Is the data for this data item to be converted using one of the three user-exit programs associated with this phrase?
Y. Un (n=1,2 or 3)

#B3. Is the data item to be stored in the integer mode?
Y. I

#B4. Is the data item to be scaled before it is stored (multiplied by some power of 10)?
Y. P
N. G. #B6

#B5. Is the power of 10 greater than zero?
Y. +n (n ranges from 1-7)
N. -n (n ranges from 1-7)

#B6. Is the data item to be stored in the system switch words?
Y. (-n) (n ranges from 1 to 15)
N. G. #B36

#B7. Does the data item require identification so that it may be identified

at execution time?
Y. NAME (NAME is any combination of 1-3 alphabetic characters. If more than three characters are provided, they are ignored.)

#B8. Is there to be a standard (default) value provided for the data item?
N. G. #B13

#B9. Is the standard data value to be logical?
Y. #B11

#B10. Is the standard data value to be literal?
Y. G. #B12
N. nnn (nnn is a numeric field containing an optional sign, decimal point and exponential modifier.) G.#B13.

#B11. Is the logical value to be FALSE?
Y. - G. #B13
N. + G. #B13

#B12. Is the count of the number of literal characters to be stored along with the literal text?
Y. 'LITERAL TEXT'
N. "LITERAL TEXT"

#B13. Is the data item to be checked for status (TRUE, FALSE, REAL, NOT FALSE)?
N. G. #B27
Y. *

#B14. Must the location be REAL for the test to pass?
Y. R G. #B18

#B15. Must the condition be TRUE for the test to pass?
Y. T G. #B18

#B16. Must the condition be FALSE for the test to pass?
Y. F G. #B18

#B17. Must the condition be TRUE or REAL for the test to pass?
Y. G. #B18
N. Proceed and define a new data item that is set by evaluation of a logical expression defining the conditions to be set. Then define a check on the new data item. G. #B27.

#B18. Is PLAN to give a standard PLAN system literal if the test fails?
Y. G. #B26

#B19. Is the program list to be modified if the test fails?
Y. G. #B25

#B20. If the test fails, is a user-supplied diagnostic to be written followed by continuation of processing?
Y. C G. #B24

#B21. Is a user-supplied diagnostic to be written if the test fails followed by initiation of the PLAN error recovery scheme?
Y. A G. #B24

#B22. Is a new phrase to be invoked if the test fails?
Y. P
N. Please tell PLAN development what you would like to do if the test fails.

#B23. Is the text of the new phrase to be given in the phrase; not provided as text in the communication array?
Y. 'PHRASE TEXT ' (terminal blank in text required)
N. (CAP) (CAP is the location within the communication array that contains the PLAN literal of the phase to be invoked. A space is provided at the end for insertion of the semicolon.) G. #B26

#B24. Is the diagnostic text to be given in the phrase; not provided in the communication array?
Y. 'DIAGNOSTIC TEXT'
N. (CAP) (CAP is the communication array location that contains the character count for the PLAN literal that is the diagnostic text.) G. #B26

#B25. Is the program list to be given within the phrase; not provided in the communication array?
Y. 'PROGA,PROGB,...'
N. (CAP) (CAP is the communication array position that contains the character count of the literal text for the list of programs to be added to the pop-up list if the test fails.)

#B26. Is there an additional test to be made against this data item (communication array) location?
Y. G. #B13

#B27. Is this data item to be set as the result of evaluation of an equation?
N. G. #C1

#B28. Is the equation arithmetic; not logical?
Y. =EXPRESSION (Expression may contain matched parentheses to indicate the order of evaluation; the arithmetic operators +, -, *, /; arithmetic constants; symbolic operands that are symbols in the symbol table for this phrase.) G. #B35
N. :

#B29. Is this operand in the expression a relational expression (contains =, >,

' or <)?
Y. (
N. OPERAND G. #B34 (The operand may be any name in the current system symbol table.)

#B30. Is the left side of the relational an expression?
Y. EXPRESSION
N. SYMBOL or CONSTANT

#B31. RELATIONAL OPERATOR (>, -, =)

#B32. Is the right side of the relational an expression?
Y. EXPRESSION
N. SYMBOL or CONSTANT

#B33.)

#B34. Is there to be another operand in the expression?
Y. logical operator G. #B29

#B35. Is there to be another expression defined for this data item?
Y. G. #B28
N. , G. #C1

#B36. Is the data item to be assigned to a specific communication array location?
Y. (n) (n ranges from 1 to the limit of the communication array.) G. #B7
N. (

#B37. Is this operand of the symbolic storage assignment a constant?
Y. constant G. #B39

#B38. Is the symbolic operand to represent the location of the symbol; not the contents of the symbol?
Y. S'symbol
N. symbol

#B39. Is there another operand in the symbolic designation?
Y. arithmetic operator G. #B37
N.) G. #B7

#C1. Are there more data items to be defined?
N. G. #D1

#C2. Is the new data item to be assigned to the next-higher communication array location, and is the format (mode, user-exit control, and scale factor) identical to the data item just defined?
Y. G. #B27
N. G. #B2

#D1. Are there additional tests to be made or values to be set by expression evaluation that require looping or branching within the expressions?
N. G. #E1
Y. $0

#D2. Will this formula require transfer to/from elsewhere in the formula area?
Y. $n (n is in the range of 1 to 1,024.)

#D3. Is this element used to assign a data value to a data item or to execute a conditional branch?
Y. NAME (any valid name within the current symbol table)
N. G. #D21

#D4. Is the data item to be set as a result of a logical formula, not an arithmetic formula?
Y. :
N. =NAME G. #D20

#D5. Is this operand to be a relational expression?
N. NAME G. #D10
Y. (

#D5.1 Is the relational expression a test for logical TRUE?

Y. NAME=+) G. #D10

#D6. Is the relational expression a test for logical FALSE?
Y. NAME=-) G. #D10

#D7. Is the relational expression a test against an EBCDIC character mask?
Y. NAME = "MASK") (MASK contains only the characters to be tested.) G. #D10

#D8. Is the left side of the relational an expression; not a simple name?
Y. ARITHMETIC EXPRESSION
N. NAME

#D9. Is the relational operator =, >, or <?
Y. relational operator)
N. What else is desired?

#D10. Is there to be another operand in this expression?
Y. logical operator (not, and, or) G. #D5

#D11. Is the data item to be set as a condition of the result of the logical expression?
Y. ?
N. , G. #D15

#D12. If the logical expression is TRUE, is the data item to be set equal to the result of a logical expression?
Y. :logical expression G. #D15

#D13. If the logical expression is TRUE, is the data item to be set equal to the result of an arithmetic expression?
  Y. = arithmetic expression G. #D15

#D14. If the logical expression is TRUE, is processing to continue at a different formula number?
  Y. $n (n is any other formula number defined in this phrase formula area.)

#D15. Is processing to continue at the next formula without change to the value of the data item if the result of the logical expression is FALSE?
  Y. G. #D19
  N. !

#D16. If the result of the logical expression is FALSE, is the data item to be set equal to the result of a logical expression?
  Y. :logical expression G. #D19

#D17. If the logical expression is FALSE, is the data item to be set equal to the result of an arithmetic expression?
  Y. =arithmetic expression G. #D19

#D18. If the result of the logical expression is FALSE, is processing to continue at other than the next formula?
  Y. $n (n is any valid formula number)

#D19. Are there more formulas required?
  Y. , G. #D1
  N. G. #E1

#D20. Is the evaluation to be as a result of an arithmetic expression (are more operands required)?
  Y. Operator operand G. #D21
  N. G. #D19

#D21. Is the element to be a branching type element?
  Y. :$n G. #D19 (n is the number of the next formula to execute.)
  N. G. #D1

#E1. ; G. #A1

#Z1. Now, go back to the sections on "Language Definition", 4.3.0 and "Language Use", 4.2.0.

4.5.0 STANDARD PLAN COMMANDS

This section discusses the statements distributed as a standard part of the PLAN system. The only command that is a programmed portion of PLAN is ADD PHRASE. All other commands must be added to the system through use of ADD PHRASE. This section provides a discussion of the facility provided by a set of these phrases that are entered into the language definition dictionary (PFILE or DFJPFIL) as a part of the PLAN system generation. On System/360 DOS and OS PLAN, program names are prefixed with the characters "DFJ". The standard phrases shown represent standards for 1130 PLAN. On System/360 there may be minor variations. These variations may be noted in the phrase listings in the appropriate Operations Manual. Spacing within literals in the phrase definitions may not accurately represent that of the distribution commands.

4.5.1 ADD PHRASE

This command is added to the language definition dictionary when it is initialized.

ADD PHRASE: ADD PHRAS, (1)0, LEVEL0, I(-13)1, PROGRAM 'PHRAS, PHUDT';

ADD PHRASE may be altered to list all added phrases by adding PIDMP to the program list.

4.5.2 ALTER PHRASE

ALTER PHRASE provides the ability to delete an existing version of a phrase and replace it with a new copy. (For use, see "PLAN Language Definition", 4.3.0)

ADD PHRASE: ALTER PHRASE, I(1)-1,LEVEL0, I(-13)1,PROGRAM 'PHRAS, PHUDT, PHUDT';

ALTER PHRASE may be altered to list all altered commands by adding PIDMP to the program list.

4.5.3 DELETE PHRASE

DELETE PHRASE provides the ability to remove commands from the language definition dictionary. (For use, see "PLAN Language Definition", 4.3.0.)

ALTER PHRASE: DELETE PHRASE, (1)-1, LEVEL0, I(-13)1, PROGRAM'PHRAS,PHUDT';

DELETE PHRASE may be altered to list all deleted commands by adding PIDMP to the program list.

4.5.4 PLAN JOB

ALTER PHRASE: PLAN JOB, LEVEL 0, I(-1) FILE, SAVED, TO, LISTS, LB, LC, LD, ERASE, COMMON, MANAGED, NERM, DEVICE, I(1)SHORT-, LONG-, STACK-, IMMEDIATE-, DRIVE0, DFI-, PFI-,(-11)UMOD, I(-13)FORM0, $0 FORM:(LONG) ?=FORM+1, FORM:(IMM)?=FORM+2, FORM:(DFI)?= FORM+4, FORM:(PFI)?=FORM+8, TO=TO+DRIVE *2048;

PLAN JOB FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
SAVED STATEMENT FILE	FILE	(-1)	I	**NOTE		
INITIAL SAVED STATEMENT	SAVED	(-2)	I			
END SAVED STATEMENT	TO	(-3)	I			=TO+DRI*2048
DATA LIST A POINTER	LISTS	(-4)	I			
DATA LIST B POINTER	LB	(-5)	I			
DATA LIST C POINTER	LC	(-6)	I			
DATA LIST D POINTER	LD	(-7)	I			
ERASABLE COMMON POINTER	ERASE	(-8)	I			
SIZE OF COMMON	COMMON	(-9)	I			
SIZE OF MANAGED ARRAY	MANAGED	(-10)	I			
ERROR FILE QUEUE COUNT	NERM	(-11)	I			
DIAGNOSTIC MODULE(*NOTE)	UMOD	(-11)	LIT			
DIAGNOSTIC DEVICE	DEVICE	(-12)	I			
DIAGNOSTIC FORMAT	FORM	(-13)	I	0		:(LON)?=FORM+1 :(IMM)?=FORM+2 :(DFI)?=FORM+4 :(PFI)?=FORM+8
SHORT FORM INDICATOR	SHORT	(1)	LOG	FALSE		
LONG FORM INDICATOR	LONG	(2)	LOG	FALSE		
STACKED ERROR INDICATOR	STACK	(3)	LOG	FALSE		
IMMEDIATE ERROR IND.	IMM	(4)	LOG	FALSE		
SAVED STATEMENT DRIVE	DRIVE	(5)	I	0		
DYNAMIC FILE ERROR IND.	DFI	(6)	LOG	FALSE		
PERMANENT FILE ERROR INDICATOR	PFI	(7)	LOG	FALSE		

*NOTE: "UMOD" and "NERM" are mutually exclusive and may not be used together.
**NOTE: Default values are not provided because the 15 PLAN switch words are automatically reset as a result of the execution of any level 0 command.

PLAN JOB provides initialization functions for any PLAN run. This command, or one that provides the functions of this command, should be the first command processed when PLAN is invoked. The command meets the requirement that a level 0 phrase be the first phrase processed and sets the parameters controlled by the system switch words. System accounting functions may be conveniently facilitated by adding the name of an accounting module as a program list to this command. A sample of the command at execution time is:

```
PLAN JOB, MANAGED 200, ERASABLE 240,
COMMON 900, LISTS 30,60,200,209, SAVED 20
TO 30 FILE 3, DRIVE 2 SHORT, STACKED,
DEVICE 102;
```

The above example illustrates:

1. The setting of the managed array to a size of 200 PLAN words.
2. The establishing of the beginning of erasable COMMON at CAP 240.

3. The defining of the total size of COMMON to 900 32-bit words.

4. The defining of four CAP indices (30, 60, 200, 209) used in referencing a maximum of four data lists.

5. The designating of the short form of diagnostic.

6. The specification of the indicator to cause error stacking (STACKED).

7. The designation of the device upon which error messages are to be printed (DEVICE 102).

The following parameter discussions (see table above) give a breakdown of the PLAN JOB options:

1. SAVED STATEMENT FILE. This parameter defines the DYNAMIC file number (1-255) from which the next PLAN statement (a saved statement) is to be executed. The parameter will not be used if the next PLAN command is not a saved statement.

2. INITIAL SAVED STATEMENT. If the next PLAN statement is to come from a saved statement file, this parameter defines the number of the first statement that will be executed. If this parameter is specified, the FILE, DRIVE, and TO parameters should also be specified.

3. END SAVED STATEMENT. If saved PLAN statements are to be executed next, this parameter defines the highest-numbered saved statement that will be executed.

4. DATA LIST POINTER. This parameter is used to define the CAP indices for up to the maximum of four possible data lists. These data lists may be referenced by PSCAN for storing data, by PARGO and PARGI for transmitting data, and by user program modules.

5. LB. This parameter provides a direct pointer to the second of the data lists defined above.

6. LC. This parameter provides a direct pointer to the third of the data lists defined above.

7. LD. This parameter provides a direct pointer to the fourth of the data lists defined above.

8. ERASABLE COMMON. This parameter defines the communication array position (CAP) that is to be treated as the beginning of erasable COMMON. Erasable COMMON extends from the CAP position identified to the end of the communication array. This parameter must be set to some positive value within the range of the communication array in order for many of the standard PLAN commands to execute. The switch word is reset to 490 each time a level 0 command is encountered.

9. SIZE OF COMMON. This parameter defines the total size of COMMON (including communication array, switch words, and resident loader).

10. SIZE OF MANAGED ARRAY. This parameter defines the number of PLAN words that are to be managed according to the level structure of the commands to be processed. If this value is set to a positive integer and statements have a level assignment, the managed array save file must be present for the saving of data.

11. ERROR FILE QUEUE COUNT. If error diagnostics are to be written onto logical file 255 of logical drive 0 instead of directly to an output device, then this parameter will specify the maximum number of messages that are to be allowed on the file before the messages are to be written to the diagnostic device.

12. DIAGNOSTIC MODULE. This parameter is used to specify the name of a user-written module that is to process error conditions rather than using the normal system processing. Note that this option precludes the error queue option and is in lieu of writing the diagnostics onto the diagnostic device.

13. DIAGNOSTIC DEVICE. If a diagnostic module is not specified, this parameter specifies the sequential file device code (see "CALL IOCS", 5.11.5) upon which the diagnostics are to be printed. This switch word is reset to 100 each time a Level 0 command is encountered.

14. DIAGNOSTIC FORMAT. This parameter should not be referenced by a user. It is set as a result of use of items 15, 16, 17, 18, 20, and 21 (see "PLAN System Diagnostic Processing", 13.3.0).

15. SHORT. The word "SHORT" is specified if the short-form option is desired. Short-form diagnostics mean that the phrase being processed when the error is detected is not listed with the error. (See "PLAN System Diagnostic Processing", 13.0.0.)

16. LONG. This parameter is used to set the long-form diagnostic indicator.

Long-form diagnostics include the EBCDIC image of the phrase which caused the error, along with the diagnostic message (see "PLAN System Diagnostic Processing", 13.0.0).

17. STACK. This parameter sets the indicator to cause error stacking. In this mode of processing, errors are written to the output device only when the error module is scheduled by the PLAN loader or when the stack overflows. If the stack overflows, the checkpoint facility must be used to allow scheduling of the error module. (See "PLAN System Diagnostic Processing", 13.0.0.)

18. IMMEDIATE. This parameter sets the indicator to cause diagnostics to be written to the output device one-by-one as they are encountered. The checkpoint file and checkpoint programming must be available to function in the IMMEDIATE mode. (See "PLAN System Diagnostic Processing", 13.0.0.)

19. SAVED STATEMENT DRIVE. This parameter specifies the PLAN DYNAMIC drive number that will be used when the SAVE statements are processed.

20. DYNAMIC FILE ERROR INDICATOR. This parameter determines the PLAN system error procedures when an error is detected by the DYNAMIC FILE support subroutines.

21. PERMANENT FILE ERROR INDICATOR. This parameter determines the PLAN system error procedures when an error is detected by the PERMANENT FILE support subroutines (see "PLAN System Diagnostic Processing", 13.0.0).

4.5.5 SET LITERAL

SET LITERAL is the command used to define standard literals or tables for storage into a GDATA type file. The literals are maintained in a manner that makes them accessible to the subroutine PHIN.

```
SET LIT, NAME'PLITF', NUMBERn, 'LITERAL',
FILEj, DRIVEm;
```

```
ADD PHRASE: SET LITERAL, PROGRAM'PDIAG',
I(-8)M, I(M)FILE254, I(M+1) NAME0, I(M+4)
DRIVE0, I(M+5)NUMBER-*RA' UNDEFINED LITERAL
NUMBER', I(M+6)LITERAL0, (M+1)TEST-*
TA'UNDEFINED FILE NAME':
(NAME>0)&(NAME<9);
```

SET LITERAL FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
ERASABLE COMMON POINTER	M	-8	I			
LITERAL FILE NUMBER	FILE	M	I	254		
LITERAL FILE NAME	NAME	M+1	I	0		
LITERAL FILE DRIVE	DRIVE	M+4	I	0		
LITERAL NUMBER	NUMBER	M+5	I	FALSE	*RA	
LITERAL TEXT OR TABLE	LITERAL	M+6	I	0		
TEST FILE NAME	TEST	M+1		FALSE	*TA	:(NAME>0) & (NAME<9)

1. ERASABLE COMMON POINTER. This parameter points to the position within the communication array defined as erasable COMMON. This parameter (Switch Word 8) is normally set with the PLAN JOB command.

2. LITERAL FILE NUMBER. This parameter defines a number to be used to process the GDATA type literal file. The parameter should be defined only in situations where 254 would cause conflict with other processing on 1130 PLAN.

3. LITERAL FILE NAME. This parameter defines the name of the GDATA file in which literal processing occurs. Note that this parameter must be given. Otherwise, the check entry defined

under "test file name" will fail and the phrase will not be executed.

4. LITERAL FILE DRIVE. This parameter defines the PERMANENT drive on which literal file is located. Failure to provide this parameter results in the assumption that the file is on PERMANENT drive zero.

5. LITERAL NUMBER. This parameter defines the identification number for the literal to be processed. Note that failure to supply a literal number will result in a phrase abort error diagnostic. If the number is the same as an existing literal, the existing literal is removed from the file prior to adding the new literal.

6. LITERAL TEXT. This parameter provides the literal text for the literal to be added to the file. If this parameter is not provided (literal length zero) the existing literal of the same number is removed from the file. Note that tables can be maintained by this command and by the PHIN/PHOUT subroutines if the following convention is followed in setting up the data:

N	K	W_1	W_2	•	•	•	W_n

where:

N	Table or literal number
K	Count of bytes in table or literal character count
W_1	First 32-bit word in table or first four literal characters
W_n	Last 32-bit word in table

7. TEST FILE NAME. (See "Literal File Name" above).

4.5.6 LIST LITERAL

LIST LITERALS is a command that produces a listing of all literals maintained in a specified literal file.

ADD PHRASE: LIST LITERALS, LEVEL 1, PROGRAM'PLITL', I(1)FILE254, NAME-*A'LITERAL FILE NAME NOT DEFINED', I(5)DRIVE0, NOD100, (35)"NUMBER LENGTH TEXT OF PLAN LITERAL";

LIST LITERAL FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
LITERAL FILE NUMBER	FILE	1	I	254		
LITERAL FILE NAME	NAME	2	I	FALSE	*A	
LITERAL FILE DRIVE	DRIVE	5	I	0		
LITERAL OUTPUT DEVICE	NOD	6	I	100		

1. LITERAL FILE NUMBER. This parameter defines a number to be used to process the GDATA type literal file. The parameter should be defined only in situations where 254 would cause conflict with other processing on 1130 PLAN.

2. LITERAL FILE NAME. This parameter defines the name of the GDATA file in which literal processing occurs. Note that this parameter must be given. Otherwise, the check entry defined under "test file name" will fail and the phrase will not be executed.

3. LITERAL FILE DRIVE. This parameter defines the PERMANENT drive on which literal file is located. Failure to provide this parameter results in the assumption that the file is on PERMANENT drive zero.

4. LITERAL OUTPUT DEVICE. This parameter defines the output device that is to be used to list the literals. The standard parameter results in the use of the current PLAN output device.

4.5.7 COMMUNICATION ARRAY DUMPS

DUMP COMMON is a command that produces a hexadecimal printout of the communication array. Identical print lines are suppressed.

DUMP MANAGED is a command that produces a hexadecimal printout of the managed portion of the communication array. Identical print lines are suppressed.

DUMP NONMANAGED is a command that produces a hexadecimal printout of the nonmanaged portion of the communication array. Identical print lines are suppressed.

DUMP SWITCHES is a command that produces a hexadecimal printout of the PLAN switch words.

Note carefully that these are blank-level phrases. Any attempt to use them following a PLAN phrase abort error will result in the phrase being skipped.

ALTER PHRASE: DUMP SWITCHES, I(-8)M, I(M) NNN-2, (M+11)A"SWITCHES", "LENGTH", I(M+15) NOD100, PROGRAM'PCDMP';

ALTER PHRASE: DUMP COMMON, I(-8)M, I(M) NNN0, 'MANAGED ARRAY', 'NONMANAGED ARRAY', "SWITCHES", "LENGTH", I(M+15)NOD100, PROGRAM'PCDMP';

ALTER PHRASE: DUMP MANAGED, I(-8)M, I(M) NNN1, 'MANAGED ARRAY', "SWITCHES", "LENGTH", I(M+15)NOD100, PROGRAM 'PCDMP';

ALTER PHRASE: DUMP NONMANAGED, I(-8)M, I(M)NNN-1, (M+6)B'NONMANAGED ARRAY', "SWITCHES", "LENGTH", I(M+14)NOD100, PROGRAM 'PCDMP';

DUMP FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
ERASABLE COMMON DEFINITION	M	-8	I			
FUNCTION SWITCH	NNN	M	I			
DUMP COMMON				0		
DUMP SWITCHES				-2		
DUMP MANAGED				-1		
DUMP NONMANAGED				+1		
OUTPUT DEVICE	NOD	M+15	I	100		

1. ERASABLE COMMON DEFINITION. This parameter, a pointer to that portion of the communication array to be used as erasable COMMON, is normally set by the PLAN JOB command.

2. FUNCTION SWITCH. The appropriate value within the word (0, -2, 1, -1) distinguishes between DUMP, DUMP SWITCHES, DUMP MANAGED, and DUMP NONMANAGED functions, respectively.

3. OUTPUT DEVICE. This parameter defines the sequential device code to be used for output.

4.5.8 FILE DUMPS

ALTER PHRASE: DUMP DYNAMIC, I(-8)M, I(M) FILE255, I(M+2)START0, I(M+3)END0, I(M+4) DRIVE0, (M+5)A"DRIVE FILE LENGTH", (M+12) NAME' 'I(M+15)NOD100, 1, PROGRAM 'PFDMP';

ALTER PHRASE: DUMP PERMANENT, I(-8)M, I(M) FILE 255, I(M+2)START0, I(M+3)END0, I(M+4) DRIVE0, (M+5)A"DRIVE FILE LENGTH", (M+12) NAME' ', I(M+15)NOD100, 0, PROGRAM'PFDMP';

DUMP DYNAMIC is a command that produces a hexadecimal printout of the PLAN DYNAMIC file. Identical print lines are suppressed.

DUMP PERMANENT is a command that produces a hexadecimal printout of a PLAN PERMANENT file. Identical print lines are suppressed.

The limits of the dump are defined by the START and END operands. If these are omitted, the entire file is dumped.

Note carefully that these phrases are blank level, and will therefore be skipped if PLAN level recovery is invoked as the result of an error in a nonblank-level phrase.

DUMP PERMANENT FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
ERASABLE COMMON INDEX	M	-8	I			
FILE NUMBER	FILE	M	I	255		
START OF DUMP	START	M+2	I	0		
END OF DUMP	END	M+3	I	0		
DRIVE	DRIVE	M+4	I	0		
FILE NAME	NAME	M+12	LIT	BLANK		
OUTPUT DEVICE	NOD	M+15	I	100		
DUMP TYPE SWITCH		M+16	I	0,1		

1. ERASABLE COMMON INDEX. This index defines the location within the communication array known as ERASABLE COMMON. The index is normally set by the PLAN JOB command.

2. FILE NUMBER. This parameter defines the file number of the file that is to be dumped.

3. START OF DUMP. This parameter defines the number of the PLAN word within the file at which the file dump is to start.

4. END OF DUMP. This parameter defines the number of the last PLAN word within the file that is to be dumped. If the parameter is not given (parameter is set to zero), the full length of the file will be dumped.

5. DRIVE. This parameter defines the PLAN DYNAMIC or PERMANENT drive number on which the file to be dumped is located.

6. FILE NAME. This parameter defines the name of the file to be dumped.

7. OUTPUT DEVICE. This parameter defines the sequential device code that will be used for output.

8. DUMP TYPE SWITCH. This parameter determines whether a DYNAMIC or a PERMANENT file is to be dumped.

4.5.9 STATEMENT SAVE COMMANDS

ALTER PHRASE:SAVE,I(-1)SW,-1,I(-8)M,I(M) FILE0,I(M+1)DRI-1, $0 SW:(FIL>0)?=FIL, SW(3):(DRI>-1)&(DRI<5)?=DRI*2048;

SAVE is a command to allow saving of the PLAN statements that follow the SAVE command on a PLAN logical file. Each statement to be saved must be prefixed with a statement number. Saving of statements is terminated by (1) a SEND command, (2) any command that does not have a statement number, or (3) another SAVE command.

SAVE FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
	SW	-1	I			
		-2	I	-1		
ERASABLE COMMON POINTER	M	-8	I			
DYNAMIC FILE	FILE	M	I	0		
DYNAMIC DRIVE	DRIVE	M+1	I	-1		*NOTE

*NOTE: $0SW:(FIL>0)?=FIL,SW(3):(DRI>-1)&(DRI<5)?=DRI*2048

1. DYNAMIC FILE. This parameter defines the number of the PLAN DYNAMIC file on which the following statements are to be saved. If this parameter is omitted, the current file number in Switch Word 1 will be used.

2. DYNAMIC DRIVE. This parameter defines the number of the DYNAMIC drive on which the following PLAN statements are to be saved. If this parameter is omitted, the current file number in Switch Word 3 divided by 2048 will be used as the DYNAMIC drive indicator.

ALTER PHRASE: SEND;

SEND is a command used to terminate the saving of a series of PLAN statements.

ALTER PHRASE: EXECUTE, I(-1)SW,0, I(-8)M, I(M)FROM 0, I(M+1)TO 0, I(M+2)FILE 0, I(M+3)DRIVE -1,(M)F*TA'INVALID STATEMENT NUMBER OR DRIVE', $0 SW:(FIL>0)?=FIL, DRI:(DRI<0)?=SW(3)/2048-.5 !:$5, DRI:(DRI<0)?=0, $5 FRO:¬((DRI>-1)&(DRI<5)) ?=+, SW(3):(TO>0)?=DRI*2048+TO !=DRI*2048, SW(2):(FRO>0)?=FRO, FRO:(SW(2)>0);

EXECUTE is a command to initiate execution of commands saved in the specified statement save file.

EXECUTE FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
	SW	-1	I			
	M	-8	I			
FIRST COMMAND TO EXECUTE	FROM	M	I	0	*NOTE1	
LAST COMMAND EXECUTED	TO	M+1	I	0		
STATEMENT FILE NUMBER	FILE	M+2	I	0		
STATEMENT DRIVE NUMBER	DRIVE	M+3	I	-1		
PARAMETER CALCULATION						*NOTE2

*NOTE1 *TA'INVALID STATEMENT NUMBER OF DRIVE'

```
*NOTE2 $0SW:(FIL>0)?=FIL,
       DRI:(DRI<0)?=SW(3)/2048-.5!:$5,
       DRI:(DRI<0)?=0,
       $5FRO:¬((DRI>-1)&(DRI<5))?=+,
       SW(3):(TO>0)?=DRI*2048+TO!=DRI*2048,
       SW(2):(FRO>0)?=FRO,
       FRO:(SW(2)>0)
```

1. ERASABLE COMMON POINTER. This parameter defines the location within the communication array of ERASABLE COMMON. The pointer is normally set by the PLAN JOB command.

2. DYNAMIC DRIVE. This parameter defines the PLAN DYNAMIC drive number that is to be used to process SAVED statements. If this parameter is omitted, the current drive specified by Switch Word 3 divided by 2048 will be used.

3. DYNAMIC FILE. This parameter defines the PLAN DYNAMIC file number that is to be used to process SAVED statements. If this parameter is omitted, the current save file specified by Switch Word 1 will be used. /

4. FIRST SAVED. This parameter defines the number of the lowest-numbered SAVED statement to be executed. If this statement cannot be located, a PLAN diagnostic (DFJ172) will be produced.

5. LAST SAVED. This parameter defines the highest-numbered SAVED statement to be executed. Execution continues from the first SAVED statement identified through continually higher-numbered statements to the statement identified with this parameter. If this parameter is omitted, only the statement indicated by Switch Word 2 will be executed.

```
SAVE, FILE 2, DRIVE 3;
6 A;
9 B;
18 C;
SEND;
```

In the above example, when the SAVE command is encountered, all the numbered statements that follow (6, 9, 18) will be stored in the PLAN DYNAMIC file 2 on drive 3. This is known as explicit saving because the statements are stored for execution at a later time, and not executed now. (See "EXECUTE" command, discussed above.) Implicit saving, is utilized where statement storage and execution are accomplished as the statements are read.

It is important to note that execution of the SAVED statements will occur by statement numeric sequence, not by position within the input SAVE stream. For example, if a statement number 15 was placed after statement 18 in the stream, it would still be executed ahead of 18 if at a later time an EXECUTE command was encountered utilizing the parameters FROM 9 and TO 18.

4.5.10 PHRASE TABLE DUMP

ALTER PHRASE: DUMP PHRASES, I(500) SYSTEM1130, I(501)NOD100, I(503)LEVEL1, LEVEL1, (200)"CHECKSUM", "PHRASE NAME", "LEVEL TYPE-OBJECT", "ENTRY SIZE", "VERB", "SUBSCRIPT NAME VALUE RANGE INDEX", "EXIT PROGRAM LIST", "SYMBOL EXIT FORMAT SCALE SUBSCRIPT EXPRESSION", "PROGRAM LIST", "TEST LOCATION ACTION", "LITERAL , LIST",SUBSCRIPT, "LOCATION MODE FACTOR EXPRESSION", (510)-*TP'CON DUM PHR I(504)DRIO';

(1130 STANDARD PHRASE)
ALTER PHRASE: CON DUMP PHRASES, (281)"INTERPRETIVE EXPRESSIONS", "VERB PROGRAMS", END OF PHRASE TABLE DUMP", PROGRAM'PTDMP', (505)"PFILE";

DUMP PHRASES is a command that produces a tabulation of the phrases that exist within PFILE.

CONTINUE DUMP PHRASES is the continuation of the DUMP PHRASES command and should not be invoked by itself.

The module PTDMP produces the phrase dump. It requires XACES, XTRAC, XPRNT, and XBIT, which are called as subroutines. PTDP1, PTDP2, PTDP3, PTDP5, and PTDP6 are also required. They are called as monitor locals on 1130 PLAN and are loaded as PLAN system local modules on OS and DOS PLAN. These modules are special purpose programs that have no use in any other environment.

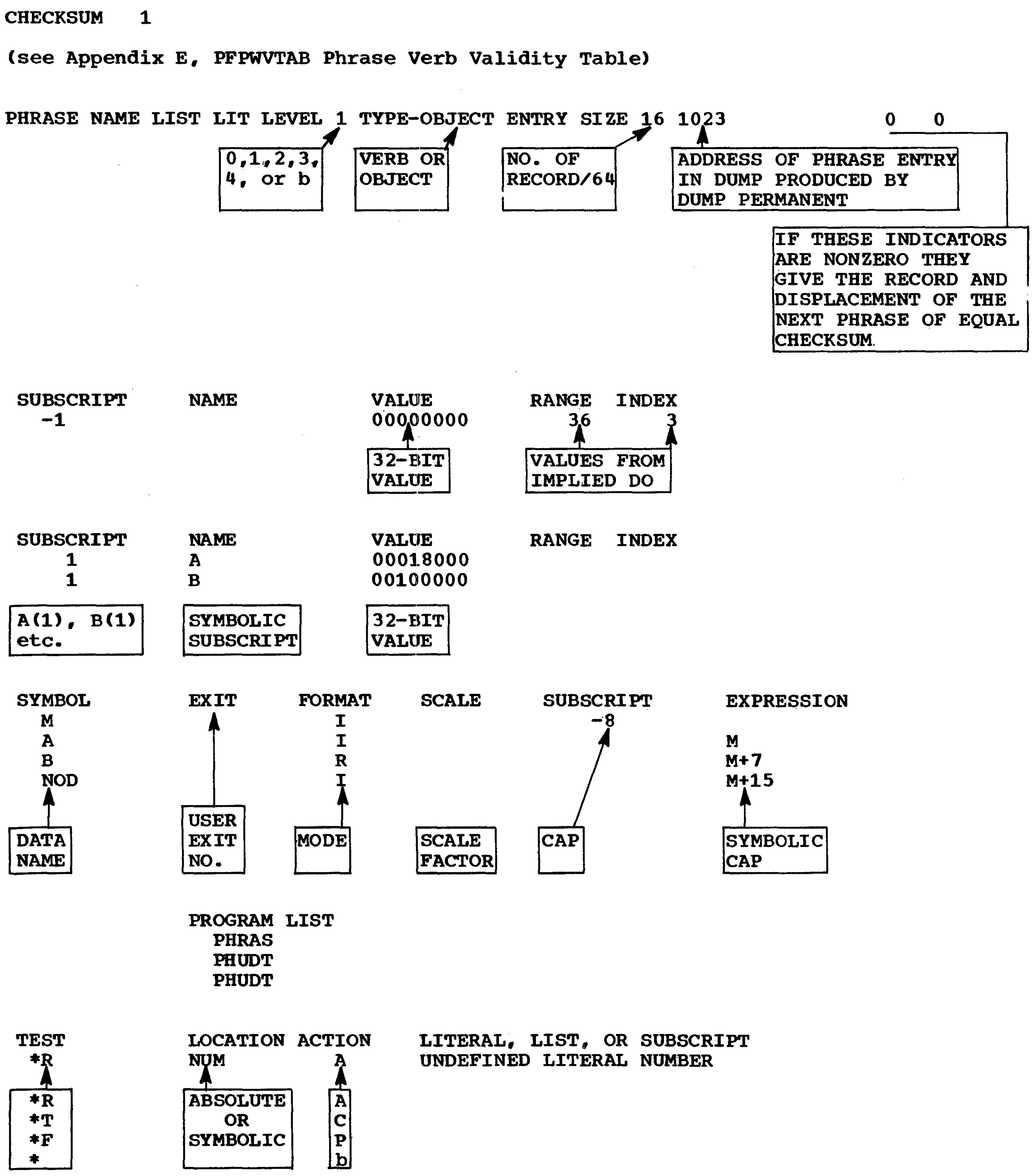
DUMP PHRASES FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
SYSTEM DESIGNATION	SYSTEM	500	I	1130, 360		
OUTPUT DEVICE	DEVICE	501	I	100		
PRINTOUT LEVEL	LEVEL	503	I	1		

1. SYSTEM DESIGNATION. This parameter defines the system for which the PFILE (PLAN language dictionary) is being dumped. The phrase for the appropriate system contains the necessary standard value so that the user should never be required to specify this parameter.

2. OUTPUT DEVICE. This parameter defines the sequential device code to be used for output.

3. PRINTOUT LEVEL. This parameter defines the complexity of the phrase listing to be produced. Each higher level incorporates all items of the lower levels.

The items listed below represent information that is produced at the various printout levels. Figure 10 shows sample lines from the dump. Enclosed items are explanatory notes about the sample output lines. It is strongly recommended that the reader make a diligent attempt to correlate the phrases as defined in this section with the listing produced with the DUMP PHR, LEVEL 6; command through use of Figure 10.



```
CHECKSUM    1

(see Appendix E, PFPWVTAB Phrase Verb Validity Table)

PHRASE NAME LIST LIT LEVEL 1 TYPE-OBJECT ENTRY SIZE 16 1023          0   0


 SUBSCRIPT     NAME          VALUE        RANGE   INDEX
   -1                        00000000       36        3


 SUBSCRIPT     NAME          VALUE        RANGE   INDEX
     1         A             00018000
     1         B             00100000


 SYMBOL        EXIT    FORMAT   SCALE    SUBSCRIPT      EXPRESSION
   M                      I                  -8
   A                      I                              M
   B                      R                              M+7
   NOD                    I                              M+15


               PROGRAM LIST
                 PHRAS
                 PHUDT
                 PHUDT


 TEST          LOCATION ACTION   LITERAL, LIST, OR SUBSCRIPT
  *R           NUM         A     UNDEFINED LITERAL NUMBER
```

Figure 10. Phrase table dump explanation

LEVEL	ITEM LISTED
0,1	Phrase name
	Phrase level
	Type (object or verb)
	Number of internal records (80-bit on 1130, 64-bit on System/360) required for phrase
	PFILE ADDRESS of phrase entry
	Chained phrase indicator (0 0 means no chained phrase)
	Checksum of phrase
2	Initialization (Default values)
	Subscript
	Name
	Value
	Range
	Index
3	Symbol Table
	Symbol
	User-exit number
	Format
	Scale factor
	Subscript
	Subscript expression
4	Program lists
5	Check entries
	Test
	Location
	Action
	Literal, list, or subscript
6	Expressions
	Data area
	Formula area

4.5.11 ERROR LISTING

```
ALTER PHRASE:  DUMP ERRORS, PRO'PEDMP';
```

DUMP ERRORS is a command that causes all diagnostics in the error queue file to be listed on the PLAN diagnostic device.

4.5.12 IOCS CONTROL ON 1130

```
ALTER PHRASE:   IOCS, LEVEL0, PRO'PIOCS',
I(1)INPUT1131, LIST1131, I(-8)1,  $0INPUT:(
INPUT=1131)?=3,       INPUT:(INPUT=2501)?=2,
INPUT:(INPUT=1442)?=1,    LIST:(LIST=1131)?=
103,    LIST:(LIST=1403)?=102,    LIST:(LIST=
1132)?=101,  INPUT:(INPUT=1)|  (INPUT=2)  |
(INPUT=3)?=INPUT!=0,     LIST:(LIST=101)   |
(LIST=102) | (LIST=103)?=LIST!=100;
```

IOCS FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
ERASABLE COMMON POINTER		-8	I	1		
PLAN INPUT DEVICE	INPUT	1	I	1131		
PLAN OUTPUT DEVICE	LIST	2	I	1131		

IOCS is a level 0 command on 1130 PLAN that allows the PLAN input and output devices to be altered.

1. INPUT. This parameter must specify the input unit that is to be used for input of the following PLAN commands. Valid arguments are 2501, 1442, and 1131.
2. LIST. This parameter must specify the output unit that is to be used for output of following PLAN diagnostics. Valid arguments are 1132, 1403, or 1131.

```
ALTER PHRASE:   CARD, I(-8)M, I(M)INPUT0,
I(M+1)LIST100, PROGRAM'PIOCS',  $0INP:(INP=
2501)?=2, INP:(INP=1442)?=1, LIS:(LIS=1403)
?=102,LIS:(LIS=1132)?=101,     INP:(INP=1)|
(INP=2)?INP!=0,              LIS:(LIS=101)|
(LIS=102)?=LIS!=100;
```

CARD FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
ERASABLE COMMON POINTER	M	-8	I			
PLAN INPUT DEVICE	INPUT	M	I	0		
PLAN OUTPUT DEVICE	LIST	M+1	I	100		

CARD is a blank-level command on 1130 PLAN that allows changing input to either card reader and/or output to either line printer.

1. INPUT. This parameter must specify the card reader from which the next PLAN input is to be read. Valid arguments are 2501 or 1442.
2. LIST. This parameter must specify the pointer on which the next PLAN diagnostic is to be printed. Valid arguments are 1403 or 1132.

ALTER PHRASE: TYPE, I(-8)M, I(M)N3, 103, PROGRAM'PIOCS';

TYPE FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSIONS
ERASABLE COMMON POINTER	M	-8	I			
PLAN INPUT DEVICE	N	M	I	3		
PLAN OUTPUT DEVICE		M+1	I	103		

TYPE is a blank-level command on 1130 PLAN that sets the console typewriter/printer as the input/output device from/to which the next PLAN input/output is to be read/written.

ALTER PHRASE: LON, LEVEL 1;

LON is a level 1 command that has no function other than to fulfill the requirement that a level 1 command be processed.

4.5.13 PAGE LENGTH DEFINITION (OS/DOS ONLY)

ALTER PHRASE: SET PAGE LENGTH, I(-8)M, I(M)PGL60, I(M+1)NOD100, PROGRAM'DFJPLENG';

SET PAGE LENGTH is a blank-level command that allows the user to specify the number of printed lines per page on a sequential device that is to contain printed output.

SET PAGE LENGTH FUNCTION	NAME	CAP	MODE	DEFAULT VALUES	CHECKING RULES	EXPRESSION
ERASABLE COMMON POINTER	M	-8	I			
PAGE LENGTH	PGL	M	I	60		
OUTPUT DEVICE	NOD	M+1	I	100		

1. PAGE LENGTH. This parameter defines the number of lines to be printed on a page before a logical EOF is generated and an automatic eject (skip to 1) is effected.
2. OUTPUT DEVICE. This parameter defines the sequential device code with which the PAGE LENGTH operand is to be associated.

ALTER PHRASE: INPUT, I(-8)M, I(M)NOD1,0, LEVEL 1, PROGRAM'DFJPIOCS';

INPUT is a command that may be issued to change the device that is assigned as the standard PLAN input device.

1. NOD. This parameter defines the number of the device that is to be used for PLAN input. The output device is not changed.

ALTER PHRASE: OUTPUT, I(-8)M, I(M)A0, I(M+1)NOD101, LEVEL1, PROGRAM'DFJPIOCS';

OUTPUT is a command that may be used to change the device that is assigned as the standard PLAN output device.

1. NOD. This parameter defines the number of the device that is to be used for PLAN output. The input device is not changed.

4.5.14 SPECIAL PURPOSE OS PHRASES

ALTER PHRASE: CREATE LOADER ENTRIES, PROGRAM'DFJLLIST';

CREATE LOADER ENTRIES is a command that gives OS PLAN the capability of referencing the RAM or LINKPAC areas.

The general format of this command is:

```
    CREATE LOADER ENTRIES:  (NAME1,....);
```

where NAME1,... is a load module name that is to be loaded into the partition via the LOAD macro and be made available as entry points for the execution of any loader call. This allows programs in the LINKPAC or RAM areas to be objects of a CALL LOCAL. The names specified in the LIST must be in the JOBLIB PDS or LINKLIB PDS.

The maximum number of names in the list is 75. Use of this command destroys any entries defined by previous use of the command.

Programs that reference blank COMMON may not be operands of this command.

ALTER PHRASE: CREATE CORE DIRECTORY, PROGRAM 'DFJCRDIR';

CREATE CORE DIRECTORY is a command that allows the user to build an in-core PDS directory of names of frequently loaded modules.

```
    CREATE CORE DIRECTORY:  (NAME1,...);
```

NAME1,... is a load module name that is placed in the in-core PDS directory to decrease load time for those modules. The names in the list must be entries in the PLANLIB PDS.

Use of this command will replace the previous directory. The maximum number of entries is 75 names.

5.0.0 PLAN SUBROUTINE SUPPORT

This section describes the functions of the various PLAN subroutines that are available for use by the application programmer.

Sections 5.1.0 through 5.10.0 list the subroutines and their specifications. Sections 5.11.0 through 5.11.11 provide the details of usage.

5.1.0 PLAN LOADER SUBROUTINES

LCHEX	This subroutine allows a user to modify the PLAN pop-up list. The current program in execution is saved for future reentry at the next executable statement when an asterisk (*) is found in the pop-up list.
LEX	This subroutine allows a user to modify the PLAN pop-up list. Transfer to PLAN then occurs to load the first (top) program defined in the pop-up list.
LIST	This subroutine allows a user to modify the PLAN pop-up list. Processing continues at the next executable statement in the calling program.
LISTB	This subroutine allows a user to add a program name to the bottom of the PLAN pop-up list. Processing continues at the next executable statement in the calling program.
LNRET	This subroutine breaks the chain of returns normally followed within PLAN LOCAL processing.
LOCAL	This subroutine allows a user to modify the PLAN pop-up list. The top program in the pop-up list is loaded and control passes to the program. The program must coexist in memory with the calling module because the calling module will be reentered at a later time. The program is not loaded if already in core. Note that the calling module remains in core with the called module; this is true of a nest of locals as well.
LREPT	This subroutine allows the re-execution of the last command processed.
LRET	This subroutine is the normal exit from a logic module. It does not modify the PLAN pop-up list. It exits to PLAN to load and transfer to the top program in the pop-up load list. If the pop-up list is empty and saved statements are not being executed, a new command is processed. If the program executing a CALL LRET was called by a CALL LOCAL, control is returned to the next executable statement after the CALL LOCAL.

5.2.0 PLAN I/O CONTROL

IOCS	This subroutine allows redefinition of the PLAN system parameters. The command input device and diagnostic print device may be shifted among supported I/O devices.

5.3.0 PLAN ERROR PROCESSING

The following six subroutines allow application logic modules to generate and process diagnostic messages through the use of the PLAN system error processing module (PERRS). The format of the diagnostic produced is identical to that produced by PLAN. The diagnostic literal is user-supplied.

ERLST	This subroutine causes all diagnostics that are in the PLAN error queue file to be printed on the PLAN system diagnostic device. Processing of the current phrase (including programs in pop-up list) is terminated. This subroutine provides the capability required for post-list error processing.
ERRAT	This subroutine interface to the PLAN error module PERRS returns to the next executable statement in the calling program. PLAN will load any remaining programs in the pop-up list. However, the next time that PSCAN is entered, PLAN level error recovery is initiated.
ERRET	Processing continues at the next executable statement following the call to the subroutine.
ERREX	This subroutine interface to the PLAN error module PERRS does not return to the calling program.

	PLAN is entered to load any remaining programs in the pop-up list.
ERROR	Processing does not return to the calling module and the PLAN level error recovery is initiated.
EWRIT	This subroutine allows the user to write messages into the PLAN error file (file 255, DYNAMIC drive 0) in a format acceptable for processing by CALL ERLST.

5.4.0 PERMANENT FILE SUPPORT

GDATA, GDAT1	These subroutines perform the file open function for PERMANENT fixed size files established outside of PLAN. The call places file location pointers in the user-defined file control block.
RDATA, RDAT1	These subroutines provide for transfer of information from a PERMANENT file location on disk to memory. Records may contain any variable number of 32-bit or 16-bit words.
WDATA, WDAT1	These subroutines provide for transfer of information from memory to a PERMANENT file location. Records may contain any desired number of words. The file is addressed by the number of words displacement from the beginning of the file.

5.5.0 DYNAMIC FILE SUPPORT

FIND, FINDL, PFND1	These subroutines perform the open function for DYNAMIC files. DYNAMIC files are established when needed (they may be permanent) as defined by execution-time logic. Disk space is assigned to the file in modular segments as required by the file. Transfer is by groups of any desired number of 32-bit or 16-bit words to or from any desired displacement within the file. PLAN DYNAMIC files may be assigned to any of eight (five on the 1130) drives. Priority may be assigned to a DYNAMIC file to allow orderly release of files if insufficient file space is available.
PFSPC	This subroutine provides the facility to request verification of the availability of a block of storage for assignment to a DYNAMIC file at a designated priority.
RELES, PREL1	These subroutines release space held by a DYNAMIC file to the pool of available disk space. RELES performs the opposite function of the FIND routine.
READ, PRED1	These subroutines transfer data from a DYNAMIC file to memory.
WRITE, PWRT1	These subroutines transfer information from memory to a DYNAMIC file. Space is automatically allocated if a write requires more space than the current file contains.

5.6.0 COMMAND RETRIEVAL AND EXECUTION

INPUT	This subroutine transfers the EBCDIC representation of the last command processed and the length of the command in characters into memory to allow processing or printing of the command.
PUSH	This subroutine provides the ability to execute commands from memory.

5.7.0 LOGICAL FUNCTIONS

FALSE	This subroutine sets a word in memory to the value of logical FALSE.
NDEF	This function subroutine allows testing of any location for the PLAN logical functions FALSE, TRUE, or REAL (nonlogical).
TRUE	This subroutine sets a word in memory to the value of logical TRUE.

5.8.0 SORT/MERGE CONTROL

GMERG	(OS/DOS only) This subroutine performs the initialization functions for the PLAN system module DFJGMRGA. The module issuing the CALL GMERG is reentered when the merge is completed.
GSORT	(OS/DOS only) This subroutine performs the initialization functions

for the PLAN system modules DFJGSRTA and DFJGSRTB. The module issuing the CALL GSORT is reentered when the sort is complete.

PSORT	This subroutine performs the initialization functions for the PLAN system module PSRTA. It causes the managed array to be saved and results in a checkpoint (CALL LCHEX) exit. The module issuing the CALL PSORT is reentered when the sort is completed.
PMERG	This subroutine performs initialization functions for the PLAN system module PMRGA. The module issuing the CALL PMERG is reentered when the merge is completed.

5.9.0 SEQUENTIAL FILE CONTROL

PLINP	This subroutine provides the input processing for overlapped, buffered transfer from a PLAN-supported input device to the system buffer.
PLOUT	This subroutine provides the output processing for buffered, overlapped transfer from the system buffer to a PLAN-supported output device.
PIOC	This function subroutine allows a user to test a device status for busy.
PBUSY	This subroutine tests all PLAN devices controlled by PLINP and PLOUT. PBUSY returns only when none are found to be busy.
PSBFA, PSBFB, PSBFC, PSBFD, PSBFE	These subroutines provide single-buffering assignment (for 1130 PLAN) for devices specified for use by the PLINP and PLOUT routines.
PDBFA, PDBFB, PDBFC, PDBFD, PDBFE	These subroutines provide double-buffering assignment (for 1130 PLAN) for devices specified for use by the PLINP and PLOUT routines.
PBFTR	This subroutine allows direct transfer between PLAN buffers, facilitating input followed by output of the same data where intervening formatting and/or processing is not required.
PEOF	This function subroutine allows testing of end-of-file conditions generated as a result of CALL PLINP or CALL PLOUT.
PCCTL	This subroutine provides device control functions such as carriage skipping and spacing and stacker selection.
PAIN, PAOUT	These subroutines provide for transfer from/to the PLAN system buffers to/from user-designated storage with variable literal (A) format control.
PIIN, PIOUT	These subroutines provide for transfer from/to the PLAN system buffers to/from user-designated storage with integer (I) format control.
PFIN, PFOUT	These subroutines provide for transfer from/to the PLAN system buffers to/from user-designated storage with floating-point (F) format control.
PEOUT	This subroutine provides for transfer of data from user-designated storage to the appropriate PLAN system buffer in exponential floating-point (E) format.

5.10.0 ARRAY AND DATA MANIPULATION

PHIN	This subroutine provides for transfer of literal data to memory. The literal file is maintained on disk by the module PDIAG through control of the PLAN command SET LITERAL.
PHOUT	This subroutine transfers literal data to a file from memory. PDIAG requires this subroutine.
PARGO	This subroutine provides transfer of data from a user array to the PLAN communication array.
PARGI	This subroutine provides the ability to move data lists from the PLAN communication array to a user array.
GTVAL, STVAL	These subroutines allow easy, efficient transmission of arrays to and from any location in storage.
BREAK	This subroutine spreads the four bytes of a 32-bit word into the low-order position (rightmost eight bits) of four words of an integer array.
PPACK	This subroutine masks the low-order byte of an integer word into any byte position of a 32-bit character array.

PUNPK	This subroutine moves any byte position of a character array into the low-order byte position of an integer word. The high-order bits are set to zero.
PCOMP	This function subroutine performs a logical comparison of one 32-bit array with a second array and sets the floating-point FORTRAN function indicator.
PHTOE	This subroutine converts hexadecimal arrays to EBCDIC arrays. The EBCDIC array produced occupies twice as many words as the hexadecimal array.
PBTST	This subroutine allows for the testing or setting of any bit or bits (0-31) within a 32-bit word. It also provides a test or extract under mask.

5.11.0 PLAN SUBROUTINE USE

This section provides a detailed description of calling sequences and performance characteristics of PLAN system subroutines. The calling sequences are shown as FORTRAN statements. Use of the subroutines by modules programmed in other languages (symbolic and assembler) must be programmed according to the FORTRAN conventions. Specific differences in the action/use of these routines between various versions of PLAN are documented in the appendices of this manual.

5.11.1 PROGRAM LINKAGE ROUTINES

```
LOADER SUBROUTINES

CALL LIST(N,L)
CALL LISTB(2,L)
CALL LEX(N,L)
CALL LCHEX(N,L)
CALL LOCAL(N,L)
CALL LRET
CALL LNRET
CALL LREPT

    N  the count of 32-bit words to move
    L  the user list array

CALL LIST (1,0)
CALL LEX (6,ARRAY(6))
```

The subroutines defined below allow a user to communicate with the PLAN loader and manipulate the pop-up list. Each subroutine in this group is named with an initial "L" to indicate its special relationship with the PLAN loader. Every PLAN logic module normally exits to the PLAN loader through one of these subroutines.

The linkage CALL LRET returns directly to the PLAN loader without modification to the pop-up list. If the pop-up list is not empty, the program named at the top of the list will be executed next. If the pop-up list is empty (0), PSCAN is loaded to process a new command. Exit from a module via CALL LRET provides a set of modules whose linkage sequence is governed by the problem description.

For creating special compile-time controlled linkages, other loader subroutines are useful. In the following examples,

N is the number of program identification words (32-bit words) to be moved to/from the pop-up list. A program name occupies two 32-bit words. Thus, a list of three program names requires that N be defined as 6. N may be an integer constant, or a subscripted or nonsubscripted integer variable.

L is the location of the array in the problem program (or communication array) that holds the words to be moved.

Positive values of N cause movement from array L to the pop-up list. Negative values of N cause movement from the pop-up list to array L and remove the moved items from the pop-up list. If the absolute value of N when N is negative is greater than the number of 32-bit words in the list, a numeric zero is transmitted to array L following the last item in the list. Zero as a value of N causes no movement. If values are moved L(1) becomes the top of the pop-up list. Additions push the old list down to position (N+1) of the pop-up list. Deletions pull the value at (N+1) up to the top.

When N is positive, the input array is scanned from end-to-start, accessing and placing in the pop-up list a 64-bit word at a time.

If a numeric zero is encountered in bits 0-31 of the 64-bit word containing a program name, the pop-up list is cleared. If the absolute value of N is odd, it is incremented by one.

To avoid reprogramming, parameters N and L should be symbolic, equivalenced to communication array locations. An argument list of (1,0) in the following calls destroys the current contents of the pop-up list whereas (0,0) leaves it unchanged.

Functions of the subroutine calls are:

CALL LIST(N,L) manipulates the pop-up list and returns to the next statement following the call.

CALL LISTB(2,L) places a single program name at the bottom of the pop-up list and returns to the next statement following the call. Note that 2 will always be the value of N for the LISTB subroutine.

CALL LEX(N,L) manipulates the load list and then loads and branches to the next program in the pop-up list.

CALL LCHEX(N,L) manipulates the pop-up list, saves the current program for later reentry, then exits to the loader. COMMON is not affected. No test is made to protect the PLAN communication array from overlay by the next module, so the module issuing the CALL LCHEX may have to save and restore parts of COMMON if the checkpoint load will overlay it. The saved program is reentered at the next instruction following the CALL LCHEX (N,L) when a left-justified asterisk is found in the pop-up list. A checkpoint may not be carried beyond phrase boundaries. In other words, if an asterisk has not been encountered in the pop-up list before the list is emptied (PSCAN is reloaded) a PLAN phrase abort (level error recovery) is initiated.

CALL LOCAL(N,L) manipulates the pop-up list, then loads and enters the next program in the pop-up list. The address of the instruction following the CALL LOCAL (N,L) is saved for a return from the LOCAL program when a CALL LRET is issued. Both the local program and the calling program will coexist in memory at the same time. Additional information on the use of local programs is contained in the appendices of this manual.

CALL LRET is the normal exit from a logic module. It does not modify the pop-up list. It exits to PLAN to load the program named at the top of the pop-up list. If the list is empty and saved statements are not being executed, a new command is processed. If the program executing a CALL LRET was called by a CALL LOCAL, control is returned to the calling program at the next executable statement following the CALL LOCAL.

CALL LNRET specifies that a normal return (CALL LRET) is not anticipated. CALL LNRET provides a means of canceling all 'LOCAL' processing in progress. CALL LNRET informs PLAN that the calling module will not return to the module that called it. A CALL LRET issued by a module after a CALL LNRET causes a return to the PLAN loader. Any OS/360 module containing a CALL LNRET may not be terminated with a RETURN statement.

CALL LREPT repeats processing of the current command. The pop-up list is not cleared by execution of CALL LREPT, but the repeated command is processed before the programs are loaded.

The following example (shown with IBM 1130 control cards) illustrates commands and programming that will perform the following functions:

Step 1 represents the 1130 System FORTRAN compilation of program "M0725".

Step 2 is the loading of the compiled module into core image (residing on disk). PLAN can only retrieve modules stored in core image.

Step 3 is the execution of the PLAN system, where program "M0725" is loaded and executed first. Program "M0788" is executed out of line by the calling of the LCHEX (CALL LCHEX(4,PLIST)), which allows the user to modify the pop-up list. The current program in execution (M0725) is saved for future reentry at the next executable statement when an asterisk is found in the pop-up list. After M0725 is reentered, a call of subroutine LEX is encountered which will manipulate the pop-up list and then load and execute to the next program named in the pop-up list. The pop-up list will then be loaded with the program names PROGA, PROGB, PROGC, and PROGD, with PROGA residing on top.

```
STEP1
// JOB
// FOR
      DIMENSION PLIST(4)
      COMMON L(625), LS(15), M(255)
      EQUIVALENCE (N,M(20)), (ABCD,M(21))
      DATA PLIST/'M078', '8', '*   '/
        .
        .
      CALL LCHEX (4,PLIST)
        .
        .
      CALL LEX (N,ABCD)
      END

STEP2
// DUP
*STORECI    WS   UA   M0725

STEP3
// XEQ PLAN

ADD   PHRASE:   LOAD   PROGRAM,   I(20)NO,
PRO'M0725';

LOAD  PROGRAM,    N8   "PROGA""PROGB""PROGC"
"PROGD";
        .
        .
      (REMAINING PLAN INPUT)
```

5.11.2 DYNAMIC FILE SUPPORT

```
DYNAMIC FILE ROUTINES

CALL FIND(ID,NPRI,NALLO,NDR)
CALL FINDL(ID,0,0,NDR)
CALL READ(ID,KDIS,KOUNT,KORE)
CALL WRITE(ID,KDIS,KOUNT,KORE)
CALL RELES(ID,0,NSQZ,NDR)

  ID     File control block
  NPRI   File priority (0,1,2,3,4)
  NALLO  Initial allocation requirement
  NDR    DYNAMIC drive code (0-7 or 0-4
         the 1130)
  KDIS   File displacement
  KOUNT  Words (32-bit) to transfer
  KORE   User array
  NSQZ   Words not to be released
```

```
C     WRITE ARRAY 1-700 AND CHECK
      DIMENSION NA(100),ID(2)
      COMMON L(625),LS(15),CA(510)
C     SET FILE CONTROL BLOCK
      ID(1)=27
C     OPEN FILE
      CALL FIND (ID,2,700,3)
C     INITIALIZE ARRAY
      DO 5 I=1,100
    5 NA(I)=I
C     WRITE 7-100 WORD RECORDS
      DO 10 I=1,7
      CALL WRITE (ID,ID(2),100,NA)
C     UPDATE ARRAY BY 100
      DO 15 J=1,100
   15 NA(J)=NA(J)+100
   10 CONTINUE
C     READ BACK 100 WORD GROUPS
      DO 25 I=1,7
      CALL READ (ID,(I-1)*100,100,NA)
      DO 25 J=1,100
C     CHECK FOR VALID NUMBER
      IF(NA(J)-(J+(I-1)*100)) 20,25,20
C     ERROR
   20 PAUSE 7
   25 CONTINUE
C     RELEASE TOTAL FILE
      CALL RELES(ID,0,0,3)
C     RETURN TO PLAN
      CALL LRET
        •
        •
        •
```

The subroutines defined below provide the DYNAMIC support for processing variable-length, discretely addressable disk data sets. All parameters associated with formation of the data sets are definable at execution time rather than at compile time. Each word in the file is discretely addressable. This allows disk to be treated as an out-of-core array.

The physical location of a data set is made available during execution by the following FORTRAN (or equivalent) call:

```
    CALL FIND(ID,NPRI,NALLO,NDR)
```

The parameters of the call have the following meanings:

ID identifies the first of a two-word file control block. Each data set (file) has a separate file control block. If the file control block is in COMMON (communication array), one CALL FIND can satisfy a series of programs and result in a saving of disk access time. If the file control block is not in COMMON, each program must issue its own CALL FIND for the file. The value stored in ID(1) by the calling program must be an integer from 1 to 255. This is the DYNAMIC file number. DYNAMIC file number 255 on DYNAMIC drive 0 is used by PLAN for error message processing. DYNAMIC files 201 to 255 on DYNAMIC drive 0 are reserved for PLAN utilities. The remaining numbers (1-200) can be used to uniquely identify user's DYNAMIC files. After the CALL FIND has been executed, ID(1) contains a coded pointer to the beginning of the data set, and ID(2) contains the current file length. ID(2) contains a zero if this is a newly established file.

DYNAMIC files are not expected to reside on more than one DYNAMIC drive. The programmer may create a sequence of data sets crossing DYNAMIC file boundaries, assuming that more than one DYNAMIC drive is available.

FIND treats ID(1) modulo 256. This means that a FIND issued to a file control block, that shows an open or closed condition, will result in the true length of the DYNAMIC (current status) being placed in ID(2). The remaining parameters for CALL FIND are used only at the time a new DYNAMIC file is opened but they must always be present.

NPRI assigns a retention priority to the DYNAMIC file. Zero sets the priority equal to the level of the command currently being processed. Priority 1 indicates that the DYNAMIC files cannot be automatically released by the system. Priorities 2, 3, and 4 are successively inferior levels of temporary data and are automatically released whenever a command of higher level (lower-numbered) is processed.

A DYNAMIC file retains the priority defined in the initial CALL FIND and is unchanged regardless of the specification in subsequent CALL FIND's, until released (CALL RELES).

PLAN will automatically release lower-priority files to create space for higher-priority files if insufficient space is available for the required higher-priority file. If a file is automatically released, the file control blocks opened for that file are not marked as closed.

NALLO, when given as a value other than 0, is used to optimize file space allocation. Normally, space is allocated to a file incrementally, only as needed. It is more advantageous to allocate space for an entire DYNAMIC file at one time if the requirement is known. NALLO provides an estimate of the expected file size and is used to calculate the number of words in the initial allocation according to the following formula:

```
    NWA = ((NALLO-1)/NSA+1) * NSA
```

where NWA is the number of 32-bit words actually allocated. NSA is the number of PLAN words in a standard unit allocation (see Appendix A, B, or C for discussions of this parameter).

If the initial allocation request is for 1000 32-bit words and the standard unit allocation is NSA=628, then 1256 words would actually be allocated.

NALLO is ignored if the file already exists. NALLO has no effect on the value of the current file size maintained in ID(2). NALLO is ignored in incremental allocations. Each additional allocation includes only one standard unit allocation.

NDR defines the DYNAMIC drive on which the DYNAMIC file is to reside. The parameter may range from 0 to 7, except as limited by the hardware configuration. This parameter specifies a logical drive in the 1130 system and is limited to the range of 0-4. In the 1130, digits to the left of the units digit in the indicator are used for verification of pack label identification as defined in Appendix A (8.6.0) of this manual.

The FINDL subroutine provides a check for the current existence of a file. Space is not allocated for the file if it does not exist. If the file does exist, the file is opened and the current true file size is placed in ID(2). If the file does not exist, the file is not opened and ID(2) is set to zero. If an error is found (for example, the drive code is invalid), the file is not opened and ID(2) is set to an error code as defined for DYNAMIC files near the end of this section. In all cases, control is returned to the calling program.

The RELES subroutine releases space held by a PLAN dynamic file to the pool of available disk space. RELES performs the opposite function of the FIND subroutine.

```
    CALL RELES (ID,0,NSQZ,NDR)
```

The CALL RELES parameters ID and NDR have the same meaning as defined for the CALL FIND. Use of this call prevents the unnecessary accumulation of temporary data. Obsolete files that are not released may degrade performance by forcing long seeks.

If NSQZ is zero, the file control block is closed and ID(2) is not altered. NSQZ provides for partially releasing space allocated to a DYNAMIC file, that is, the first NSQZ words of the file are retained. In actuality, if NSQZ is other than 0, the file control block is not closed. The current file length indicators are updated when necessary and disk space is released to the available pool whenever complete allocation units are found to be free. The true file size is set to the value of NSQZ if it is greater than NSQZ. On drives other than 0, priority 1 files are released only by action (CALL RELES) of the programmer (logic module). All files on DYNAMIC drive 0 are released when a level 1 command is processed. Therefore, permanent DYNAMIC files may not reside on DYNAMIC drive 0.

On DYNAMIC drive 0, all DYNAMIC files (including priority 1 files) are released automatically when a level 1 PLAN statement is processed. Logical files of priorities 2, 3, and 4 on other DYNAMIC drives are automatically released when a higher-level phrase is processed, but priority 1 files must be released by CALL RELES.

Automatic release of files with a priority less than 1 is accomplished whenever a command with a higher level (lower-numbered) than the file priority is processed. Thus, a level 1 command results in release of all files with a priority of 2, 3, or 4; level 2 commands result in release of files with a priority of 3 and 4; level 3 commands result in release of files with a priority of 4. Open file control blocks are not closed. A further attempt to process a released file results in a phrase abort and PLAN error recovery is initiated. The automatic release function is also invoked if the DYNAMIC drive is filled and a request for space for a higher-priority file is generated from the WRITE subroutine.

The CALL RELES function should be the last file function executed for any file whose control block is not in COMMON and whose data has no future use.

Transfer of data from memory to the DYNAMIC file is accomplished with a CALL WRITE. Transfer from the DYNAMIC file is accomplished with CALL READ.

```
    CALL READ (ID,KDIS,KOUNT,ARRAY)
    CALL WRITE(ID,KDIS,KOUNT,ARRAY)
```

ID is the first word of the file control block as defined for the CALL FIND. A coded pointer to the DYNAMIC file is set in ID(1) by the FIND routine. The user must not alter this word. For the READ/WRITE to be successfully executed the file control block must show a properly opened file (ID(1)>256).

KDIS is the number of 32-bit words beyond the beginning of the file at/to which transfer is to take place. The KDIS value for the first word in the file is zero. Therefore, the value for KDIS is always (N-1), where N represents the number of the first word to transfer.

KOUNT is the number of 32-bit words to be transferred with this READ/WRITE operation. On a CALL WRITE, ID(2) will be set to KDIS + KOUNT if KDIS + KOUNT is greater than the current ID(2). This action will also cause the true file size (as maintained in the file on disk) to be updated if this WRITE is causing an expansion to the true file size. This update may be eliminated by the user if he writes a word to the end of the desired file at the beginning of the file write sequence.

ARRAY is the location in the user's program or in the communication array into which the data is to be read or from which the data will be written. Data transfer starts at ARRAY(1) and continues through ARRAY (KOUNT). Transfer continues from KDIS through KDIS + KOUNT - 1 on disk.

The FIND and WRITE subroutines maintain an updated count of the length of the file in PLAN words. An end-of-file (EOF) will be detected on CALL READ if KDIS + KOUNT is greater than ID(2) (second word of the file control block) or greater than the true file size. The condition is indicated by setting ID(1) to the file number (ID(1) <256). The following FORTRAN statement provides a test for the condition:

```
    IF (ID(1)-256) 1,1,2
```

Statement 1 is executed if the file is closed as defined above. Statement 2 is executed if a successful READ operation occurs.

The preclosing of files (EOF) can also occur on CALL WRITE. The WRITE routine causes physical storage to be incrementally assigned to the DYNAMIC file as needed during execution. If storage cannot be allocated to provide the required file space, even by releasing a lower-priority file, an end-of-file (EOF) is detected on WRITE.

Any EOF condition detected is indicated by setting ID(1) to its original value, in effect, closing the file. If an attempt is made to read or write a closed file, the READ/WRITE routine will generate an EOF diagnostic, terminate processing of the current statement, initiate level error recovery, and load PSCAN to process the next PLAN statement. The test listed above can be used to prevent an unplanned termination.

If a CALL FIND is subsequently executed on a file control block that has been closed because of EOF detection, the file will be reopened and the current available length of the file will be placed in ID(2). If the EOF condition resulted from a CALL WRITE, the subsequent CALL FIND results in a file length indication equal to that which existed before the CALL WRITE that precipitated the EOF condition.

```
TEST DYNAMIC FILE SPACE AVAILABILITY
-----------------------------------------

CALL PFSPC(0,NPRI,NALLO,NDR)

  0     Indicates a reserved parameter
  NPRI  Priority at which space
        is desired
  NALLO Location for available space
        indicator
  NDR   DYNAMIC drive on which space is
        desired
-----------------------------------------
C     SET PRIORITY
      NPRI=4
C     FIND SPACE AT THIS PRIORITY
    1 CALL PFSPC (0,NPRI,KT,2)
C     IS 2950 WORD AVAILABLE
      IF (KT-2950) 5,5,15
C     SET TO HIGHER PRIORITY
    5 NPRI=NPRI-1
C     ARE ALL PRIORITIES CHECKED
      IF (NPRI) 10,10,1
C     EXIT AND TERMINATE COMMAND
   10 CALL LEX (1,0)
C     OPEN 2950 WORD FILE AT LOW PRIORITY
   15 ID(1)=2
      CALL FIND (ID,NPRI,2950,2)
        •
        •
        •
```

A test may be made at any time to determine the space available for a DYNAMIC file at any given priority. If the space available is greater than the maximum size of a PLAN file, the result is set to the maximum sizeof a DYNAMIC file. The test is accomplished with a call to the PFSPC subroutine.

```
    CALL PFSPC(0,NPRI,NALLO,NDR)
```

NPRI defines the priority of the file on DYNAMIC drive NDR for which the available space is desired. If NPRI is 4, NALLO is set equal to the total number of 32-bit words (up to a maximum file size) currently unassigned to any PLAN file. If NPRI is 1, 2, or 3, NALLO is set equal to the unassigned file space plus any space currently assigned to any lower-priority files. If NPRI is 0, the level of the current command will be assumed, and processing continues as outlined above. A zero will be placed in NALLO if any error is detected by PFSPC.

PLAN procedures for DYNAMIC file errors are invoked on the basis of the DYNAMIC FILE ERROR INDICATOR (see "PLAN JOB" 4.5.4).

The following exceptional conditions can be detected by the DYNAMIC file subroutines.

0. An ID argument specifying an unopened file control block on CALL READ or CALL WRITE.
1. KDIS + KOUNT greater than ID(2) or true file size on CALL READ.
2. An invalid NDR or ID argument on CALL FIND or RELES.
3. An invalid ID argument on CALL READ or WRITE.
4. Invalid KDIS or KOUNT argument on CALL READ or WRITE.
5. DYNAMIC drive not available.
6. Insufficient space for allocation on CALL FIND or WRITE.
7. Reserved.
8. (1130 only) Pack ID not equal on validity check.
9. (1130 only) PFIND not in PLAN library.

If the DFI indicator in the PLAN JOB command is used, conditions 1-9 in the above list result in closing the file control block and a negative number is stored in ID(2). The negative number is an integer (the absolute value of which is shown in the above list) that, when added to 120, corresponds to a diagnostic number that is produced as a result of the error when in the immediate mode.

Condition 0 when encountered always results in a PLAN phrase abort following generation of PLAN diagnostic DFJ120.

The example shown below illustrates logic of a check that should be programmed following each CALL WRITE if optional mode of operation is selected in which the DYNAMIC FILE ERROR INDICATOR is on.

```
      EQUIVALENCE(ID2,ID(2))
C     WRITE A RECORD
      CALL WRITE (ID,KDIS,KOUNT,ARRAY)
C     DID WRITE CLOSE THE FILE
      IF (ID-256) 3,3,30
C     SELECT ERROR TYPE - IS EOF ON
    3 IF (ID2) 5,35,35
    5 KD2=-ID2
      GO TO (10,15,20,25,30),KD2
C     LOGICAL END OF FILE
   10 CONTINUE
        •
        •
   15 INVALID FIND/RELES PROCESSING
        •
        •
        •
C     INVALID READ/WRITE ERROR PROCESSING
   20 CONTINUE
        •
        •
        •
C     NEGATIVE KDIS/KOUNT ERROR PROCESSING
   25 CONTINUE
        •
        •
        •
C     (1130 ONLY) DYNAMIC DRIVE AVAILABLE
   30 CONTINUE
        •
        •
        •
C     EOF PROCESSING
   35 CONTINUE
```

The above files are referred to as PLAN DYNAMIC files. The following points summarize the characteristics of these files:

1. A DYNAMIC file is established only when program logic determines the file is required.
2. A DYNAMIC file is assigned physical disk space in modular blocks only when the space is required.
3. Every position (32-bit word) within the DYNAMIC file is discretely addressable. This allows disk to be treated as an extension of memory.

5.11.3 PERMANENT FILE SUPPORT

```
PERMANENT FILE ROUTINES

CALL GDATA (ID,NAME,LR,NDR)
CALL RDATA (ID,KDIS,KOUNT,KORE)
CALL WDATA (ID,KDIS,KOUNT,KORE)

  ID     File control block
  NAME   File name (EBCDIC)
  LR     Physical record length in bytes
  NDR    PERMANENT drive number
  KDIS   Record displacement in file
  KOUNT  Record length
  KORE   User array

      DIMENSION ID(2),A(2),W(100)
      COMMON L(625),LS(15)...
      EQUIVALENCE(ID(2),ID2)
      DATA A/'NAME','F'/
C     FILE NUMBER 1
      ID(1)=1
C     OPEN FILE
      CALL GDATA(ID,A,I,0)
C     ZERO ARRAY
      DO 5 I=1,100
    5 W(I)=0.
C     READ 1ST WORD
      CALL RDATA (ID,0,1,W)
C     IS WORD ZERO
      IF(W(1))8,25,8
C     WRITE FILE WORDS TO ZERO
    8 KT=100
    9 W(1)=0.
      DO 20 I=1,ID2,100
C     IS PARTIAL WRITE RQD
      IF (ID(2)-I+1-KT) 10,15,15
   10 KT=ID(2) -I+1
   15 CALL WDATA(ID,I-1,KT,W)
   20 CONTINUE
   25 CONTINUE
        •
        •
        •
```

A second type of direct access file support is provided by PLAN subroutines. This class of support is designed for files established outside of PLAN. The file is fixed-size and is permanent in terms of the ability to establish or release the file as a result of program logic within a PLAN module. Characteristics of these files and the procedures for establishing them are closely dependent upon the monitor or operating system in control. More specific restrictions or capabilities are outlined in the appendices of this manual.

The subroutines to provide this support are listed below. The subroutine arguments are the same as defined above for the PLAN DYNAMIC files. The calling sequence is:

```
CALL GDATA (ID,NAME,LR,NDR)
```

CALL GDATA performs functions for the PERMANENT file that CALL FIND does for the DYNAMIC file, except that no disk space is allocated.

The parameters of the calls listed above have the following meaning:

ID is the first word of the two-word file control block. Each file to be referenced must have its own file control block. If the file control block is in COMMON (the communication array), one CALL GDATA can satisfy a series of programs and result in a saving of disk access time. If the file control block is not in COMMON, each program must issue its own CALL GDATA for the file. The value stored in ID(1) by the calling program must be an integer from 1-255. This is the PERMANENT file number. After the CALL GDATA has been executed, ID(1) contains a coded pointer to the data set, and ID(2) contains the file length in PLAN words.

NAME, is an eight-character (64-bit) file name left-justified and padded with blanks. On the 1130, only the first five characters are significant. On DOS, only the first seven characters are significant. NAME is unused under OS/360 PLAN but must be provided by JCL (see <u>OS Problem Language Analyzer (PLAN) Operations Manual</u> H20-0596). If the length of the file name is less than eight characters the remaining characters must be padded with blanks. Additional information on procedures for this action is contained in the appropriate appendix under "PERMANENT File Support".

An automatic equivalence between NAME and the file number in ID(1) is implied, and no further equivalencing is required, except as defined in Appendices B and C. The file number must be specified in ID(1) even though the file is identified by name. Example:

```
DIMENSION NAME(2), ID(2)
DATA NAME/'DATA','F'/
ID(1)=25
CALL GDATA (ID,NAME,LR,0)
```

The above defines PERMANENT file 25, which is named DATAF on PERMANENT drive 0 if

LR contains the physical length in bytes as a fixed-point integer after CALL GDATA has been executed. On 1130 PLAN this value is a constant 640 (320 1130 words).

The file size determined when GDATA is called represents the total file size; not just the portion that has been written. Therefore, the user should implement a convention (for example, storing the true file size in the first word of the file) for maintaining the true file size if it is necessary at any time to determine how much data has been written within the extent limits of the file.

NDR contains the number of the PERMANENT drive on which the file exists, and may range from 0-7.

```
    CALL RDATA (ID,KDIS,KOUNT,ARRAY)
    CALL WDATA (ID,KDIS,KOUNT,ARRAY)
```

KDIS is the number of 32-bit words beyond the beginning of the file at/to which transfer is to take place. The KDIS value for the first word in the file is zero. Therefore, the value for KDIS is always (N-1), where N represents the sequence within the file of the first word to be transferred.

KOUNT is the number of 32-bit words to be transferred with this RDATA/WDATA operation. On a CALL WRITE ID(2) will be set equal to KDIS + KOUNT if KDIS + KOUNT is greater than the current ID(2). Any attempt to issue a RDATA/WDATA outside the true file size will result in the file control block being marked as closed.

ARRAY is the user's data array to/from which data is to be transferred from/to the file.

PLAN procedures for PERMANENT file errors are invoked on the basis of the PERMANENT FILE ERROR INDICATOR (see "PLAN JOB" 4.5.4).

The following exceptional conditions can be detected by the PERMANENT file support subroutines.

1. An invalid NDR or ID argument on CALL GDATA.

2. An invalid file ID argument on CALL RDATA or WDATA.

3. Negative KDIS or KOUNT arguments on CALL RDATA or WDATA.

4. An ID argument specifying an unopened file control block on CALL RDATA or WDATA.

5. KDIS + KOUNT greater than ID(2) of the file control block or the actual file size.

In the normal mode of operation, cases 1 through 4 are considered phrase abort conditions and cause a diagnostic message to be issued and PLAN level recovery procedures invoked.

Case 5 is considered a normal program event (EOF), and the only action ever taken is to mark the file control block as closed. ID(1) is reset to the file number to mark the file as closed.

If the PERMANENT file error indicator in Switch Word 13 is on, cases 1 through 3 cause a value of -1 to -3, respectively, to be placed in ID(2) of the file control block, and ID(1) is reset to the file number (the file control block is marked as closed).

Case 4 always causes a phrase abort condition.

5.11.4 ONE-WORD INTEGER SUPPORT

16-BIT FILE SUPPORT

```
CALL PFND1(ID,NPRI,NALLO,NDR)
CALL PRED1(ID,KDIS,KOUNT,ARRAY)
CALL PWRT1(ID,KDIS,KOUNT,ARRAY)
CALL PREL1(ID,0,NSQZ,NDR)
CALL GDAT1(ID,NAME,LR,NDR)
CALL RDAT1(ID,KDIS,KOUNT,ARRAY)
CALL WDAT1(ID,KDIS,KOUNT,ARRAY)
```

ID	Two-word file control block that must be located on an even boundary
NPRI	File priority
NALLO	Initial allocation requirement
NDR	Logical drive number
KDIS	Displacement within the file
KOUNT	Number of words to transfer
ARRAY	User's data array
NSQZ	Number of words to not release
NAME	File name
LR	Record length in bytes

Samples of use of these commands can be drawn from the blocks "DYNAMIC File Support" and "PERMANENT File Support" if the following name substitutions are made:

FIND	to	PFND1
READ	to	PRED1
WRITE	to	PWRT1
RELES	to	PREL1
GDATA	to	GDAT1
RDATA	to	RDAT1
WDATA	to	WDAT1

The one-word (16-bit) integer option of FORTRAN and other nonstandard storage options are supported by PLAN direct access file routines. Subroutines PRED1, RDAT1, PWRT1, WDAT1, PFND1, PREL1, and GDAT1 function with one-word integer data in a manner identical to READ, RDATA, WRITE, WDATA, FIND, RELES, and GDATA with ASA standard (32-bit) integer. The values of KDIS and KOUNT are given as 16-bit word counts. The file length count (ID(2)) is maintained as a 16-bit word count. Conversions for odd-word counts and odd boundaries in one-word integer arrays are automatic. Note that the maximum actual file size that may be processed with one-word integer support is one-half the number of machine words that are attainable with the 32-bit support.

The file control block is organized in the same manner as defined for the "DYNAMIC File Support" and "PERMANENT File Support" and must be on an even word boundary. The equivalences between standard ASA and one-word integer support are shown in the following diagram:

STANDARD ASA

ID(1)	ID(2)

ONE-WORD SUPPORT ON 1130

ID(2)	ID(1)	ID(4)	ID(3)

ONE-WORD SUPPORT ON System/360

ID(1)	ID(2)	ID(3)	ID(4)

5.11.5 UTILITY SUBROUTINES

```
 LOGICAL TEST FACILITY
--------------------------------------
 IF (NDEF(ARG)) 1,2,3
 CALL TRUE(ARG)
 CALL FALSE(ARG)

   ARG   User word
   1     FALSE exit
   2     TRUE exit
   3     REAL exit
--------------------------------------
       DIMENSION A(3)
C      SET A(3) REAL
       A(3) = 0
C      SET A(2) TRUE
       CALL TRUE (A(2))
C      SET A(1) FALSE
       CALL FALSE (A)
C      TEST, FALSE, TRUE, REAL
       DO 6 I=1,3
       IF (NDEF(A(I))) 1,2,3
     1 J=1
       GO TO 4
     2 J=2
       GO TO 4
     3 J=3
     4 IF(I-J) 5,6,5
C      TEST ERROR
     5 PAUSE 5
     6 CONTINUE
         •
         •
```

NDEF(ARG) is an arithmetic function that allows testing of any location for the PLAN logical functions TRUE, FALSE, or REAL. Example:

    IF(NDEF(ARG)) 1,2,3

1. Statement 1 will be executed next if ARG is FALSE (7FFFFFFF Hexadecimal).

2. Statement 2 will be executed next if ARG is TRUE (80000000 Hexadecimal).

3. Statement 3 will be executed next if ARG is anything else.

The value of ARG is unchanged by execution of the NDEF function.

CALL TRUE(ARG) is a subroutine that sets ARG to the value of logical TRUE.

CALL FALSE(ARG) is a subroutine that sets ARG to the value of logical FALSE.

```
CURRENT COMMAND RETRIEVAL

CALL INPUT(N,ARRAY)
N       Size of user array in 32-bit words
ARRAY   Array in which to store command
        image

      DIMENSION A(114),IA(1)
      COMMON L(625),LS(15)
      EQUIVALENCE (A(1),IA(1),IA1)
      CALL PSBFA(100)
        •
        •
        •
C     READ PHRASE
      CALL INPUT (114,A)
C     PRINT PHRASE IMAGE
      NWDS = (IA1+7) /4
      DO 10 I=2,NWDS,30
      CALL PAOUT(100,1,120,A(I))
   10 CALL PLOUT(100)
```

CALL INPUT(N,ARRAY) is a subroutine that writes into memory the EBCDIC representation of the last PLAN command processed and the length of the command in characters to allow interrogation or printing of the command.

ARRAY is the name of the array that will contain the image in A4 format at the end of execution of the input subroutine. N is the length of ARRAY in 32-bit words. ARRAY (1) is set to the total number of characters (fixed-point) in the statement. The number of characters placed in memory equals the length of the statement unless the statement is greater than 4*(N-1) characters. The maximum number of characters to be read is 4*(N-1). Any unused positions in the array are set to blank but are not counted.

```
COMMAND EXECUTION FROM MEMORY

CALL PUSH(LITL)
  LITL PLAN literal representing the
       command to execute

ADD PHRASE: ...(27)'DUMP '...
    •
    •
    •
CALL PUSH(CA(27))
```

CALL PUSH(LITL) allows a user to call for execution of a command from within a user-written program module. LITL is a pointer to the character count of a PLAN literal. The literal is a PLAN statement minus the semicolon with at least one blank following to provide space for insertion of the semicolon. The PUSH subroutine replaces the blank with a semicolon and links to PSCAN to execute the phrase. The module issuing the CALL PUSH is not reentered. The pop-up list is not cleared.

```
LITERAL FILE MAINTENANCE

CALL PHIN(ID,I,A)
CALL PHOUT(ID,I,A)

  ID     GDATA open file control block
  I      Literal identification number
  A      Array containing PLAN literal

ADD PHRASE: LITERAL, (5)'TEST PHIN AND
PHOUT';
LITERAL;
        •
        •
        •

      DIMENSION A(20),FNAME(2),ID(2)
      COMMON L(625),LS(15),CA(510)
      DATA FNAME/'FNAM','E'/
        •
        •
        •
      ID(1) = 5
      CALL GDATA(NFCB, FNAME, KT, 2)
C     WRITE LITERAL TO FILE
      CALL PHOUT (ID,12,CA(5))
C     RETRIEVE LITERAL
      CALL PHIN (ID,12,A)
C     FIND WORDS IN LITERAL
      KT = (CA(5)+3)/4
C     COMPARE IN-OUT LITERAL
      DO 5 I=1,KT
      IF (CA(I+5) -A(I+1)) 4,5,4
    4 PAUSE
    5 CONTINUE
        •
        •
        •
```

CALL PHIN(ID,I,A) is a subroutine that retrieves a PLAN literal or table from a PLAN PERMANENT file and places it in memory, starting at location A. Literal number I in the file defined by the open GDATA file control block ID is a PLAN literal whose length in characters (bytes) is placed in A(1). The text of the literal is placed in A(2) to A(2+(N-1)/4), where n is the length placed in A(1). Literals or tables retrieved by PHIN must have been written using PHOUT or the SET LITERAL command.

Following execution of the CALL PHIN, A(1) will contain an integer value as defined in the table shown below:

INTEGER IN A(1)	MEANING
+N	The number of characters in the literal.
0	The file control block ID was found not to be open.
-1	The file is not a valid literal file.
-2	The requested literal number is greater than any literal contained in the file.
-3	The requested literal number is not contained within the literal file.

CALL PHOUT(ID,I,A) is a subroutine that adds PLAN literals or tables to a PLAN PERMANENT file. The literal number is I and the character count (bytes) followed by the literal (table) is located in memory at location A. The file to which the literal is to be added is defined by the open GDATA file control block located at ID. The SET LITERAL command invokes execution of the PDIAG module, which in turn calls PHOUT.

```
MODIFY STANDARD PLAN DEVICES

CALL IOCS (INPUT,LIST)

 INPUT  Device number for standard PLAN
        input device
 LIST   Device number for standard PLAN
        output device

C     MODIFY INPUT DEVICE
      CALL IOCS(3,0)
C     MODIFY OUTPUT DEVICE
      CALL IOCS (0,102)
C     MODIFY INPUT AND OUTPUT DEVICE
      CALL IOCS (2,101)
```

CALL IOCS (INPUT,LIST) is a subroutine that allows a logic module to alter the PLAN statement input device and diagnostic list device. All parameters are in the fixed-point mode. A value of zero for either parameter does not alter the existing value. Note valid device parameters for each PLAN system are listed in the appropriate appendix. A value of 0 for NOD for the sequential I/O routine specifies the PLAN input device, and 100 specifies the current PLAN output device.

5.11.6 ERROR INTERFACE SUBROUTINES

```
GENERATE DIAGNOSTIC MESSAGE

  CALL ERROR(N1,N2,LIT)
  CALL ERREX(N1,N2,LIT)
  CALL ERRET(N1,N2,LIT)
  CALL ERRAT(N1,N2,LIT)

   N1 Error number
   N2 Error code
   LIT PLAN literal containing
      diagnostic text

ADD PHRASE:  TEST, (25)'TEST DIAGNOSTIC';
TEST;
        •
        •
        •
      COMMON L(625),LS(15),M(100)
C     TERMINATE PHRASE AND MODULE
      CALL ERROR (123,N,M(25))
C     TERMINATE MODULE
      CALL ERREX (M,27,0)
C     GENERATE DIAGNOSTIC AND RETURN
      CALL ERRET (1,2,M(25))
C     TERMINATE PHRASE BUT NOT MODULE
      CALL ERRAT (105,0,0)
```

CALL ERROR/ERREX/ERRET/ERRAT(N1,N2,LIT) are subroutines that access the system error processor to produce error diagnostics. In each call, N1 and N2 are user-selected identification numbers that may be used for diagnostic messages. LIT is the word containing the character count of a PLAN literal that is to make up the diagnostic message. The literal text in PLAN literal format is found in LIT(2) to LIT(n), where n equals the character count minus one divided by four plus two.

The ERROR subroutine aborts the current PLAN statement (initiates PLAN level error recovery);

ERREX returns to the PLAN loader to execute the next program in the pop-up list;

ERRET returns to the next statement in the calling program;

ERRAT returns to the next statement in the calling program, processes the remaining programs in the pop-up list, and effects PLAN level error recovery the next time that the pop-up list is found empty. Note that a phrase abort means PLAN level error recovery procedures are initiated. These calls, when necessary, save the calling module, load the error processing module, and restore the calling module if saved. The calling module, therefore, does not need the output device routine and so is more economical of memory utilization than inline error coding.

On the 1130, if COMMON in the calling module is larger than the maximum protected by PLAN, the programmer must use a PLAN file to save and restore the additional COMMON if error processing is in the immediate mode, if a user error module is required to process the diagnostics, or if the diagnostics will cause an overflow of the PLAN error stack. (Note that overflow of the PLAN error stack is unpredictable to the programmer.)

Line 1 of the diagnostic locates the error and identifies it with the programmer's catalog codes N1 and N2. Underlined portions of the message are variable. The literal text of the last command processed precedes all diagnostics resulting from the phrase if the long-form diagnostic is selected (see "Switch Words", 4.3.21).

The format and meaning of the diagnostic produced by the routines defined above are as follows:

```
DFJ000 001-100 TEXT
       101-199 TEXT
       201-299 TEXT
       301-399 TEXT
       401-450 TEXT

CCCnnn *A* mmmmm SEQ=yyy ID=ccccc
PG=xxxxxxxx DIAGNOSTIC
```

The underlined segments of the diagnostic message are variable.

TEXT this field of up to five lines contains the current PLAN statement. It is printed only if the long-form diagnostic is printed. Character positions are printed to the left of the text.

CCC is component code indicating whether the error was generated by PLAN (CCC=DFJ) or by the user (CCC=***).

nnn is the three-position error number provided as N1 in the call to the PLAN error subroutines. For PLAN errors (DFJ), nnn is in the range of 1-99 for PHRAS errors; 101-199 for execution errors; 201-299 for PSCAN errors; 701-799 for 1130 PLAN errors; 801-899 for DOS PLAN errors; 901-999 for OS PLAN ERRORS.

A is an action code. E indicates an exit from PLAN; R indicates a PLAN level error recovery; C indicates continued processing; O indicates a pause for operator intervention.

mmmmm is the error code provided as N2 in the call to the PLAN error subroutines.

SEQ=yyy is the three-position sequence of the PLAN statement currently being processed relative to the first statement processed following the last level 0 command.

ID=ccccc provides the five-character contents from the identification field of the last PLAN input record processed before the diagnostic routine call.

PG=xxxxxxxx is the name of the module that issued the call to the PLAN error subroutines.

DIAGNOSTIC is the text of the literal indicated by LIT in the diagnostic routine call. If LIT is zero (literal character count) in the calling sequence, no literal message will be written, and the field is filled with asterisks. The literal message is limited to 76 characters.

LIST ERROR FILE
CALL ERLST

CALL ERLST is a subroutine that may be called to force a dump of the error message queue file (DYNAMIC file 255, drive 0). Processing of the current statement is terminated and control is passed to PSCAN to process the next input record. This subroutine supports the technique of post-listing diagnostics. The PLAN error message queue file is automatically dumped when a level 0 or level 1 phrase is encountered or when a /* input statement is processed.

```
|WRITE DIAGNOSTIC TO PLAN ERROR FILE          |
|---------------------------------------------|
|CALL EWRIT(NCTL,ARRAY)                       |
|                                             |
| NCTL   Associated carriage control          |
|        function                             |
| ARRAY  User array containing the            |
|        diagnostic                           |
|---------------------------------------------|
|ADD PHRASE:TEST,(25)'SAMPLE DIAGNOSTIC';     |
|TEST;                                        |
|        •                                    |
|        •                                    |
|        •                                    |
|        COMMON L(625),LS(15),CA(510)         |
|        CALL EWRIT (1,CA(25))                |
```

CALL EWRIT(NCTL,ARRAY) is a subroutine that allows a user to enter diagnostics into the

PLAN error message queue file (file 255, drive 0) in a format that may be processed by ERLST. Any diagnostic written to the file will be automatically purged as defined under CALL ERLST. NCTL in the calling sequence defines the carriage control functions to be associated with this diagnostic and has the same meaning as defined for CALL PCCTL. Array is a pointer to the first word (character count) of the PLAN literal that contains the diagnostic text. The maximum number of characters in the literal is 120.

5.11.7 SORT/MERGE

```
SORTING AND MERGING
-----------------------------------------
CALL PSORT(ID)
CALL PMERG(ID,JD,KD)

  ID File control block of SORT file or
     MERGE output file
  JD file control block of first
     merge file
  KD file control block of second
     MERGE file
-----------------------------------------

  COMMON L(625),LS(15),CA(510),KA(1)
  DIMENSION JD(2),KD(2),ID(2)
  EQUIVALENCE (KA(1),(A(1))
C FIND ERASABLE COMMON
  NEC = LS(8)
  ID(1) = 1
  JD(1) = 27
  KD(1) = 34
C SET UP SORT CONTROL FIELDS
C RECORD SIZE
  KA(NEC) = 25
C SORT/FIELDS/RECORD
  KA(NEC+1) = 1
C FIRST SORT KEY
  KA(NEC+2) = 3
  KA(NEC+3) = 12
  KA(NEC+4) = 1
  CALL FIND (ID,0,0,1)
  CALL FIND (KD,0,0,0)
  CALL FIND (JD,0,0,1)
  CALL PSORT (ID)
  CALL PSORT (KD)
  CALL PMERG (JD,ID,KD)
```

CALL PSORT(ID) is a subroutine that initializes and calls in the PLAN DYNAMIC FILE SORT facility. ID is a pointer to the first word of the open file control block of the file to be sorted. The file to be sorted is replaced by the sorted file. ID(2) must reflect the file size to be sorted. If the entire file is to be sorted, the CALL FIND automatically satisfies this requirement. However, if only a portion of the file is to be sorted or merged, ID(2) must be modified to reflect the intent. COMMON outside of that defined by Switch Word 9 may be destroyed.

CALL PMERG(ID,JD,KD) is a subroutine that invokes the DYNAMIC file MERGE facility. ID is a pointer to the first word of the open file control block for the file that is to receive the merged file. JD and KD are pointers to the file control blocks of the DYNAMIC files to be merged. JD(2) and KD(2) must reflect the file sizes to be merged.

PSORT/PMERG results in the overlay by the PLAN loader checkpoint facility (CALL LCHEX) of the module issuing the call with PSRTA/PMRGA. Use of these functions requires uniform-length logical records written by the DYNAMIC file routine WRITE. Sort/merge keys may be located at random within the record. They may be binary, alphameric, or numeric (integer or floating-point). The sorted file replaces the original file on disk. The merge phase creates a file from any two existing files. Parameters that control sorting and merging must be stored by the calling program in the first positions of ERASABLE COMMON as defined by Switch Word 8.

TYPE OF SORT	ASCENDING/ DESCENDING & MODE	DISPLACEMENT =RELATIVE BYTE	LENGTH
ALPHAMERIC	± 1	0,1,2, ...	# OF CONTIGUOUS ELEMENTS
ASA STANDARD INTEGER	± 2	0,4,8, ...	# OF CONTIGUOUS ELEMENTS
32-BIT FLOATING-POINT	± 3	0,4,8, ...	# OF CONTIGUOUS ELEMENTS
32-BIT LOGICAL BINARY	± 4	0,4,8, ...	32-BIT EXTRACTION MASK
16-BIT INTEGER	± 5	0,2,4, ...	# OF CONTIGUOUS ELEMENTS
LONG-PRECISION (SYSTEM/360 ONLY)	± 6	0,8,16, ...	# OF CONTIGUOUS ELEMENTS

Figure 11. Sort control fields

Figure 11 illustrates the sort control fields and their meanings. The types of sorts and merges that may be invoked are:

Alphameric. Any sort field definition may define a sort key of from 1 to 256 consecutive EBCDIC characters to be sorted. The field must not extend beyond the end of the logical record.

ASA Standard Integer. Each sort field definition may define a sort key of from 1 to 512 consecutive ASA fixed-point integer numbers to be sorted. The field must not extend beyond the end of the logical record.

Floating-Point. Specifications for the floating-point sort are identical to those defined above for ASA standard integer.

Logical Binary. Each sort field definition defines a 32-bit word that is to be matched against the extraction mask (AND) and then sorted in logical sequence.

Half-Word or One-Word Integer. Each sort field definition may define from 1 to 1024 16-bit consecutive binary integers to be sorted. The field must not extend beyond the end of the logical record. Since the record length is a definition of 32-bit words, it is not possible to sort the file as a contiguous series of 16-bit integers.

Long-Precision (System/360 only). Each sort field definition may define from 1 to 256 64-bit consecutive long-precision, floating-point binary numbers. The field must not extend beyond the end of the logical record.

The ascending/descending mode indicator is a binary integer that is positive if the sort field is to be sorted into ascending sequence and is negative if a descending sort is indicated. The absolute value of the integer indicates the type of sort as shown.

The displacement is an integer value that is a relative byte displacement from the beginning of the record to the leftmost byte of the sort field.

The length field defines the number of consecutive elements to be sorted except for the logical sort. For logical sorts the field is a mask that is combined by a logical AND with the word to be sorted.

The first word of erasable COMMON must contain the logical record length in 32-bit words. It must be a positive integer not greater than 512.

The second word contains the number of three-word sort control groups (M) and must be a positive integer in the range 99>M<(L/3-2), where L is the length of ERASABLE COMMON.

Words 3,6,9... contain the mode indicators as shown in Figure 11, column 2.

Words 4,7,10 contain the displacement of the first element in the sort field.

Words 5, 8, 11... contain the count of consecutive elements or the extraction mask as defined in the length column in Figure 11.

The following example shows the status of erasable COMMON required to effect an ascending sort on a file with a record length of 20 words. The sort is floating-point on word 5 within word 12 (major field).

ERASABLE COMMON	CONTENT	
1	20	Record length
2	2	Number of sort keys per record
3	3	First three-word group
4	44	(major field key
5	1	indicators)
6	3	Second three-word group
7	16	(minor field key
8	1	indicators)

Exception conditions are handled as follows:

1. When the CALL PSORT/PMERG is executed, and an indicated file is found to be closed, PLAN level error recovery (phrase abort) is initiated.

2. A merge file found to be unsorted results in a phrase abort.

3. If the length of the file divided by the specified record length is not an integral value, the short record is undisturbed.

4. If an error is found in the sort/merge definition in erasable COMMON, a phrase abort is initiated.

5. On PMERG insufficient space to complete the merge will result in a phrase abort condition.

5.11.8 SORT/MERGE KEY DEFINITION

The diagrams below indicate how disk data set displacement varies for different sort/merge keys rather than the logic of core-array storage. Array position corresponds to in-core subscript of record when read from disk.

Displacement Keys for ASA Integer Data and ASA Floating-point Data

ARRAY POSITION	I(1)		I(2)		I(3)	
KEY	0		4		8	

Displacement Keys for 16-Bit Integer Data

ARRAY POS.	I(1)	I(2)	I(3)	I(4)	I(5)	I(6)
KEY	0	2	4	6	8	10

Displacement Keys for EBCDIC (Literal) Information

ARRAY POS.	A	L	P	H	A	N	U	M	E	R	I	C
KEY	0	1	2	3	4	5	6	7	8	9	10	11

Displacement Keys for Binary SORT

ARRAY POSITION	32-BIT	FORTRAN	WORDS
KEY	0	4	8

5.11.9 PLAN SEQUENTIAL I/O ROUTINES

The following subroutines provide overlapped, buffered I/O capability from a module via subroutine calls. Total compatibility between all systems supported by PLAN is provided. Reread capability is also provided within the framework of FORTRAN, as well as variable input/output formatting capability.

This set of routines is broken into the following general classifications:

1. Buffer Assignment (1130 only)
2. Input/Output
3. General Control
4. Format Control

```
BUFFER ASSIGNMENT

  CALL PSBFA(NOD)
  CALL PSBFB(NOD)
  CALL PSBFC(NOD)
  CALL PSBFD(NOD)
  CALL PSBFE(NOD)
  CALL PDBFA(NOD)
  CALL PDBFB(NOD)
  CALL PDBFC(NOD)
  CALL PDBFD(NOD)
  CALL PDBFE(NOD)

    NOD   PLAN device code definition
```

The buffer assignment routines provide single or double buffer assignment for each device used within a module. Each device to be used within the module must be specified as an argument of a buffer assignment routine. Two different devices should not be associated with a particular buffer set, that is, each subroutine should be called only once within any module. The buffer assignment routines are not required (they will execute as a no-op) in System/360 OS and DOS PLAN.

CALL PSBFA(NOD), CALL PSBFB(NOD), CALL PSBFC(NOD), CALL PSBFD(NOD), and PSBFE(NOD) are five calls that allow five single-buffer sets to be assigned to input/output devices (NOD). NOD defines the device to be associated with a particular buffer set. The allowable values are the same as those defined for CALL IOCS.

CALL PDBFA(NOD), CALL PDBFB(NOD), CALL PDBFC(NOD), CALL PDBFD(NOD), and CALL PDBFE(NOD) are five calls that allow five double-buffer sets to be assigned to input/output devices (NOD). Use of double-buffer sets allows <u>automatic</u> overlapped input/output operations. NOD defines the device to be associated with a particular buffer set. The allowable values are the same as those defined for INPUT and LIST under CALL IOCS.

```
READ/WRITE SEQUENTIAL FILES
-----------------------------------------
CALL PLINP(NOD)
CALL PLOUT(NOD)

  NOD   PLAN device assignment
-----------------------------------------

C     READ AND LIST A FILE
      CALL PSBFA(100)
      CALL PDBFA(0)
    5 CALL PLINP(0)
      CALL PLOUT(100)
C     TEST FOR END OF FILE
        •
        •
        •
C     IF NOT END-OF-FILE
      GO TO 5
```

The input/output routines provide for transfer of information from/to device NOD to/from a buffer as defined in the call to the buffer assignment routine. A pointer is set to the buffer to allow communication with the format control routines. When double buffering is specified, automatic overlapped transfer is initiated where possible. These routines wait until previously initiated I/O on the current buffer is complete.

CALL PLINP(NOD) is a subroutine call to transfer a record from the device NOD to the buffer set associated with NOD. Overlapped, look-ahead reading is initiated where possible. NOD may be any device specified for INPUT under CALL IOCS. Positions outside the range of the buffer are considered to be blank. An invalid NOD causes the call to function as a NOP.

CALL PLOUT(NOD) is a subroutine call to transfer a record from the buffer set associated with device NOD to device NOD. Overlapped operation is initiated when possible. NOD may be any device specified for LIST under CALL IOCS. Positions outside the range of the buffer are lost. The buffer is automatically cleared before formatting begins. An invalid NOD specification causes the CALL to function as a NOP.

```
TEST DEVICE STATUS
-----------------------------------------
IF(PIOC(NOD))1,2,3

  NOD   PLAN device designation
   1    Exit for busy
   2    Exit for invalid device
   3    Normal exit
-----------------------------------------

      CALL PSBFA (101)
      CALL PSBFB(102)
      CALL PDBFA(1)
      CALL PDBFB(2)
    1 DO 25 I=1,2
C     IS DEVICE VALID AND NOT BUSY
    2 IF (PIOC(I)) 2,30,5
C     CHECK OUTPUT DEVICE
    5 IF (PIOC(I+100)) 5,30,10
   10 CALL PLINP(I)
C     TRANSFER INPUT BUFFER TO OUTPUT
C     BUFFER (SEE TRANSFER BUFFER
C     CONTENTS)
      CALL PBFTR (I,I+100)
      CALL PLOUT (I+100)
C     TEST END OF FILE
        •
        •
        •
   25 CONTINUE
      GO TO 1
C     INVALID DEVICE
   30 PAUSE 30
```

The busy status of any device may be tested with the following function test:

```
    IF(PIOC(NOD)) 1,2,3
```

Statement number 1 is executed if the device specified by NOD is busy. Statement number 2 is executed if NOD is an invalid device specification. Statement number 3 is executed if the device specified by NOD is not busy.

```
QUIESCE I/O
-----------------------------------------
  CALL PBUSY
-----------------------------------------
       •
       •
       •
C    WAIT UNTIL ALL I/O IS COMPLETE
     CALL PBUSY
       •
       •
       •
```

CALL PBUSY is a subroutine that returns control to the calling program only when all devices controlled by CALL PLINP and CALL PLOUT are found to be not busy. This call need not be issued before terminating any module in which CALL PLINP or CALL PLOUT are issued because the PLAN loader performs the function. However, any instructions to an operator requiring a change in a device status should be preceded by a call to PBUSY.

```
TRANSFER BUFFER CONTENTS
-----------------------------------------
CALL PBFTR(NODF,NODT)

  NODF  PLAN device code of "from" buffer
  NODT  PLAN device code of "to" buffer
-----------------------------------------
(See "Test Device Status" for example)
```

CALL PBFTR(NODF,NODT) is a subroutine that results in the transfer of data from the buffer associated with NODF to the buffer associated with NODT. The shortest buffer controls the transfer termination.

The following general control routines provide miscellaneous control functions associated with a particular device (NOD).

```
DEVICE CONTROL
-----------------------------------------
CALL PCCTL(NOD,KTL)

  NOD   PLAN device assignment
  KTL   Control function code
-----------------------------------------

        •
        •
        •
C     DOUBLE-SPACE LISTING
      CALL PCCTL (100,-2)
C     IS PAGE RESTORE REQUIRED
      IF (PEOF(100)) 5,5,10
    5 CALL PCCTL (100,1)
C     PRINT
   10 CALL PLOUT(100)
        •
        •
        •
```

CALL PCCTL(NOD,KTL) is a subroutine call that defines a control procedure, to be executed in conjunction with the <u>next</u> input/output operation on device NOD as a result of CALL PLOUT(NOD) or CALL PLINP(NOD). Allowable values of KTL are shown in the following table. The call will be ignored if NOD is a device on which the function may not be executed.

Note carefully that the most recent call to PCCTL prevails. For example, a request for a double-space following a request for a page skip without an intervening call to PLOUT will result in a missed page skip. A page skip request when the carriage is at channel 1 <u>will</u> result in a skip to a new page.

N	Function
1-12	Skip paper to channel N
0	Suppress space before printing (note that suppress space results in no carriage return on the 1130 Console typewriter or 1050 on the DOS System)
-1	Single-space before printing (note that this is automatic)
-2	Double-space before printing
-3	Triple-space before printing
13	Select stacker 1
14	Select stacker 2

```
TEST END-OF-FILE STATUS
------------------------------------------
IF (PEOF(NOD)) 1,2,3

  NOD   PLAN device code definition
   1    Logical EOF on exit
   2    Physical EOF or invalid device
        code exit
   3    Normal exit
------------------------------------------
See "Device Control" for example
```

The physical end-of-file indicator is turned on by reading or punching the last card. The condition may be detected by testing the indicator with the function test PEOF. The logical end-of-file indicator is turned on by reading a record with UREND in positions 1-5 or by sensing channel 12 on the printer. Channel 12 is simulated on OS and DOS PLAN by a line counter. The record containing the logical end-of-file may be accessed by the conversion routines if desired. The logical EOF indicator remains on for only one cycle.

The end-of-file test is a function test as shown below. Statement 1 is executed if the logical end-of-file indicator is "on"; statement 2 is executed if the physical end-of-file indicator is "on" or an invalid device is tested; statement number 3 is executed if neither end-of-file indicator is "on".

```
    IF (PEOF(NOD)) 1,2,3
```

NOD is the device on which the input/output operation was executed that might have set the end-of-file indicator. Note that when the physical end-of-file indicator is turned "on" the device cannot be accessed until PLAN is rescheduled (a new execution of PLAN is initiated).

The procedures below are followed by PLAN (PSCAN) when end-of-file conditions or special input records are processed.

CONDITION	ACTION
LOGICAL EOF (UREND)	Initiate a phrase abort with the appropriate diagnostic.
PHYSICAL EOF	Return to monitor after completion of processing of the current command.
/*	Return directly to monitor.
//	Return control to monitor. A diagnostic is issued and the record is not processed by monitor.

```
DATA CONVERSION ROUTINES
------------------------------------------
CALL PIIN(NOD,I,NW,ARRAY)
CALL PIOUT(NOD,J,NW,ARRAY)
CALL PAIN(NOD,I,NW,ARRAY)
CALL PAOUT(NOD,J,NW,ARRAY)
CALL PFIN(NOD,I,NW,ARRAY)
CALL PFOUT(NOD,J,NW,ARRAY)
CALL PEOUT(NOD,J,NW,ARRAY)

  NOD   PLAN device specification
  I     Input record position from which
        to extract field
  J     Output record position into which
        the field is to be stored
  NW    Number of characters in the field
        or field width and decimal
        position indicator
ARRAY   User array
------------------------------------------

C    LIST SEQUENCE ERRORS IN CARD DECK
     COMMON L(625), LS(15), M(510),
    1K(8), KK(8),A(2)
     DATAA/'ERRO','ORS'/
C    OPEN BUFFER
     CALL PDBFA(0)
     CALL PSBFA(100)
     NERR = 0
C    CLEAR OLD SEQUENCE
     DO 5 I=1,8
   5 K(I) = 0
   6 CALL PLINP(0)
C    GET NEW SEQUENCE
     DO 10 I=1,8
  10 CALL PAIN(0,72+I,1,KK(I))
C     CHECK SEQUENCE
     DO 15 I=1,8
     IF (K(I)-KK(I)) 19,15,35
  15 CONTINUE
C    EQUAL ERROR
     GO TO 35
C    TEST END-OF-FILE
  19 IF (PEOF(0)) 20,20,25
C    SAVE OLD SEQUENCE
  25 DO 30 I=1,8
  30 K(I) = KK(I)
     GO TO 6
C    GENERATE COUNT OF ERRORS
  20 CALL PIOUT(100,1,5,NERR)
     CALL PAOUT(100,6,7,A)
     CALL PLOUT(100)
C    EXIT MODULE
     CALL LRET
C    ERROR IN SEQUENCE
  35 CALL PBFTR(0,100)
     CALL PLOUT(100)
     NERR = NERR+1
     GO TO 19
     END
```

The format control routines provide for transfer and format conversion of data from/to the system buffer associated with the device to/from user-specified memory. The names of routines that transfer data from the system buffer are suffixed with the characters "IN". The names of routines that transfer data to the system buffer are suffixed with the characters "OUT". The second character of the subroutine name specifies the format conversion type.

ROUTINE	CONVERSION
PIIN	Integer input conversion
PFIN	Floating-point input conversion
PAIN	Hollerith (A-format) named input conversion
PIOUT	Integer output conversion of fixed-point numbers
PFOUT	Decimal output conversion of floating-point numbers with rounding
PEOUT	Exponential output conversion of floating-point numbers with rounding
PAOUT	Output conversion of named Hollerith data

The input conversion routines have the following arguments in the call list:

```
    CALL PXIN (NOD,I,NW,ARRAY)
```

Argument	Function
NOD	The input device (see 5.11.5) on which the data to be converted was read as the last record.
I	The relative position within the record of the first character of the field to be converted.
NW	The number of positions within the record to be converted, starting at I. In PFIN, the tens and hundreds positions define the field width. The units position defines the number of decimal positions in the data field. An included decimal point or explicit exponent overrides the units position specification.
ARRAY	The 32-bit word that is to receive the converted data value.

The output conversion routines have the following arguments in the call list:

```
    CALL PXOUT (NOD,J,NW,ARRAY)
```

Argument	Function
NOD	The output device (see CALL IOCS under "Utility Subroutines", 5.11.5) on which the converted data record is to be written on the next output record.
J	The relative position within the output buffer at which the first position of the converted data is to be stored.
NW	The number of positions to be occupied by the converted data field, starting at J. Data converted by PIOUT, PFOUT, and PEOUT is right-justified. Leading zeros are suppressed. In PFOUT, and PEOUT the tens and hundreds positions define the field width. The units position defines the number of decimal positions. If possible, the exponential format is used when the field width is too small to allow the decimal format.
ARRAY	The location in memory from which the data to be converted is to be taken.

The following rules for these routines indicate action taken by the PLAN I/O conversion routines under various data conditions:

<u>Routine</u> <u>Conversion Rules</u>

PIIN
1. Leading blanks to and following the sign are ignored.
2. Signs may be +, -, or &.
3. Digits are collected after the first numeric until the specified field is processed or a nonnumeric character is processed.
4. A fixed-point zero is the result of processing no numerics.
5. All numbers are treated modulo (the maximum positive or maximum negative fixed-point number).
6. Example:
   b-b156 = -156
   1-56 = 1
   bbb.b = 0

PFIN
1. Leading blanks to the left of the sign or within an exponent field are ignored.
2. Other blanks are treated as zero.
3. Input collection is stopped when a second decimal is processed.
4. Exponents may be represented and preceded by E, by leading blanks, and a sign, or by a sign.
5. The collection is stopped by a nonblank, nonnumeric following the exponent.
6. Numbers too small to be represented are set to zero.
7. Numbers too large to be represented are set to FALSE.
8. Example:
   bb+5b.b6 = 50.06
   7bbb+b5 = 700000000
   -.7E3 = -700
   1.E1bb=10

PAIN	1. Characters are collected and placed in successive array words in A4 format. 2. The unused portion of the last word is not disturbed.
PIOUT	1. Leading zeros are suppressed. 2. The sign, if minus, is placed adjacent to the leftmost digit. 3. Truncation is from the left with the sign being truncated first.
PFOUT	1. Leading zeros to the left of the decimal point are suppressed. 2. No position is required for the sign unless it is negative. Any minus sign is placed to the left of the most significant digit or the decimal point if there are no digits left of the decimal point. 3. Truncation from the right is automatic. 4. Truncation from the left results in a call to PEOUT. 5. If (W-D-S-M-1) is less than zero, the call is treated as a call to PEOUT. W = number of output positions D = number of digits to right of decimal S = number of significant digits left of decimal point M = 1 if the number is negative and 0 if the number is positive
PEOUT	1. No sign position is required unless the sign is negative. 2. Positive exponents are in the form: E±nn 3. If (W-D-M-S) is equal to or less than 0 when D=0, then D is set equal to W-M-6. If this D is negative, the output field is set to asterisks.
PAOUT	1. Characters are transferred from successive array words that are assumed to be in A4 format.

The following example illustrates the sample program to read and list cards and accumulate the total of the data punched in columns 21-27 of the data cards. The data deck is terminated with a card punched UREND (logical EOF) in cc. 1-5.

```
      CALL PDBFA(1)
      CALL PDBFB(101)
      SUM = 0.
      CALL PCCTL (101,1)
    5 CALL PLINP(1)
      IF (PEOF(1)) 20,20,15
   15 CALL PFIN(1,21,70,A)
      SUM = SUM+A
      CALL PBFTR(1,101)
      CALL PLOUT (101)
      IF (PEOF(101))5,5,10
   10 CALL PCCTL(101,1)
      GO TO 5
   20 CONTINUE
           •
           •
      END
```

Note that a buffer may be used for both output conversion routines and input conversion routines. This allows a user to utilize the facilities of the conversion routines for converting internal data.

The following example illustrates a program that reads a deck of cards, and adds a three-character identification field and five-position sequence field before reproducing the deck on the system card punch and listing it on the system printer. The starting sequence number, increment, and identification field are read at the start of each deck.

```
      COMMON L(625),LS(15),M(510)
C     ASSIGN BUFFER FOR CONTROL INFORMATION
C     INPUT
      CALL PSBFA (3)
C     ASSIGN PUNCH FILE BUFFER
      CALL PDBFB(104)
C     ASSIGN PRINT FILE BUFFER
      CALL PSBFB(102)
C     ASSIGN READ FILE BUFFER
      CALL PDBFA(2)
C     READ CONTROL INFORMATION
   10 CALL PLINP(3)
C     CONVERT ID FIELD FROM POSITIONS 1-3
      CALL PAIN(3,1,3,A)
C     CONVERT STARTING SEQUENCE NUMBER
C     POSITIONS 5-9
      CALL PIIN(3,5,5,I)
C     CONVERT SEQUENCE INCREMENT FROM
C     POSITIONS 10-14
      CALL PIIN(3,10,5,INC)
C     EJECT TO NEW PAGE
   15 CALL PCCTL(102,1)
C     READ CARD
   20 CALL PLINP(2)
C     MOVE RECORD TO PUNCH - PRINT BUFFER
      CALL PBFTR(2,104)
      CALL PBFTR(2,102)
C     MOVE IDENT TO OUTPUT
      CALL PAOUT(102,73,3,A)
      CALL PAOUT(104,73,3,A)
C     MOVE SEQUENCE TO OUTPUT
      CALL PIOUT(102,76,5,I)
      CALL PIOUT(104,76,5,I)
C     FILL LEADING ZEROS
```

```
      DO 23 K=76,79
      CALL PAIN(102,K,1,KK)
      CALL PUNPK (KK,KK,1)
      IF (KK-64) 21,21,24
   21 CALL PAOUT(102,K,1,240)
   23 CALL PAOUT(104,K,1,240)
C     WAS EOF PROCESSED ON READ
   24 IF (PEOF(2))10,30,25
C     PUNCH CARD
   25 CALL PLOUT(104)
C     PRINT
      CALL PLOUT(102)
C     INCREMENT SEQUENCE
      I=I+INC
C     IS THIS LAST LINE ON PAGE
      IF(PEOF(102))15,15,20
C     RETURN TO PLAN ON PHYSICAL EOF
   30 CALL LRET
      GO TO 30
      END
```

If NOD equals zero for any function associated with input, the current PLAN input device is used. Use of 100 for any function associated with output results in use of the current PLAN output device (see 5.11.5).

The following example illustrates the truncation procedures of PIOUT. A FORTRAN program is shown followed by the output attained:

```
      DATAV1/'A='/,V2/'B='/,V3/'C='/,V4/'D='/,
      V5/'E='/,V6/'F='/
      DATAV7/'G='/,V8/'H='/,V9/'I='/,VA/'J='/
      CALL PSBFA(12)
      CALL PAOUT(102,01,2,V1)
      CALL PIOUT(102,03,8,-32767)
      CALL PAOUT(102,11,2,V2)
      CALL PIOUT(102,13,7, 32767)
      CALL PAOUT(102,20,2,V3)
      CALL PIOUT(102,22,6,-32767)
      CALL PAOUT(102,28,2,V4)
      CALL PIOUT(102,30,5, 32767)
      CALL PAOUT(102,35,2,V5)
      CALL PIOUT(102,37,5,-32767)
      CALL PLOUT(102)
      CALL PAOUT(102,42,2,V6)
      CALL PIOUT(102,44,4, 32767)
      CALL PAOUT(102,48,2,V7)
      CALL PIOUT(102,50,3, 32767)
      CALL PAOUT(102,53,2,V8)
      CALL PIOUT(102,55,2, 32767 )
      CALL PAOUT(102,57,2,V9)
      CALL PIOUT(102,59,1, 32767)
      CALL PAOUT(102,60,2,VA)
      CALL PIOUT(102,62,59,0)
      CALL PLOUT(102)
    1 CALL LRET
      GO TO 1
      END
```

```
A=   -32767B=   32767C=-32767D=32767E=32767
F=2767G=767H=67I=7J=
```

5.11.10 ARRAY MANIPULATION

```
ARRAY MANIPULATION

CALL PARGO(LS,ARRAY)
CALL PARGI(LS,ARRAY)
CALL GTVAL(ARRAY,KOUNT,DATA,NSUB)
CALL STVAL(ARRAY,KOUNT,DATA,NSUB)

  LS    Switch word containing pointer
  ARRAY User data array
  KOUNT Words to transfer
            (TO array for GTVAL)
            (The FROM array for STVAL)
  DATA  User data array
            (The FROM array for GTVAL)
            (The TO array for STVAL)
  NSUB  Data array initial subscript

C    MOVE ARRAY C-A-D-B-A
     COMMON L(625),LS(15),NA(10),NB(10),
     NC(10),ND(10)
     DO 5 I=1,10
     NA(I)=0
     NB(I)=10
     NC(I)=20
   5 ND(I)=30
     LS(4)=21
     NC(1)=9
     CALL PARGO(4,NA)
C    TRANSFER A TO D
     CALL GTVAL(ND,10,NA,1)
     LS(4)=11
     CALL PARGI(4,ND)
C    TRANSFER B TO A
     CALL STVAL(NB,10,NA,1)
C    ARRAYS SHOULD BE EQUAL
     DO 25 I=1,31,10
     IF (NA(I)-9)10,15,10
  10 CALL ERROR (111,NA(I),0)
  15 KK=I+1
     KKK=KK+8
     DO 25 K=KK,KKK
     IF(NA(K)-20) 20,25,20
  20 CALL ERRET(112,NA(K),0)
  25 CONTINUE
     CALL LRET
     GO TO 25
```

Subroutines PARGO and PARGI provide a mechanism for easy manipulation of data through system Switch Words 4-7.

Arrays to be transmitted by PARGO and PARGI must be in the following PLAN array format.

FIXED-POINT COUNT OF DATA VALUES	1ST ELEMENT	2ND ELEMENT	...	LAST ELEMENT

If the resulting "TO" array address is lower in COMMON than the communication array, the call is ignored. No diagnostic is issued.

CALL PARGO(LS,ARRAY) moves a data array from ARRAY to the PLAN communication array pointed to by PLAN Switch Word LS. Array (1) is assumed to contain the number of words "M" that are to be transferred from Array(2) to Array(M+1). The count is transferred to the position indicated in Switch Word LS. The data list is transferred to the following positions.

CALL PARGI(LS,ARRAY) moves a data list from the PLAN communication array (pointed to by PLAN Switch Word LS) to ARRAY. The first position of ARRAY receives the integer count of the number of data values. The remainder of the array contains the data list.

The following example shows use of the PARGO routine in transferring array F, in a module, to communication array location 20. Example:

```
    COMMON L(625), LS(15), M(255)
    LS(4)=20
    CALL PARGO(4,F)
```

In the above example, if F(1) contained a 10, then communication array position 20 would be set to 10 and F(2) through F(11) would be transferred to communication array positions 21 through 30.

The following routines allow easy, efficient transmission of arrays or parts of arrays to and from any location in storage. Note carefully that arrays to be processed must start on 32-bit boundaries.

CALL STVAL(A,N,B,I)) moves N 32-bit words from A(1) through A(N) to B(I) through B(I+N-1).

CALL GTVAL(A,N,B,I) moves N 32-bit words B(I) through B(I+N-1) to A(1) through A(N).

5.11.11 BIT, BYTE AND CHARACTER PROCESSING

```
COMPARE LOGICAL ARRAY
-----------------------------------------
IF(PCOMP(A,B,N)) 1,2,3

  A     User's first data array
  B     User's second data array
  N     Number of words to compare
  1     A less than B exit
  2     A equals B exit
  3     A greater than B exit
-----------------------------------------

C    1130 FLOATING-POINT USED IN EXAMPLE

     DIMENSION A(5),B(5)
     COMMON L(625),LS(15),M(255)
     DATA A/'ABCD'/, B/'BCDE'/
     DO 10 J=1,5
     A(J)=J
  10 B(J) = J
     IF (PCOMP(A,B,5)) 90,20,90
  20 A(3)=2.
C    COMPARE HEX 40000082 WITH 60000083
     IF (PCOMP(A,B,5)) 30,80,80
  30 A(3)=40.
C    COMPARE HEX 50000086 WITH 60000083
     IF(PCOMP(A,B,5)) 40,80,80
  40 IF(PCOMP(B,A,5)) 70,70,50
  50 CALL LRET
C    BRANCH HIGH ERROR
  70 PAUSE 70
     GO TO 50
C    BRANCH LOW ERROR
  80 PAUSE 80
     GO TO 50
C    BRANCH EQUAL ERROR
  90 PAUSE 90
     GO TO 50
     END
```

Two 32-bit arrays may be logically compared through use of the PCOMP function. The following example illustrates the FORTRAN statement necessary to compare five words (32-bit logical) of array A to five words of array B:

```
IF(PCOMP(A,B,5)) 1,2,3
```

Statement number 1 is executed if the first word of array A that is not equal to the corresponding word of array B is less than the word in array B. Statement number 2 is executed if the entire array A is equal to array B. Statement number 3 is executed if the first unequal word of array A is greater than the corresponding word of array B. Care should be taken when comparing numeric arrays to note the logical order of bits.

```
HEXADECIMAL TO EBCDIC
-----------------------------------------
CALL PHTOE(A,B,N)

  A     Hexadecimal array
  B     EBCDIC converted array
  N     Number of words in array A
-----------------------------------------

C    LIST SWITCH WORDS
     COMMON L(625),LS(15),M(510)
     DIMENSION W(30)
     CALL PSBFA(100)
     CALL PHTOE(LS,W,15)
     DO 10 I=1,15
     CALL PAOUT(100,10*I-9-I/11*100,8,
    1W(2*I-1))
     IF (I-10) 10,5,10
   5 CALL PLOUT(100)
  10 CONTINUE
     CALL PLOUT(100)
        •
        •
        •
```

CALL PHTOE(A,B,N) is a subroutine call that converts a hexadecimal array A to an EBCDIC array B in A4 format. N words from array A are converted to 2*N words in array B. PHTOE allows any data to be dumped as a hexadecimal listing. The following example shows array A and array B (hexadecimal representation) following the call to PHTOE:

```
Array A: 60000082C1C2C3C4
Array B: F6F0F0F0F0F0F8F2C3F1C3F2C3F3C3F4
```

```
LOGICAL TEST FUNCTIONS

CALL PBTST(NOP,NWORD,NBIT,NSKIP)

  NOP   Operation to perform
  NWORD Word upon which test is to be
        made
  NBIT  Bit to test or test mask
  NSKIP Test result indicator

C     CLEAR WORD
      CALL PBTST(0,A,0,NSKIP)
C     TURN EVERY OTHER BIT ON
      DO 5 I=1,31,2
    5 CALL PBTST(1,A,I,NSKIP)
C     SET MASK TO EXTRACT ALL BUT BITS
C     7-15 FROM WORD
      CALL PBTST(-1,B,0,NSKIP)
      DO 10 I=7,15
   10 CALL PBTST(3,B,I,NSKIP)
C     EXTRACT
      CALL PBTST(12,A,B,NSKIP)
C     BITS 1,3,5,17,19,21,23,25,27,29,
C     AND 31 SHOULD BE ON
C     LIST BITS ON AND TURN THEM OFF
      CALL PSBFA(100)
      DO 15 I=1,32
      CALL PBTST(7,B,I-1,NSKIP)
      IF (NSKIP)12,15,12
   12 CALL PIOUT(100,1,5,I-1)
      CALL PLOUT (100)
   15 CONTINUE
         •
         •
         •
```

CALL PBTST(NOP,NWORD,NBIT,NSKIP) is a subroutine call to provide manipulative and testing functions for any of the 32-bits of the 32-bit word NWORD. The bit specified by NBIT is in the range of 0-31. NOP specifies the setting, resetting, or testing operation to be executed. If a test operation is defined and all bits in the test mask match or the specified bit is on, NSKIP is set to one; if no bits in the test mask match or the specified bit is off, NSKIP is set to zero; if only part of the bits match, NSKIP is set to minus one. If NOP specifies a test under mask operation, NBIT is the test mask rather than a bit position. If NOP specifies an extract under mask, NBIT is the PLAN word that is to receive the extracted field and that contains the extraction mask. The following table lists the valid operation codes:

OP CODES	FUNCTION
-1	Turn all bits "on"
0	Set all bits "off"
1	Set the specified bit "on" (1)
2	Invert the specified bit; if 1 set to 0, if 0 set to 1
3	Set the specified bit "off" (0)
4	Test the specified bit
5	Test the specified bit and set it "on"
6	Test the specified bit and invert
7	Test the specified bit and set it "off"
8	Test the bits corresponding to the specified mask
9	Test the bits corresponding to the test mask and set them "on"
10	Test bits corresponding to the test mask and invert them
11	Test the bits corresponding to the test mask and set them "off"
12	Test and extract the bits corresponding to the bits in the test mask

The following table illustrates the functions of various calls to PBTST and shows values before and after execution:

NOP	NWORD BEFORE	NBIT	NSKIP	NWORD AFTER
-1	7FFFFFFF	-	-	FFFFFFFF
0	7FFFFFFF	-	-	00000000
1	7FFFFFFF	0	-	FFFFFFFF
2	7FFFFFFF	1	-	3FFFFFFF
2	7FFFFFFF	0	-	FFFFFFFF
3	7FFFFFFF	1	-	3FFFFFFF
3	7FFFFFFF	0	-	7FFFFFFF
4	7FFFFFFF	1	1	7FFFFFFF
4	7FFFFFFF	0	0	7FFFFFFF
5	7FFFFFFF	1	1	7FFFFFFF
5	7FFFFFFF	0	0	FFFFFFFF
6	7FFFFFFF	1	1	3FFFFFFF
6	7FFFFFFF	0	0	FFFFFFFF
7	7FFFFFFF	1	1	3FFFFFFF
7	7FFFFFFF	0	0	7FFFFFFF
8	7FFFFFFF	F0000000	0	7FFFFFFF
8	7FFFFFFF	70000000	1	7FFFFFFF
9	7FFFFFFF	F0000000	0	FFFFFFFF
9	7FFFFFFF	70000000	1	7FFFFFFF
10	7FFFFFFF	F0000000	0	8FFFFFFF
10	7FFFFFFF	70000000	1	0FFFFFFF
11	7FFFFFFF	F0000000	-1	0FFFFFFF
11	4FFFFFFF	70000000	1	0FFFFFFF
12	7FFFFFFF	F0000000	1	7FFFFFFF
	NBIT after 70000000			
12	7FFFFFFF	70000000	1	7FFFFFFF
	NBIT after 70000000			

```
BYTE MANIPULATION

CALL PPACK(I,A,N)
CALL PUNPK(I,A,N)
CALL BREAK(A,J)

  A     Word containing four bytes
  I     Integer word with one
        right-justified byte
  N     Byte position indicator
  J     Four-word integer array

C     REVERSE ORDER OF FOUR BYTES
      DATA A/'ABCD'/
      DIMENSION N(4)
      CALL BREAK(A,N)
      DO 5 I=1,4
    5 CALL PPACK(N(I),A,5-I)
C     GET 2ND BYTE AND TEST FOR 'C'
      CALL PUNPK(N,A,2)
      IF (N(1)-195) 10,15,10
C     ERROR
   10 PAUSE 10
   15 CONTINUE
        •
        •
        •
```

CALL PPACK(I,A,N) is a subroutine that masks the rightmost eight bits of the integer I into the byte position of array A specified by N. Other bytes within A are unchanged.

The following example illustrates a call to PPACK to place the letter B (decimal equivalent is 194) into bits 0-7 of the word at A. Example:

```
    CALL PPACK(194,A,1)
```

CALL PUNPK(I,A,N) is a subroutine that inserts the byte specified by N in array A into the rightmost byte of the integer word I. Bits to the left of the inserted byte in I are cleared.

CALL BREAK(A,J) is a subroutine that spreads the four bytes of word A into the low-order byte of the four-word integer array J. High-order bits in array J are set to zero. This subroutine call is useful in separating alphameric data in A4 format into a form that allows ready testing within FORTRAN. The following FORTRAN statements test a literal string and indicate the position of the first comma encountered. The string is assumed to be stored in array A. The location of the comma will be stored in J1.

```
    DIMENSION I(4),A(...
      J=1
    5 J1=J/4+1
      CALL BREAK(A(J1),I)
   15 J1=J-(J-1)/4*4
      IF(I(J1)-107) 20,25,20
   20 J=J+1
      IF(J1-4) 15,5,15
   25 CONTINUE
        .
        .
        .
```

6.0.0 PROGRAMMING CONVENTIONS

This section provides a guide to, and the regulations for, writing PLAN logic modules. A PLAN logic module is a piece of program code that has been singly compiled or assembled and stored/link-edited into the program library. The application (problem-solver) programmer follows the regulations for writing and storing a normal mainline program but observes the standards of PLAN as discussed below.

There are specific functions that should not be used since they are detrimental (or fatal) to the successful PLAN execution. These functions vary between different systems and are specifically detailed in the appropriate appendix of this manual. In general, any function that gives control to the monitor or operating system must not be used when PLAN has control. Where functions are specifically restricted because of adverse performance, an equal or more powerful function is provided by a PLAN subroutine or function. Linkage conventions compatible with those of FORTRAN are used in all versions of PLAN.

6.1.0 COMMON LAYOUT

The common statement in any program must protect PLAN by providing the proper COMMON layout. The items listed below are the items (some optional) included in COMMON. Sizes are stated in 32-bit words.

Item:	Loader
Size:	625 (Required)
Function:	This portion of the loader area (in COMMON) contains the PLAN loader and must remain in memory throughout an entire PLAN execution (until PLAN returns control to the monitor or operating system).

Item:	System Switch Words
Size:	15 (Required)
Function:	A communication control area required for controlling PLAN but accessible to the user for modification by program or command.

Item:	Managed Array
Size:	Variable (Optional)
Function:	This array (in COMMON) is that portion of the communication array that is to be managed according to the PLAN level concept. Communication between the application program and problem-describing commands is through the communication array (managed and nonmanaged). The size of the managed array is described to the PLAN system at language definition time (see "Switch Words"); however, if undefined, it is assumed to have a length of 0 words.

Item:	Nonmanaged Array
Size:	Variable (Optional)
Function:	This array (in COMMON) plus the previously described managed array constitute the communication array, that is, the area used for communication of data input via PLAN commands to the application logic modules and between various logic modules. This portion of the communication array, however, is not managed according to PLAN level designations. This array may also be described as that portion of COMMON -- not included in the loader area, system switch words, or managed array -- that will not be overlaid by any PLAN system module, such as the statement scan module (PSCAN). The size of this array may be variable and depends on (1) the size of the managed array, and (2) the system configuration on which this run is executed. On 1130 PLAN the minimum size of COMMON protected from overlay by PLAN modules is 510 words. Execution of ADD PHRASE commands may overlay some of the communication array. (See the appropriate appendix for specific communication array size specifications.)

Item:	User COMMON (1130 PLAN only)
Size:	Variable (Optional)
Function:	This portion of COMMON may contain any desired user arrays required for transfer of data between logic modules. Certain PLAN system functions (logic modules) may, however, overlay this portion of COMMON. These functions are:

Module	Used When
PSCAN	A new PLAN statement must be processed.
PERRS	A system error diagnostic must be generated by the PLAN diagnostic processor.

PSRTA	The PLAN file sort facilities are required.
PMRGA	The PLAN file merge facilities are required.
PHRAS	A new command is to be entered into the language dictionary.

7.0.0 PLAN SYSTEM CASE STUDY

This section shows the programming, language definition, and language use involved in an application system designed to generate solutions to a problem. All the facilities of PLAN are not illustrated. The section serves to illustrate proper formation of PLAN modules and PLAN commands.

7.1.0 PROBLEM DEFINITION

The problem to be solved in this case study development will be defined in increasing degrees of complexity. An initial problem definition is solved through all involved steps, including the FORTRAN programs required. Additional requirements are then added to the problem definition, and the steps to attain the new solution are added.

The following geometric equivalences may be derived from the elements defined in Figure 12.

TAN θ = BEVEL
SLOPE^2 = BASE^2 + RISE^2
SIN θ = RISE/SLOPE
COS θ = BASE/SLOPE

Figure 12 illustrates the terminology used in various aspects of this case study.

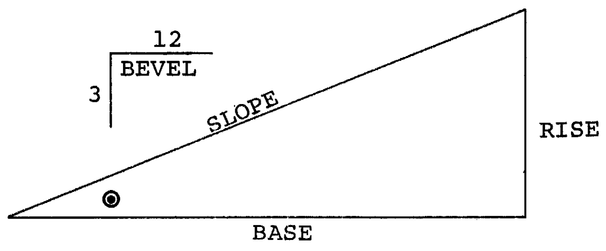


Figure 12. Terminology for sample problem

The problem to be solved in this case study is defined below. Two solutions to the problem are provided. The problem to be solved is:

1. Solve a right triangle when any two of the five values (BASE, RISE, SLOPE, BEVEL, ANGLE) are given. Note that BEVEL and ANGLE may not be the given two values since they are mutually exclusive. Assume that values will be given in decimal. Results should be rounded. Suggested logic modules might be (1) CALCULATION, (2) PRINT, and (3) CSERR.

   Provide a facility for validating data to ensure that two-out-of-five values are given. The errors that could occur are underspecification and overspecification. Note also that specification of ANGLE and BEVEL is a special case of underspecification. This testing may be accomplished with either logical testing within a phrase or a separate logical testing module.

7.2.0 LANGUAGE DEFINITION

The following list is an itemization of the functions to be performed through problem-oriented language statements. These are as follows:

1. The system (communication array) must be initialized, that is, a level 1 command must be defined.

2. Data items must be named to allow input of any desired value.

3. Literal information must be defined for heading printouts.

The following section provides the elements of the phrase definition and a brief description of the function achieved:

ELEMENT:	ADD PHRASE: TRIANGLE SOLUTION,
FUNCTION:	The command TRIANGLE SOLUTION is defined, and will be recognized as the dictionary entry pointer to the following context definition.
ELEMENT:	LEVEL 1,
FUNCTION:	The phrase TRIANGLE SOLUTION does not depend on any other phrase or data. The entire managed communication array is to be initialized to logical FALSE when the phrase is encountered.
ELEMENTS:	(70)BASE, (73)RISE, (76)SLOPE, (79)ANGLE, (82)BEVEL,
FUNCTION:	The possible input values (BASE, RISE, SLOPE, ANGLE, BEVEL) are named and assigned positions within the communication array.
ELEMENT:	(20)TEST+*T',,'*T P'CSERR ',
FUNCTION:	Position 20 of the communication array is assigned the name TEST. A logical TRUE is defined as an initialization value. A test is defined to check for the pres-

ence of logical TRUE. If the location is not found to contain logical TRUE, the phrase named CSERR is entered into the PLAN input stream and is executed. Note that check-entries are not evaluated until all expressions have been evaluated (see "PSCAN Execution Sequence", 4.3.25).

ELEMENT: PROGRAMS 'CALC',
FUNCTION: Program CALC is to be placed in the pop-up list whenever the command TRI SOL is encountered.

ELEMENTS: I(21)N0,J1,I(9)K0,
FUNCTION: Three counters are defined and initialized for use in the formula evaluations defined below.

ELEMENTS: I(-8)M, (M)A'DUM ',
FUNCTION: A literal is set that is used to cause execution of the DUMP command whenever a program error is detected (see "Standard PLAN Commands", 4.5.0).

ELEMENTS:
```
$4:¬BASE(J)?$6, N=N+1, $6 J=J+3,
:(J=16)?$7!$4,       $7:(N=2)?$9,
$8TEST=-,  :$20, $9:(BAS<0)?$11,
:(RIS<0)?$12,  :(SLO<0)?$13,  :(
ANG<0)?$14,   :(BEV<0)?$15,   :(
ANG>90)?$16,  ;(ANG)  &  (BEV)?$
17!$20,  :$20,  $11K=1,  :$8,$12
K=2, :$8, $13K=3, :$8, $14  K=4,
:$8,  $15 K=5, :$8, $16K=6, :$8,
$17K=7, :$8, $20;
```
FUNCTION: The formula area is utilized to test the possible data values for FALSE. If the value is not FALSE, a count (N) is made. After all five possible data values have been tested, a test is made to determine if two non-FALSE data values exist. If the number of non-FALSE values is not two, TEST is set to FALSE. A test is also made to assure that the two given values are not ANGLE and BEVEL. Note that a value of FALSE in TEST causes the check-entry defined above to fail and thereby place CSERR in the pop-up list. A negative value for RISE, BASE, ANGLE, BEVEL, or SLOPE or an angle greater than 90 results in TEST being set to FALSE.

Figure 13 illustrates the logic of the expressions defined in the preceding element.

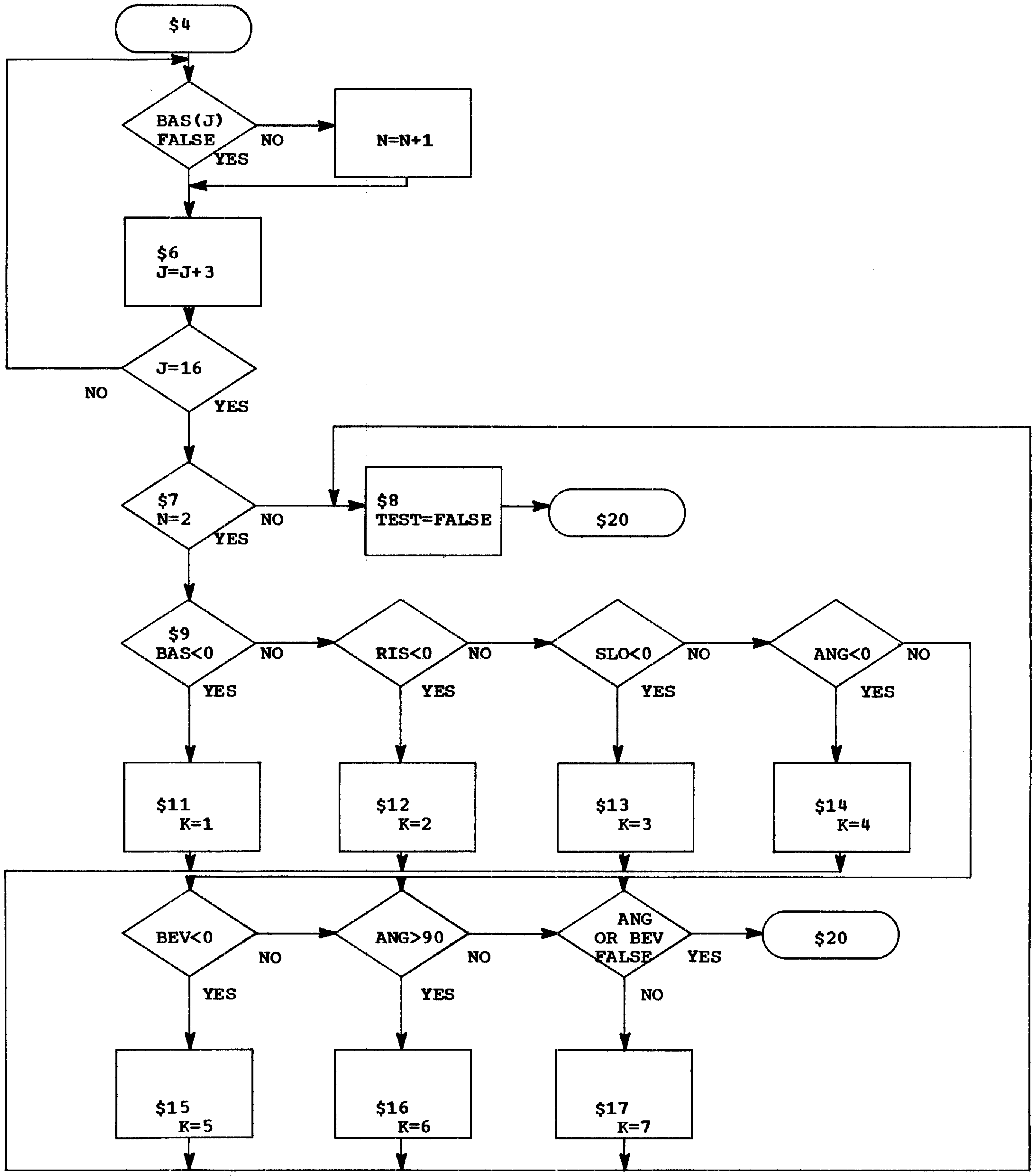


Figure 13. Expression logic

The segments of the phrase defined above are combined here in their entirety but condensed to a minimal representation. Spaces in the following example must be eliminated to prevent exceeding the 450-character statement length limit.

```
ADD PHR:  TRI SOL, LEV1, (70)BAS,  (73)RIS,
(76)SLO, (79)ANG, (82)BEV, (20)TEST+*T',,',
*TP'CSERR",PRO'CALC',I(21)N0,  J1,  I(9)K0,
I(-8)M,   (M)A'DUM COM ',      $4:¬BAS(J)?$6,
N=N+1,  $6J=J+3,:  (J=16)?$7!$4, $7:(N=2)?$
9, $8TES=-,:$20,  $9:(BAS<0)?$11,:(RIS<0)?$
12,  :(SLO<0)?$13, :(ANG<0)?$14, :(BEV<0)?$
15, :(ANG>90)?$16, :(ANG) &  (BEV)?$17!$20,
:$20,  $11K=1,  :$8,$12  K=2,  :$8, $13K=3,
:$8, $14 K=4, :$8, $15  K=5,  :$8,  $16K=6,
:$8, $17K=7, :$8, $20;
```

An additional phrase is added to the system to provide control for printing of results.

PHRASE: ADD PHRASE: ANSWER, (30)BASE-, RISE-, SLOPE-, ANGLE-, BEVEL-, (38)ALL-, (40)'BASE =', (45) 'RISE =', (50)'SLOPE =', (55) 'ANGLE =', (60)'BEVEL =', I(39) N5, PROGRAM'PRINT';

FUNCTION: This phrase allows a user to request results by entering a request for answers and indicating the items to be printed. An entry of ALL indicates that all five values are to be printed. The alphameric constants are provided as phrase-defined literals. The PRINT program is to be placed into the pop-up list each time the ANSWER phrase is encountered.

7.3.0 PROGRAMMING

This section provides the code for the FORTRAN modules CALC, PRINT, and CSERR.

```
C     THE NAME OF THIS PROGRAM IS CALC
C     IT PROVIDES TRIANGLE SOLUTIONS
C     DEFINE PLAN PROTECTION COMMON
      COMMON L(625), LS(15), M(255)
      EQUIVALENCE(BASE,M(10)),(RISE,M(11))
     1,(SLOPE,M(12)),(PBEVE,M(82)),
     2(PBASE, M(70)), (PRISE, M(73)),
     3(PSLOP, M(76)), (PANGL, M(79)),
     4(ANGLE, M(13)), (BEVEL,M(14)), (MX,
     5LS(8)),
      BASE = PBASE
      RISE = PRISE
      SLOPE = PSLOP
      ANGLE = PANGL
      BEVEL = PBEVE
C     DETERMINE WHICH VALUES ARE GIVEN
      K2=-1
C     SET LOOP TO CHECK FIVE VALUES
      DO 5 I=10,14
C     IS VALUE FALSE
      IF (NDEF(M(I)))5,5,10
C     SEE "NDEF" UNDER "PLAN SUBROUTINE
C     USE"
C     CONTINUE LOOP
    5 CONTINUE
C     IF LOOP FALLS THROUGH TRI SOL COMMAND
C     HAS ERROR
C     RETRIEVE DUMP LITERAL AND EXECUTE AS
C     COMMAND
C     LITERAL PLACED IN ERASABLE COMMON
C     EXECUTE COMMAND TO DUMP
C     COMMUNICATION ARRAY
    6 CALL PUSH(M(MX))
C     IS THIS FIRST DEFINED VALUE FOUND
   10 IF (K2) 15,20,20
C     SET TO SECOND INDEX
   15 K2=0
C     SAVE FIRST VALUE INDEX
      K1 = I-9
C     IS INDEX VALID
      IF (K1-4) 5,6,6
C     SAVE SECOND VALUE INDEX
   20 K2 = I-9
C     SELECT ON FIRST VALUE
      GO TO (25,30,35),K1
C     BASE IS KNOWN
C     SELECT ON SECOND VALUE
   25 GO TO (6,40,60,65,75),K2
C     RISE IS KNOWN
   30 GO TO (6,6,80,85,95),K2
C     SLOPE IS KNOWN
   35 GO TO (6,6,6,100,110),K2
C     BASE AND RISE ARE KNOWN - CALCULATE
C     OTHERS
   40 SLOPE = SQRT (BASE*BASE+RISE*RISE)
   45 BEVEL = RISE/BASE
      N1 = 1
      GO TO 130
C     RETURN TO PLAN LOADER
   55 CALL LRET
C     BASE AND SLOPE ARE KNOWN
   60 RISE = SQRT (SLOPE*SLOPE - BASE*BASE)
      GO TO 45
C     BASE AND ANGLE ARE KNOWN
   65 N1 = 1
      GO TO 120
   70 SLOPE = BASE/COS(TEMP)
      RISE = SQRT (SLOPE*SLOPE - BASE*BASE)
      GO TO 55
C     BASE AND BEVEL ARE KNOWN
   75 N1 = 2
      GO TO 130
C     RISE AND SLOPE ARE KNOWN
   80 BASE = SQRT (SLOPE*SLOPE-RISE*RISE)
      GO TO 45
C     RISE AND ANGLE ARE KNOWN
   85 N1 = 3
      GO TO 120
   90 SLOPE = RISE/SIN(TEMP)
      GO TO 80
C     RISE AND BEVEL ARE KNOWN
   95 N1 = 4
      GO TO 130
C     SLOPE AND ANGLE ARE KNOWN
  100 N1 = 5
      GO TO 120
  105 RISE = SLOPE * SIN(TEMP)
      BASE = SLOPE * COS (TEMP)
      GO TO 55
```

```
C      SLOPE AND BEVEL ARE KNOWN
  110 N1 = 6
      GO TO 130
  120 TEMP = ANGLE*.0174532965
      GO TO (125,70,125,90,125,105),N1
  125 BEVEL = SIN(TEMP)/COS(TEMP)
      GO TO(70,70,90,90,105,105),N1
  130 ANGLE = ATAN(BEVEL)*57.2958
      GO TO (55,120,95,120,105,120),N1
      END


C      THE NAME OF THE PROGRAM IS PRINT
C      IT PRINTS THE RESULTS OF TRIANGLE
C      SOLUTIONS
      COMMON L(625), LS(15), M(255)
      EQUIVALENCE (N,M(39)), (NS,M(38))
C      SET OUTPUT DEVICE BUFFER
      CALL PSBFA(100)
C      SET LOOP TO PROCESS FIVE VALUES
      DO 15 I = 1,N
C      ARE ALL VALUES TO BE PRINTED
      IF (NDEF(NS)) 5,10,5
C      IS THIS VALUE TO BE PRINTED
    5 IF (NDEF(M(I+29)))15,10,15
C      PLACE DATA NAME IN BUFFER
   10 CALL PAOUT (100,5,M(5*I+35),
     1M(5*I+36))
C      PLACE VALUE IN BUFFER
      CALL PFOUT (100,6+M(5*I+35), 83,
     1M(I+9))
C      WRITE FROM BUFFER
      CALL PLOUT (100)
   15 CONTINUE
   20 CALL LRET
C      SINCE A FORTRAN PROGRAM CANNOT END
C      WITH A CALL TO A SUBROUTINE, THE
C      FOLLOWING DUMMY INSTRUCTION IS
C      INSERTED TO SATISFY THE FORTRAN
C      COMPILER
      GO TO 20
      END
```

The command CSERR is executed if error conditions are found while executing the TRIANGLE SOLUTION command. The function of this command is to set up literal information for logging of errors.

ADD PHRASE: CSERR, PROGRAM'CSERR', (1)'IS 'IN ERROR', (25)'SPECIFICATION ERROR', I(11)NOD100, (40)'BASE ', (45)'RISE ', (50) (55)'ANGLE', (60)'BEVEL';

The module to process the error indicators is given below:

```
C      THE NAME OF THIS PROGRAM IS CSERR
C      IT LISTS ERRORS IN TRIANGLE SOLUTION
C      DEFINITIONS
C      DEFINE PLAN COMMON
      COMMON L(625), LS(15), M(255)
      EQUIVALENCE (K,M(9)),(NOD,M(11))
C      ESTABLISH BUFFER FOR PLAN DIAGNOSTIC
C      DEVICE
      CALL PSBFA(NOD)
C      FIND THE ERROR
      LDX = K+1
      GO TO (5,25,35,40,45,47,45,50), LDX
C      OTHER THAN TWO VARIABLES SPECIFIED
C      IN TRI SOL COMMAND
C      INVALID DEFINITION LITERAL
    5 CALL PAOUT(NOD,1,M,M(2))
      CALL PLOUT(NOD)
C      LIST THOSE VARIABLES DEFINED
      DO 15 I = 70,82,3
C      IS ELEMENT DESCRIBED
      IF (NDEF(M(I)))15,10,10
C      LIST ITEM
   10 IDX = (I-67) / 3 * 5 + 36
      CALL PAOUT(NOD,5,5,M(IDX))
      CALL PLOUT(NOD)
   15 CONTINUE
C      RETURN TO PLAN
   20 CALL LRET
C      BASE IS IN ERROR
   25 CALL PAOUT(NOD,1,5,M(41))
   30 CALL PAOUT (NOD,7,M,M(2))
   32 CALL PLOUT(NOD)
      GO TO 20
C      RISE IS IN ERROR
   35 CALL PAOUT (NOD,1,5,M(46))
      GO TO 30
C      SLOPE IS IN ERROR
   40 CALL PAOUT(NOD,1,5,M(51)))
      GO TO 30
C      ANGLE IS IN ERROR
   45 CALL PAOUT (NOD,1,5,M(56))
      GO TO 30
C      BEVEL IS IN ERROR
   47 CALL PAOUT(NOD,1,5,M(61))
      GO TO 30
C      BEVEL ANGLE IS INVALID DEFINITION
   50 CALL PAOUT (NOD,1,5,M(56))
      CALL PAOUT (NOD,7,5,M(61))
      CALL PAOUT (NOD,13,2,M(2))
      CALL PAOUT (NOD,16,M(25),M(26))
      GO TO 32
      END
```

7.4.0 ALTERNATE SOLUTION

The module CSERR and command CSERR can be eliminated altogether by making use of the check-entry facility through an expanded version of the TRI SOL command. Spaces shown in the following example may need to be removed to keep the phrase from exceeding 450 characters.

ALT PHR:TRI SOL, LEV1,(70)BAS,(73)RIS, (76)SLO, (79)ANG,(82)BEV,(20)-:(BAS<0)?=+, -:(RIS<0)?=+, -:(SLO<0)?=+, -:(ANG<0)?=+, -:(BEV<0)?=+, -:(ANG>90)?=+, -:(ANG=-)| (BEV=-)?=-!=+, TEST, I(8)J1,N0, -*P'CON TRI SOL', $4:BAS(J)?$5!$6, $5N=N+1, $6J=J+1, :(J=16)?$7!$4, $7TES:(N=2)?$8!=-, $8;

ALT PHR:CON TRI SOL, (20)*FA'BASE NEGATIVE', *FA'RISE NEGATIVE', *FA'SLOPE NEGATIVE',*FA'ANGLE NEGATIVE', *FA'BEVEL NEGATIVE', *FA'ANGLE GREATER THAN 90', *FA'ANGLE AND BEVEL MAY NOT BOTH BE

```
DEFINED',  *FA'INVALID PARAMETER SPECIFICA-
TION' PRO'CALC';
```

8.0.0 APPENDIX A: 1130 PLAN SPECIFICATIONS

This appendix contains additional information about the specifications and use of the PLAN system on the IBM 1130 Data Processing System. Included is information of additional PLAN features that allow a user to make better use of features unique to the 1130. Note that use of these features may create code that is dependent upon running within the 1130 version of PLAN. Specific references to compatibility considerations are provided in Appendix J (18.0.0).

8.1.0 USER EXITS

User-exit routines for 1130 PLAN must have a COMMON statement defined carefully according to the formulas shown below:

If 8K PSCAN is used:

NFW = (CORE-1500)/2

If 16K PSCAN is used:

NFW = (CORE-2000)/2

Where:

NFW	is the total number of 32-bit words that must be specified in the COMMON statement.
CORE	is the machine size (8192, 16,384, or 32,768) specified by monitor (core parameter in monitor load) at the time the user module is core-imaged.

Index register 1 provides a pointer to a communication block that may be used by the user-exit program, provided that the program is written in the assembly language.

Displacement	Contents of Word(s) Addressed
0-23	These locations contain linkages to subroutines that may be useful in the user-exit routine. They are used by a linkage of the type: ` BSI X1 N` / ` DC ARG` — N is the displacement defined in this table. Each linkage is followed by additional information as shown below. ARG in the examples is the address of the parameter required for the appropriate subroutine.
0	These words provide linkage to the floating-point load (FLD) routine.
3	These words provide linkage to the floating-point store (FSTO) routine.
6	These words provide linkage to the floating-point add (FADD) routine.
9	These words provide linkage to the floating-point subtract (FSUB) routine.
12	These words provide linkage to the floating-point multiply (FMPY) routine.
15	These words provide linkage to the floating-point divide (FDIV) routine.
18	These words provide linkage to the floating-point to integer conversion routine (IFIX). The floating-point number is assumed to be in the floating-point accumulator. The result is placed in the accumulator. The DC following the BSI is replaced with a NOP for this linkage.
21	These words provide linkage to the integer to floating-point conversion routine (FLOAT). The integer is assumed to be in the accumulator. The result is placed in the floating-point accumulator. The DC during the BSI is replaced with a NOP for this linkage.

The subroutines described above are further defined in the manual IBM Subroutine Library (C26-5929).

Index register 3 must point at the transfer vector of the user-exit routine when a CALL IUSER is issued. The register will automatically be pointed to the PSCAN transfer vector (floating accumulator) following any linkage on index register 1, as outlined above.

8.2.0 COMMUNICATION ARRAY SPECIFICATIONS

COMMON has a maximum allowable size that is a function of the machine size and the maximum size PLAN module. The maximum size communication array for 1130 PLAN using the 8K version of PSCAN and PERRS is computed from the following formula:

```
    NWDS = (KORE -(8192-1020))/2
```

The commands "ADD PHRASE:", "ALTER PHRASE:", and "DELETE PHRASE:" will cause overlay of the communication array. The 16K versions of PSCAN and PERRS allow protection of a communication array as defined by the formula shown below, where KORE is the machine size of a 16K or 32K system.

```
    NWDS=(KORE-13056)/2-640
```

NWDS is the number of 32-bit words that may be contained in the communication array.

KORE is the number of 1130 machine words in the object machine. Simplified, the formula becomes:

```
    NWDS = KORE/2-6528
```

8.3.0 PROGRAMMING RESTRICTIONS

The following 1130 FORTRAN statements should not be used because of their detrimental effect on the execution of PLAN. Alternate facilities are listed for each option.

To avoid overriding the PLAN processor or endangering another user's job, the following 1130 FORTRAN statements should not be executed.

CALL EXIT	This subroutine call creates a premature return to monitor. A CALL LRET should be used instead to return control to PLAN. CALL LEX(1,0) will return control to PLAN and clear the pop-up list.
STOP	This statement has the same effect as CALL EXIT when processing is restarted.
CALL LINK	The PLAN loader subroutines (LRET, LEX, LIST) must replace this call to allow PLAN to remain in control.
DEFINE FILE, READ(a'b) WRITE(a'b)	The PLAN file routines (FIND-READ-WRITE or GDATA-RDATA-WDATA) provide for discretely addressable, execution-time definable files, and should be used instead of these statements. Any function disturbing the contents of working storage may not be intermixed with the execution of PLAN dynamic file routines. If these files reference only files defined in the FIXED AREA, they may coexist with PLAN file processing routines.

8.4.0 OVERLAY STRUCTURE

The CALL LOCAL provides one level of program overlay under 1130 PLAN.

The local program is core-imaged and stored by name in the 1130 program library the same as any other PLAN module. The program must be core-imaged to a fixed address that does not jeopardize the calling program. The following approach may be used to generate the LOCAL modules:

1. Write FORTRAN module that is to be called as a PLAN LOCAL where ddddd represents the decimal address of the beginning of the executable code. This address must allow for 30 words above the end of the program issuing the CALL LOCAL.

```
// JOB
// FOR
*LIST ALL
*ORIGIN ddddd
   COMMON L(625),LS(15),M(510)
C  TEXT OF PROGRAM FOLLOWS
   •
   •
   •
   RETURN
   END
// DUP
*DELETE                NAMEA
*STORE      WS   UA    NAMEA
```

2. Execute PLAN.

Note that the variable parameter list provided in System/360 OS/DOS PLAN is not supportable by 1130 PLAN.

Each module issuing a CALL LOCAL must allow for a 42-word (16-bit) block at the end of the load (transfer vector) for the saving of the necessary control information.

8.5.0 IOCS DEVICE PARAMETERS

Under 1130 PLAN, the IOCS subroutine and sequential file subroutines parameters are the standard device codes shown below:

PLINP INPUT	PLOUT LIST	MEANING
0		Current PLAN input device
	100	Current PLAN output device
1		1442 Card Reader
	101	1132 Printer
2		2501 Card Reader
	102	1403 Printer
3		1131 Console Keyboard
	103	Console Printer
	104	1442 Punch

A value of 0 for NOD for the sequential I/O routine specifies the PLAN input device and 100 specifies the current output device.

IOCS is a level 0 command on 1130 PLAN that allows the PLAN input and output devices to be altered.

The general format of the command is:

```
IOCS, INPUT n, LIST m;
```

a. INPUT. This parameter must specify the input unit that is to be used for input of following PLAN commands. Valid arguments are 2501, 1442, and 1131.

b. LIST. This parameter must specify the output unit that is to be used for output of following PLAN diagnostics. Valid arguments are 1132, 1403, 1442, or 1131.

CARD is a blank-level command on 1130 PLAN that allows changing input to either card reader and/or output to either line printer.

The general format of the command is:

```
CARD, INPUT n, LIST m;
```

a. INPUT. This parameter must specify the card reader from which the next PLAN input is to be read. Valid arguments are 2501 or 1442.

b. LIST. This parameter must specify the printer on which the next PLAN diagnostic is to be printed. Valid arguments are 1403 or 1132.

TYPE is a blank-level command on 1130 PLAN that sets the console typewriter/printer as the input/output device from/to which the next PLAN input/output is to be read/written. (See "PLAN Standard Phrases", 4.5.0, for a detailed description of this command.)

8.6.0 DYNAMIC FILE SUPPORT

The positions of the NDR parameter to the left of the decimal units position are treated as a communication array pointer. The contents (16-bit) of the indicated communication array are tested against the cartridge identification provided by the DCIP (disk pack initialization) routine at monitor generation time. If the cartridge is not found, an error is given.

The basic unit of allocation for 1130 PLAN files is four sectors. This provides storage for 628 32-bit words. Specification in the CALL FIND of a NALLO parameter may override the four-sector allocation.

The 1130 PLAN file routines allow pack changing as defined by the following specifications:

1. A cartridge change is not allowed on logical drive 0.
2. The user must assure that packs to be dismounted do not contain any required program or PLAN file.
3. The cartridge change may be effected by the following subroutine call:

```
CALL PLNUP(MPKID,NDR,MCODE,MLDR)
```

MPKID	This parameter contains the 16-bit cartridge identification mask, in the range 0001-7FFF (hexadecimal), that is used to check the validity of the mounted cartridge.
NDR	This parameter defines the logical drive number in the range of 0-4. This field may also contain a value of 100 to specify that the pack is to be mounted on the first available drive. The resulting drive code is returned in the MLDR parameter.
MCODE	This parameter is the return code as follows: 1. Pack successfully mounted 2. Invalid pack ID or drive code 3. Current pack cannot be dismounted 4. Requested pack is already mounted on another logical drive. The drive number is returned in the MLDR parameter. 5. Operator decided not to mount the pack (see "Pack Changing Instructions" in the 1130 Operations Manual). 6. Invalid sequence of logical drives, for example, logical drive 3 is requested, but logical drive 1 and 2 are not mounted. 7. No available logical drive. If

NDR equals 100, all available drives are already mounted.

MLDR This parameter contains the drive code if the pack is mounted successfully.

Note that any monitor (SUP, DUP, FOR, or ASM) function that uses working storage on a pack on which PLAN files reside may result in destruction of those PLAN files.

8.7.0 PERMANENT FILE SUPPORT

The file that is processed by GDATA-RDATA-WDATA is initially established outside of 1130 PLAN. One means of establishing a file is illustrated in the example shown below. Additional information may be found in IBM 1130 Disk Monitor System, Version 2, System Introduction (C26-3709).

```
// DUP
*STOREDATA  WS  UA  FILE  nnnn
```

FILE specifies the identification of the file by which it is stored (and subsequently identified in the GDATA call), and nnnn defines the number of sectors that are to be assigned to the file. This allocation is never changed by PLAN.

8.8.0 EXTENDED PRECISION SUPPORT

The following set of subroutines provides support for extended precision floating-point in the form of conversion subroutines. PLAN floating-point functions support only standard floating-point. Care must be exercised if the control card *EXTENDED PRECISION is used to adjust communication array references to adjust for the 48-bit word size. It is recommended that *ONE WORD INTEGERS be used and that COMMON be defined as integer arrays with twice the normal word count.

Special care must be taken when using this support to assure that the PLAN loader COMMON specification equals 625 32-bit words and that the Switch Word COMMON specifications equal 15 32-bit words.

CALL PEXTP (FROM, TO, COUNT) converts a standard precision floating-point array starting at FROM, to an extended precision floating-point array, starting at TO. The number of values converted is specified by COUNT. The result is in the following form:

```
|      |              | |        |        |
|UNUSED|CHARACTERISTIC|S|MANTISSA|MANTISSA|
└──────┴──────────────┴─┴────────┴────────┘
 0    7 8           15 16 17    31 0     15
```

S=Sign of Mantissa

CALL PENRM (FROM,TO,COUNT) converts an extended precision floating-point array starting at FROM, to a standard precision floating-point array, starting at TO. The number of values converted is specified by COUNT. The result is in the following form:

```
|   |        |        |              |
| S |MANTISSA|MANTISSA|CHARACTERISTIC|
└───┴────────┴────────┴──────────────┘
 0  1      15 16    23 24           31
```

S=Sign of Mantissa

CALL PEPCK (FROM,TO,COUNT) packs an extended precision floating-point array as stored by the PLAN user exit in location FROM into array TO in the standard 1130 FORTRAN array format. COUNT is the number of variables to convert.

CALL PEUPK (FROM,TO,COUNT) expands a standard 1130 extended precision array (occupying three words per variable) at location FROM into a PLAN user exit extended precision array (occupying four words per variable) at location TO. COUNT is the number of words to convert.

CALL PIPCK (FROM,TO,COUNT) packs integer data in array FROM stored in PLAN input form to array TO. Array TO is in one-word integer form. COUNT is the number of integers transferred. Formats of FROM and TO are given below:

```
FROM    |       |      |       |      |
        |INTEGER|UNUSED|INTEGER|UNUSED|
        └───────┴──────┴───────┴──────┘
         0    15 16  31 0    15 16  31


TO      |       |       |       |       |
        |INTEGER|INTEGER|INTEGER|INTEGER|
        └───────┴───────┴───────┴───────┘
         0    15 16   31 0    15 16   31
```

CALL PIUPK(FROM,TO,COUNT) expands a standard one-word integer array to standard PLAN input format. The formats of the FROM and TO arrays are exchanged as shown in the above example.

PLAN USER EXIT 1 (EXIT1) is utilized through a routine that will allow collection and conversion of input data in extended precision.

The resultant numbers are placed in the communication array on even-word boundaries, starting at the location specified in the phrase definition. The extended precision number is stored in two adjacent normal precision 32-bit word locations. The format of the result is:

```
|       |       | |         |         |       |
|UNUSED|   C   | S|MANTISSA|MANTISSA|UNUSED|
└──────┴───────┴─┴─────────┴─────────┴──────┘
 0    7 8    15 16 17     31 0      15 16   31
```

C=Characteristic of Number
S=Sign of Mantissa

8.9.0 EXPANDED LOADER FUNCTIONS

Two subroutines are provided to allow temporary use of the loader overlay area for temporary data storage.

CALL LSAV saves the current status of the loader area, including the switch area (540 FORTRAN words), exclusive of the bootstrap area. CALL LSAV must be issued before data is stored in the loader area. Control returns to the next statement after CALL LSAV.

CALL LRLD must be used to restore the saved status of the loader before any other call to a PLAN loader function, after a CALL LSAV has been issued. Control returns to the next instruction after CALL LRLD. PLAN loader functions include CALL ERRET, CALL ERRAT, CALL ERROR, CALL ERREX, CALL GDATA, CALL RDATA, CALL WDATA, CALL INPUT, CALL PHIN, CALL PUSH, CALL PHOUT, CALL GDAT1, CALL WDAT1, CALL RDAT1, and all routines prefixed with an L.

8.10.0 SYSTEM FILE DEFINITIONS

Two optional files are used by 1130 PLAN. They are PDATA and PCHPT. This section defines the method of computing the required file size in sectors.

File:	PDATA
Required if:	Level 2, level 3, or level 4 phrases are used and Switch Word 10 is nonzero (data is to be managed by levels) or CALL PSORT or CALL PMERG functions are used.
Size Rqd:	(M+159)/160*3, where M is the size of the managed array as defined by Switch Word 10.
	In addition, (L+159)/160, where L is the size of the communication array (words must be provided) if PSORT/ PMERG are used.

File:	PCHPT
Required if:	CALL LCHEX is used, if abnormally large numbers of errors result in an overflow of the error stack, if the IMMEDIATE option for error processing is used, if a user module is used to process errors or if PSORT and PMERG are used. Although PLAN will operate in many environments without a checkpoint file, it is recommended that the system contain room for at least one checkpoint level.
Size Rqd:	DB*16/320, where DB is the sum total of disk bytes used by the programs (those to be checkpointed simultaneously) when they were stored in the program library.

9.0.0 APPENDIX B: SYSTEM/360 DOS PLAN SPECIFICATIONS

This appendix contains additional information about the specifications and use of the PLAN system on the IBM System/360 under the Disk Operating System. Included is information of PLAN features that allow a user to make better use of features unique to the Disk Operating System. Note that use of these features may create code that is dependent upon running within the System/360 DOS version of PLAN. Specific references to compatibility considerations are provided in Appendix J (18.0.0).

9.1.0 DOS/360 PLAN SYSTEM

The DOS PLAN system is initiated as a DOS/360 job step. Once in execution it assumes the responsibility of loading other problem program load modules within the partition. PLAN must be run in the background partition.

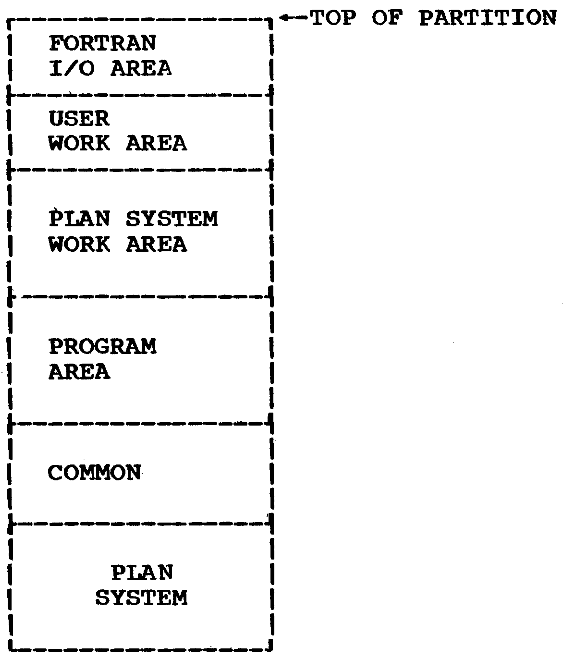


Figure 14. DOS PLAN storage utilization

The PLAN system is a part of blank COMMON. It is 2560 bytes long (640 32-bit words). Every module loaded by PLAN must have a blank COMMON control section and must protect this area with a dummy array at the beginning of COMMON.

The <u>program area</u> starts at the top of blank COMMON and extends upward to the PLAN <u>system work area</u>. All modules loaded by PLAN must be link-edited so that they fall entirely in this area.

The PLAN <u>system work area</u> contains PLAN tables and I/O buffers required to perform all PLAN I/O operations. The size of this area is variable, ranging upward from a minimum of approximately 3500 bytes.

The <u>user work area</u> is an array declared at PLAN initialization time. Its address, if present, is passed to every module loaded by PLAN. The default length of this area is zero.

The <u>FORTRAN I/O area</u> is an array declared at PLAN initialization time. This array is used by the FORTRAN I/O package for I/O areas, etc. This area must contain at least 512 bytes if FORTRAN I/O is used. The default length of this area is zero.

9.2.0 COMMON CONTROL

COMMON is managed and referenced in DOS PLAN according to the following procedures.

1. The PLAN subroutines reference COMMON through a blank COMMON control section of 640 words.

2. The length of COMMON may be altered whenever a mainline (NON-LOCAL) module is loaded. PLAN Switch Word 9 controls the minimum length of COMMON. All modules loaded by PLAN must have a COMMON control section at least as long as the value specified in Switch Word 9.

9.3.0 PROGRAM AREA CONTROL

All modules loaded by PLAN must meet the following requirements:

1. A COMMON control section at least as long as the value currently in Switch Word 9.

2. The module must be link-edited so that it falls entirely within the program area.

3. Modules to be loaded as LOCALs must be link-edited so they do not overlay the calling module.

9.4.0 USER-EXIT PROGRAMMING

The PSCAN user-exit program must be written to expect the standard System/360 FORTRAN subroutine linkage conventions.

The user-exit program must be link-edited so that its origin is above the end of the PLAN module DFJPSCAN.

9.5.0 COMMUNICATION ARRAY SPECIFICATION

The size of COMMON protected from overlay by PLAN modules is 640 32-bit words plus the amount added to the origin of PLAN system modules at the time they are link-edited (See "Generating a Tailored PLAN System" in the DOS PLAN Operations Manual). Data will not be stored by PSCAN into the communication array beyond the smaller of (1) the origin of PSCAN or (2) the value contained in Switch Word 9. PSCAN will give an error diagnostic and abort if an attempt is made to store beyond these limits.

9.6.0 PROGRAMMING RESTRICTIONS

The following System/360 FORTRAN statements should not be used because of their detrimental effect on the execution of PLAN. Alternate facilities are listed for each option.

CALL EXIT	This subroutine call creates a premature return to the DOS supervisor. A CALL LRET should be used instead to return control to PLAN. CALL LIST(1,0) will return control to PLAN and clear the pop-up list.
STOP	This statement has the same effect as CALL EXIT when processing is restarted.
CALL DUMP	This statement creates a premature end to the PLAN execution. Therefore, the CALL PDUMP, followed by a CALL LRET, should be used.

Any PLAN module that issues a CALL LNRET must exit by a PLAN loader call (may not use a RETURN statement).

9.7.0 CORE MANAGEMENT

The PLAN loader provides management of core assignment to allow coexistence of independently written, functionally dependent pieces of code.

The user is provided with special arguments that, when encountered in the pop-up list, indicate the limits of the functionally dependent modules. The left parenthesis indicates the start of a string of module names for which the user desires coexistent residence. The right parenthesis indicates the end of the string. Figure 15 represents the pop-up list containing a list of programs. Programs M0716 through M0725 are to be grouped in memory concurrently.

```
-----┐      ┌----
     |M0712|
     |M0756|
     | (   |
     |M0716|
     |M0796|
     |M0732|
     |M0725|
     | )   |
     |M0749|
     | 0   |
     └-----┘
```

Figure 15. Loader pop-up list

The systems programmer in determining the scheduling control, that is, which modules may coexist within the partition, must recognize and/or account for the following conditions:

1. If more modules are grouped (bounded in the pop-up list with parentheses) than can coexist, those modules that will not fit are not loaded concurrently.
2. If space can be found, all parenthetically grouped modules are loaded into the partition. Entry is made to the first program named after the left parenthesis.
3. Loading of a module results only if the module does not already exist in memory.
4. If the left/right parenthesis is encountered when entering data into the pop-up list without a corresponding right/left parenthesis, the unmatched parenthesis is ignored. Therefore, parenthetically grouped programs must be added to the pop-up list with a single loader subroutine call.
5. If the left or right parenthesis is to be inserted in the pop-up list, it must be left-justified in two FORTRAN words.
6. Program lists, verb lists, and check-entry program lists include the parenthetical groupings in literal form. Example:

```
...,PROGRAMS 'M0713, (M0726, M0733,
M0792), M0796',...
```

7. The combination of the parenthetical program grouping and the use of command input of program lists gives the user the power to add segments (modules) to his root structure at execution time.

8. If all programs indicated in the coexistent grouping cannot be loaded because of conflicting core residence requirements (programs should be link-edited so they do not overlay each other) the right parenthesis is floated forward in the list to include those programs for which coexistent loading was accomplished.

   The original right parenthesis is deleted and a right parenthesis is regenerated in the pop-up list at a position that indicates the last program which was successfully loaded.

9. A negative call to the program linkage routines (value of N is negative) is required to interrogate the pop-up list for successful loading of the coexistent programs.

10. Parenthetical grouping is acceptable but ignored on the 1130 version of PLAN.

11. All program lists to be inserted into or to be extracted from the pop-up list must begin on a full-word boundary.

9.8.0 RETURN LINKAGE

The FORTRAN RETURN statement functions exactly like the CALL LRET PLAN loader call. Register 14 is used to cause a return from the logic module to the PLAN loader. PLAN modules that contain CALL LNRET may not exit via RETURN. FORTRAN subroutines which modify variables passed to them as arguments must use the FORTRAN RETURN statement.

9.9.0 OVERLAY STRUCTURE

The System/360 DOS PLAN System provides a local overlay structure that provides the mechanism for common usage of multiple-purpose control sections. This type of processing is typified by an application in which the mainline serves only to provide linkage to logic segments that perform specific functions, and provides the basic hardware routines.

The following logic module is considered appropriate for an application of the type listed above. It is assumed in the example that a command would initially load the example module and define the local tasks to be completed by entries in the pop-up list.

```
      EXTERNAL ARG1, ARG2,...
    1 CALL LOCAL(0,0,ARG1,ARG2,...,ARGN)
      GO TO 1
      END
```

The local module would then be written in the following form:

```
SUBROUTINE NAME
CALL ARG1(X,Y,Z)
CALL ARG2(P,Q,R)
    .
    .
    .
RETURN
```

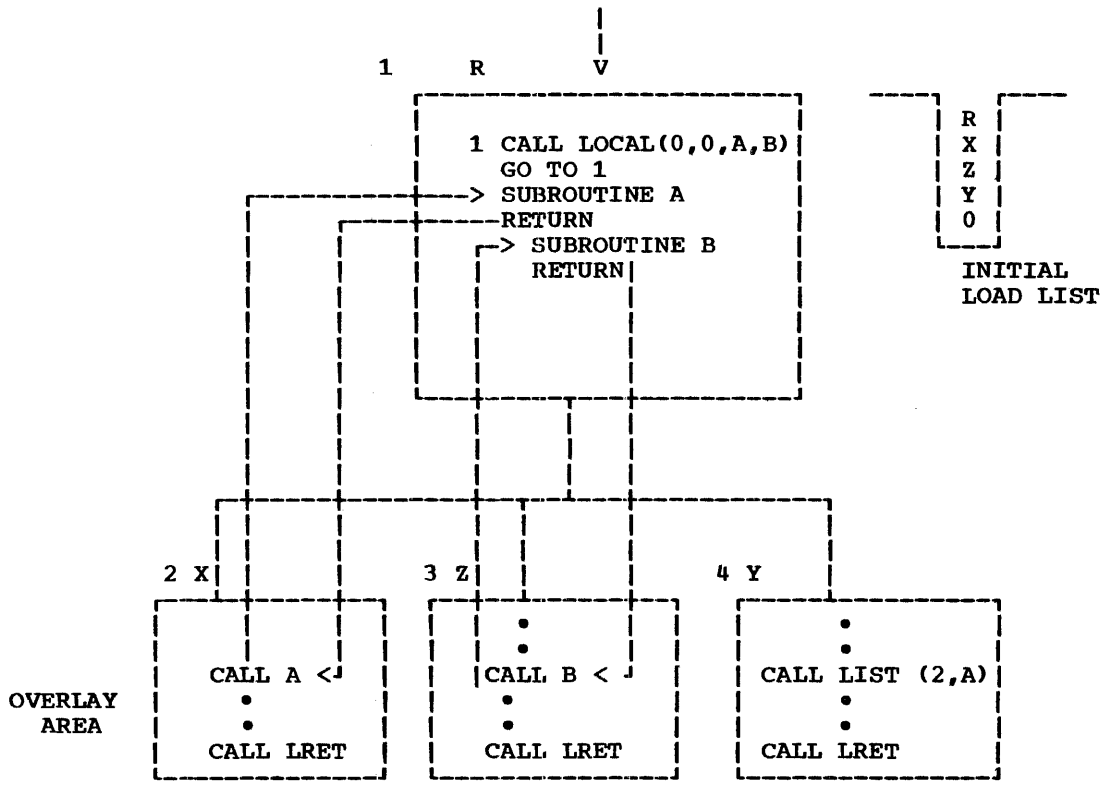


Figure 16. DOS overlay structure

Return from the LOCAL immediately loads the next module indicated in the pop-up list until the list is found to be empty. At that time, control is given to PSCAN for processing a new command. The logic module shown in the above example would incorporate all multiple-use subroutines required by the local modules.

The use of CALL LOCAL in a source program suggests detailed knowledge of an installation's core storage boundaries. There must be room enough for all load modules that are implied by any sequence of CALL LOCALs without intervening RETURNS. Since core use is an installation variable, it is not good practice to use CALL LOCAL in general purpose modules. This call is designed for root modules containing shared subroutines to use in invoking a hierarchical overlay scheme.

A program module that has issued CALL LOCALs and has not regained control may not be the object of another CALL LOCAL.

9.10.0 PLAN SYSTEM CHECKPOINT

The following regulations govern execution and control of the checkpoint facility within the OS version of PLAN (CALL LCHEX):

1. Checkpoints can be reloaded only within the limits of the phrase from which they were written. This means that any checkpoint that has not been reloaded when the end of the phrase is encountered -- that is, when the pop-up loader is found to be empty -- is destroyed. No warning message is issued.

2. If the checkpoint return (*) is encountered while in local mode, the local processing is terminated and the checkpoint is reloaded.

3. Any input/output error while reading or writing the checkpoint data set results in a phrase abort, and PLAN level error recovery is initiated. This action is also true when insufficient space is available in the checkpoint data set.

4. The DOS checkpoint facility has a unique feature that enables the PLAN subroutine LCHEX to function in a manner similar to the LOCAL facility. This is accomplished by not actually writing a checkpoint when requested but instead marking all modules in the program area as ready to be checkpointed. Any time a program that is marked as such is about to be overlayed by the loading of another program, the

physical write to the checkpoint file takes place. This allows the user to take advantage of additional core without reprogramming modules that use LCHEX by relink-editing the modules called by LCHEX.

5. There is no logical restriction on the number or level of checkpoints that a user may execute. A physical limit based on the size of the checkpoint data set may produce a real limit or error condition as outlined in 3 above.

6. Checkpoint restarts are executed in a reverse order from which they are written, that is, last in-first out.

7. After a checkpoint is taken, the status of all data sets except system data sets (those data sets processed by CALL PLINP, CALL PLOUT, CALL GDATA, and CALL FIND) must not be altered until the checkpoint is restarted. This is a user responsibility and no check is made by PLAN to prevent such an alteration. If a data set status is altered while a checkpoint is in effect, the results are unpredictable.

8. COMMON is not protected between the time that a checkpoint is taken and the restart is loaded. It is the user's responsibility to save and reload those parts of COMMON that might be destroyed and that must be present for continued execution of the checkpointed module.

9. Floating-point registers are not restored when a checkpoint is restarted.

9.11.0 DYNAMIC FILE SUPPORT

The NALLO parameter provided with CALL FIND is used to optimize space allocation. The basic unit of allocation for an DOS PLAN file is 1350 32-bit words.

The positions of the NDR parameter other than the units position are not interrogated by DOS PLAN. Each logical file can contain up to 147 discontiguous allocations. Thus, if normal allocation is allowed as the file is written, the maximum file size is restricted to 220,500 32-bit words. If the NALLO parameter of the CALL FIND subroutine is utilized, the maximum file size is 49,150,350 32-bit words.

Each logical drive may contain a maximum of 149 discontiguous free areas. This means that in cases of <u>extreme</u> discontiguous allocation a file may be destroyed.

9.12.0 PERMANENT FILE SUPPORT

The DOS version of PLAN provides support for files established outside of PLAN with the following characteristics:

1. Files are limited to one extent.
2. Files must be organized as a sequential or direct access file.
3. Physical records are fixed length.
4. Track overflow feature may not be used.
5. There may be no truncated records.
6. There may be no keys.
7. There may be no control characters.
8. The extent must be on cylinder boundaries.
9. The entire extent must contain formatted records.

The file is initially established by a user-written routine using an access method. The RDATA/WDATA subroutines read the file to establish the format (record size) for the file.

9.13.0 IOCS DEVICE PARAMETERS

Under System/360 DOS PLAN, INPUT and LIST correspond to sequential devices defined at PLAN initialization time. See the <u>DOS Problem Language Analyzer (PLAN) Operations Manual</u> (H20-0597) for additional information on the definition of PLAN data sets.

9.14.0 SEQUENTIAL FILE SUPPORT

The following steps outline the manner in which certain special conditions are handled on the DOS/360 version of the sequential I/O subroutines (PLINP/PLOUT/PEOF/PCCTL).

Two subroutines are provided under DOS PLAN that allow specification of page length and status switching (CLOSE) for PLINP/PLOUT data sets.

CALL PPAGL(NOD,N) is a subroutine used to specify the number of lines to be used as the page length for those data sets containing printed output.

A call to PPAGL sets the current line count to the page length specified. It also forces the next carriage control operation to be a skip to 1 unless overridden by an intervening call to PCCTL. If N is 0, a default of 60 is used. The maximum value of N is 32,767.

CALL PENDF(NOD) is a subroutine that may be used to close a sequential data set. If a data set is in output status, an EOF is written after the last record. Both PLINP and PLOUT data sets are repositioned to the beginning of the data set.

1. Maximum record size for any input/output record is 32,760 characters.

2. Records may be blocked within the physical limits of the specified device.

3. A PLINP/PLOUT call to an invalid device is ignored.

4. In order to effect carriage control, that is, for PCCTL to be functional, the CCTRL option must be specified for the data set at PLAN initialization time (see "PLAN System Initialization" in the DOS PLAN Operations Manual).

5. The following items are specifications for the PEOF routine:

   a. Logical EOF is set when:
      (1) A "UREND" is read by CALL PLINP. The logical EOF will be reset by the next CALL PLINP to the data set.
      (2) The line count is zero for output data sets (CALL PLOUT) using the CCTRL option.

   b. Physical EOF is set when:
      (1) PHYSICAL EOF is read by a CALL PLINP.
      (2) A CALL PLINP is issued to a device not capable of input.
      (3) A CALL PLOUT is issued to a device not capable of output.
   c. A CALL PLOUT is issued to a data set in input status (a CALL PLINP had previously been issued).
   d. A CALL PLINP is issued to a data set in output status (a CALL PLOUT had previously been issued).

6. The following specifications pertain to the carriage tape simulation functions on an output device (CALL PCCTL):

   a. The maximum page length is 32,767 lines.
   b. Default page length is 60 lines.
   c. If the CCTRL option is specified, a line count is maintained and an automatic eject (skip to carriage channel 1) is set when the line count reaches zero.
   d. Maintenance of the line count is suspended when a CALL PCCTL is issued for a skip to channels 2-12.
   e. Maintenance of the line count is resumed when a CALL PCCTL is issued for a skip to channel 1.

A PLAN utility program, DFJPLENG, allows the user to set the page length to be used on an output file that is to contain data to be printed. This utility must be invoked by the standard PLAN command.

```
SET PAGE LENGTH, NOD xxx, PGL yyyyy;
```

where xxx is a number up to three digits equivalent to the NOD argument for the subroutines PLINP and PLOUT, and yyyyy is a number up to five digits to be used as the page length for the specified NOD.

9.15.0 PERMANENT FILE SORT/MERGE

CALL GSORT(ID) and CALL GMERG(ID,JD,KD) provide the identical functions for PERMANENT files as CALL PSORT and CALL PMERG do for DYNAMIC files.

10.0.0 APPENDIX C: SYSTEM/360 OS PLAN SPECIFICATIONS

This appendix contains additional information about the specifications and use of the PLAN system on the IBM System/360 under the Operating System. Included is information of PLAN features that allow a user to make better use of features unique to the operating system. Note that use of these features may create code that is dependent upon running within the System/360 OS version of PLAN. Specific references to compatibility considerations are provided in Appendix J (18.0.0).

10.1.0 OS/360 PLAN SYSTEM (LOADER)

The PLAN system is initiated by an OS/360 job step. Once in execution it assumes the responsibility of loading other problem program load modules within the partition or region.

Figure 17 illustrates the PLAN system use of main storage.

```
┌────────────────┐<--TOP OF PARTITION
|  PLAN SYSTEM   |
|  WORK AREA     |
├────────────────┤
|                |
|  MANAGED       |
|  FREE STORAGE  |
├────────────────┤
|                |
|  NONMANAGED    |
|  FREE STORAGE  |
|                |
├────────────────┤
|                |
|  PROGRAM       |
|  AREA          |
|                |
├────────────────┤
|                |
|                |
|     COMMON     |
|                |
|                |
|- - - - - - - - |
|                |
|      PLAN      |
|     SYSTEM     |
|                |
└────────────────┘
```

Figure 17. OS PLAN storage utilization

The PLAN system is a part of blank COMMON. It is 640 32-bit words long. Every load module that contains a blank COMMON control section must protect this area with a dummy array at the beginning of COMMON.

The total PROGRAM/COMMON area is under control of the PLAN system. Within this area, load modules are located as high as possible. The size of COMMON is variable. It begins at the bottom of the partition and extends towards the programs loaded at that time. The default length of the PROGRAM/COMMON area is 66 per cent of the partition/region size.

The Managed and Nonmanaged Free Storage areas are used to honor GETMAIN requests from problem programs. By default the size of the Nonmanaged Free Storage area is zero.

The PLAN system work area contains PLAN tables and I/O buffers required to perform all PLAN I/O operations. This area ranges upward from 3K bytes, depending upon: (a) use of the RAM and LINKPAC areas for reentrant modules and access methods, and (b) the number of optional data sets used by the PLAN job.

The length of the PROGRAM/COMMON area and the nonmanaged free storage area may be varied by the user at execution time through EXEC card parameters.

10.2.0 COMMON CONTROL

COMMON is managed and referenced in System/360 OS PLAN according to the following procedures:

1. The PLAN loader subroutines reference COMMON through a "BLANK COMMON" control section of 2560 bytes.

2. The PLAN loader, when loading modules, deletes the "Blank Common" control section from the module and modifies "Blank Common" references to point to PLAN COMMON.

3. The length of COMMON may be altered whenever a new load module is brought into core. It will be as long as required by any resident load module and never shorter than the length specified as a data variable in the loader Switch Word 9.

4. For those languages that cannot generate a BLANK COMMON CSECT, a virtual type ADCON referencing the name "PLANB-COM" will be resolved to point to PLAN COMMON. The load module containing

these references must not contain an actual CSECT named PLANBCOM.

10.3.0 PROGRAM AREA CONTROL

One or more load modules that are brought into core by a single loader entry form one program segment.

The PLAN system manages the program area by segment level. When PLAN is requested to load a module not in core, all segments in memory, assigned a segment level greater than the segment level of the module issuing the loader call, are released. This allows overlay processing but does not require overlay definition and link editing.

10.4.0 OS FREE STORAGE CONTROL

The PLAN system maintains several pointers concerned with the MANAGED FREE STORAGE area. Whenever a program segment is released, the system uses these pointers to perform the following maintenance:

1. DELETE modules that the segment loaded via the LOAD macro.
2. Close data sets that were left open by the segment.
3. Use the FREEMAIN macro to release all core obtained by the segment's use of the GETMAIN macro.

The user must be aware of the implications of the above maintenance procedures. Programs that reside in lower-level (higher-segments) that are called as LOCALs may issue the GETMAIN macro only for temporary use. Whenever a segment is released, all areas in MANAGED FREE STORAGE obtained by the GETMAIN macro are released. This includes both the segment and all modules or subroutines called as LOCALs by the segment.

If a NONMANAGED FREE STORAGE area is declared, it is the user's responsibility to maintain this area.

The use of managed or nonmanaged FREE STORAGE is controlled by a PLAN system indicator that may be dynamically controlled by the user through use of the PLAN utility modules DFJUMC and DFJUNC. Invoking module DFJUMC informs the PLAN system that managed free storage is to be used to honor GETMAIN macro requests; DFJUNC causes PLAN to effect the use of nonmanaged free storage to honor GETMAIN requests. Either of these routines may be invoked as a subroutine, by a CALL LOCAL, or as a PROGRAM entry associated with a phrase.

10.5.0 PROGRAM USE OF FREE STORAGE

Two subroutines are provided to allow the user to control the area of OS FREE STORAGE that is used to honor GETMAIN requests.

CALL DFJUMC sets the system status to indicate that the managed area of OS FREE STORAGE is used for GETMAINs.

CALL DFJUNC sets the system status so that the nonmanaged area of OS FREE STORAGE is used for GETMAINs.

10.6.0 PROGRAM AREA MANAGEMENT

The PLAN loader provides management of core assignment to allow coexistence of independently written, functionally dependent pieces of code.

The user is provided with special arguments that, when encountered in the pop-up list, indicate the limits of the functionally dependent modules. The left parenthesis indicates the start of a string of module names for which the user desires coexistent residence. The right parenthesis indicates the end of the string. Figure 18 represents the pop-up list containing a list of programs. Programs M0716 through M0725 are to be grouped in memory concurrently.

```
-----┐       ┌----
     |M0712|
     |M0756|
     | (   |
     |M0716|
     |M0796|
     |M0732|
     |M0725|
     | )   |
     |M0749|
     | 0   |
     └-----┘
```

Figure 18. Loader pop-up list

The systems programmer in determining the scheduling control, that is, which modules may coexist within the partition, must recognize and/or account for the following conditions:

1. If more modules are grouped (bounded in the pop-up list with parentheses) than can coexist, those modules that will not fit are not loaded concurrently.

2. If space can be found, all parenthetically grouped modules are loaded into the partition with the entry to the program named following the left parenthesis.

3. Loading of a module results only if the module does not already exist in memory.

4. If the left/right parenthesis is encountered when entering data into the pop-up list without a corresponding right/left parenthesis, the unmatched parenthesis is ignored. Therefore, parenthetically grouped programs must be added to the pop-up list with a single loader subroutine call.

5. If the left or right parenthesis is to be inserted in the pop-up list, it must be left-justified in two 32-bit words.

6. Program lists, verb lists, and check-entry program lists include the parenthetical groupings in literal form. Example:

   ....,PROGRAMS 'M0713, (M0726, M0733, M0792), M0796',...

7. The combination of the parenthetical program grouping and the use of command input of program lists gives the user the ability to add segments (modules) to his root structure at execution time.

8. If all programs indicated in the coexistent grouping cannot be loaded because of insufficient partition size, the right parenthesis is floated forward in the pop-up list to include those programs for which coexistent loading was accomplished.

   The original right parenthesis is deleted and a right parenthesis is regenerated in the pop-up list at a position that indicates the last program which was successfully loaded.

9. A call with a negative value of N is required to interrogate the pop-up list for successful loading of the coexistent programs.

10. Parenthetical grouping is acceptable but ignored on the 1130 version of PLAN.

11. The left and right parentheses and all programs associated with the indicated coexistent grouping must be added to the pop-up list with a single call to the PLAN loader subroutines, or both parentheses must be included in a phrase-defined program list.

12. All program lists to be inserted into or to be extracted from the pop-up list must begin on a full-word boundary.

13. Use of XCTL is prohibited in PLAN modules. The use of LINK or ATTACHED is allowed. Any program that is "linked" to by a module loaded by the PLAN loader may use the XCTL, LINK, or ATTACH macros. The linked-to program may also be in overlay mode.

14. The "overlay structure" is not supported in PLAN modules, except as defined in 13.

15. Modules loaded by PLAN may not be in overlay or scatter mode or contain TESTRAN symbol cards.

16. Load modules must be marked as executable by the link editor.

Load modules may simply succeed one another serially in the program area, occupying a minimum amount of core; or they may reside in core together in a manner similar to that supported by the OS/360 overlay supervisor.

The principal feature of PLAN loading is that load modules sharing core do not have to be link-edited together. One or more load modules that are brought into core by a single loader entry form one segment.

If a CALL LOCAL is issued during execution of a program within a segment, the loader does not release the calling segment. It attempts to load the new segment into the program area. This process may continue through several levels, as long as core is available. Failure to load a module that must be immediately entered will generate a PLAN diagnostic and allow the next PLAN statement to be executed. Failure to load a module that does not have to be entered immediately causes the left-hand parenthesis to be moved as indicated in step 8 above and execution continues.

CALL LOCAL does not immediately free storage; but the space used by inactive segments will be reclaimed if needed by the loader. Thus, modules that CALL one another in a loop can share core and execute with maximum efficiency, if they all fit within the available region.

The loader keeps track of RETURNs, CALLs outside of a load module, and CALL LOCALs to allow proper release and acquisition of space for load modules. The term "execution level" is defined as the number of CALL LOCALs that have not been canceled by RETURNs. The segment level is the "depth" of segments within the program area.

If names are parenthesized in the pop-up list, all load modules named inside the parentheses are treated as one segment.

The following narrative discusses the method used by OS PLAN for program area control and management.

Initial entry to the loader (Figure 19) finds the pop-up lists as shown. Parentheses call for three modules to form one segment. Program area now appears as shown in Figure 19(a). Module A is entered. The execution level is 1 (no CALL LOCALs yet). During its run, A issues a CALL LOCAL, transferring the names D and E in parentheses to the pop-up list as shown in Figure 20. Note that the name of A was removed from the pop-up list when it was loaded. Execution is at level 2. (A CALL LOCAL was issued.)

```
──┐     ┌──      ┌───────────┬──┐
  │ (A  │        │ MODULE A  │  │
  │     │        │ - - - - - │  │
  │  B  │        │ MODULE B  │  ├──> SEGMENT 1
  │     │        │ - - - - - │  │
  │  C) │        │ MODULE C  │  │
  │  O  │        ├───────────┼──┘
  └─────┘        │           │
                 │           │
                 │ - - - - - │
                 │ COMMON    │
                 └───────────┘
```

Figure 19. Initial entry to loader
Figure 19(a). Contents of program area

D is now at the top of the pop-up list but not in core. Since D was accessed by CALL LOCAL, it will be loaded as an additional segment in core. Parentheses define E in the same segment. Core now looks like Figure 19(a). Control passes to D at its entry point. D issues a CALL LOCAL without changing the pop-up list.

```
──┐      ┌──   ┌───────────┐
  │ (D   │     │           │
  │  E)  │     │ SEGMENT 1 │
  │  B   │     │           │
  │  C)  │     ├───────────┼──┐
  │  O   │     │    D      │  │
  └──────┘     │ - - - - - │  ├──> SEGMENT 2
               │    E      │  │
               ├───────────┼──┘
               │           │
               │ * * * * * │
               │           │
               │ COMMON    │
               │           │
               └───────────┘
```

Figure 20. Caller released from list
Figure 20(a). A bank load from call local

The load list (Figure 21) now has E at its top. E is in core. No new loading is required. The segment level remains 2, but execution is now at level 3. E issues a CALL LOCAL, adding C to the pop-up list. Control passes to C (in core already at segment level 1). Execution is at level 4. Figure 21(a).

Assume C RETURNS. It was called from E, which is reentered. Execution level = 3. E also RETURNS. It was called from D, which is reentered. Execution level = 2.

D now issues a new CALL LOCAL, adding F to the pop-up list (Figure 21(b)). F is not in core, so it becomes segment level 3. Core appears as in Figure 22. Control passes to F. Execution level = 3.

```
──┐     ┌──         ┌──────┐          ──┐    ┌──
  │ E)  │        ┌──│ A    │            │ F  │
  │     │        │  │ ---  │            │    │
  │ B   │        │  │  B   │            │ B  │
  │     │     1  │  │ ---  │            │    │
  │ C)  │        │  │    C │<──┐        │ C) │
  │ O   │        │  ├──────┤   │        │ O  │
  └─────┘        └─ │      │   │        └────┘
                 ┌──│ D    │   │
              2  │  │ ---  │   │
                 └─>│      ├───┘
                    │   E  │
                    └──────┘
```

Figure 21. No change to load list
Figure 21(a). Module called is already in core
Figure 21(b). CALL LOCAL transfers control

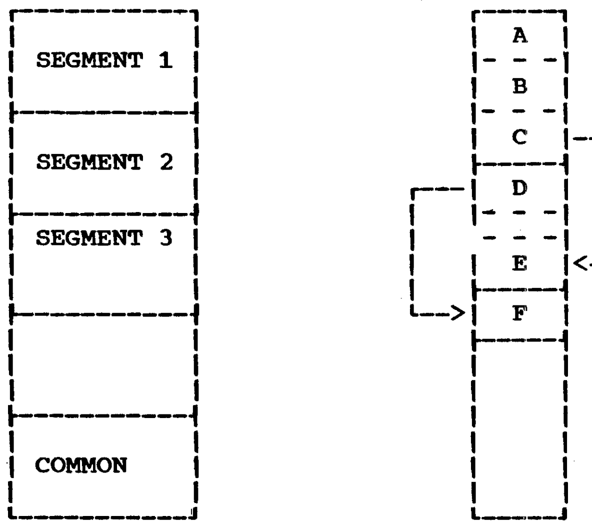


Figure 22. Control passes to a new segment
Figure 22(a). Contents of segments

Program F RETURNs. D is reentered. Execution level = 2. D RETURNS to A. Execution level = 1.

A issues CALL LOCAL, adding F to the pop-up list, F is in core, so it is entered. Execution level = 2.

F RETURNS. A receives control. Execution level = 1.

A issues a CALL LOCAL, adding H to the pop-up list. H is not in core. Control is in segment 1 and execution level is 1, so higher segments are released. Core now looks like Figure 23. Control goes to H. Execution level = 2.

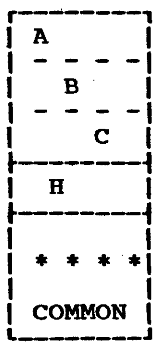


Figure 23. New segment replaces released segment

H issues a CALL LEX, adding E to the pop-up list. E is loaded and overlays H. (The reader should justify this statement.) E returns and A issues a CALL LOCAL without adding to the pop-up list. B will be entered in segment 1. B returns to A, which again issues a CALL LOCAL with no change to the pop-up list. C will be entered. After a return from C, if the loader is entered again without adding to the pop-up list, the list will be empty.

When loading from an empty pop-up, the PLAN module DFJPSCAN is loaded by default, to obtain the user's input. (A zero in the pop-up list appears to be the end of the list; so any program can return to the input reader by loading the name 0.)

10.7.0 RETURN LINKAGE

The FORTRAN RETURN statement functions exactly like the CALL LRET PLAN loader call. Register 14 is used to cause a return from the mainline (logic module) to the PLAN loader. PLAN modules that contain CALL LNRET or that are reentered at a primary entry may not exit via RETURN. FORTRAN subroutines which modify variables passed to them as arguments must use the FORTRAN RETURN statement.

CALL EXIT should be used to terminate a module to assure that buffers have been purged and data sets closed when non-PLAN I/O is incorporated within a module.

10.8.0 EXECUTION-TIME LINKAGE EDITING

Because the PLAN loader has full control of the region or partition, it can resolve references between load modules that were not link-edited together before execution.

While loading a module, all unresolved ADCONS pointing to entries in in-core segments will be resolved.

External subroutine references that are not resolved at link-edit time are effectively treated as CALL LOCALs by the PLAN system at execution time.

The restrictions on subroutines called in this manner are:

a. Standard linkage conventions must be used

b. Function subroutines may return answers only in FPR0 or GPR0

The two sets of coding shown below are equivalent and correct. The V-con for SUBRTN in set 2 may be unresolved following link-editing.

```
SET1
REAL*4 NAME(2)/'SUBRTN'/
  •
  •
  •
CALL LOCAL (2,NAME,ARG1,ARG2,ARG3)
END
```

```
SET2
  •
  •
  •
CALL SUBRTN (ARG1, ARG2, ARG3)
  •
  •
  •
END
```

Unresolved branch type (v) ADCONS that are to be resolved by the PLAN load at execution time are restricted. References to the ADCON must be direct. For example:

```
    L    15,=V(NAME)
    BALR 14,15
```

Offset referencing as shown below will not function correctly and will probably cause termination of the PLAN JOB step. In other words, IBCOM= cannot be called as a LOCAL.

```
    L   15,=V(NAME)
    BAL 14,N(0,15)
```

10.9.0 USE OF THE LINKPAC AND RAM AREAS

A PLAN utility program (DFJLLIST), that gives the PLAN system the capability of referencing the LINKPAC or RAM area, is provided. This utility must be invoked by the PLAN command:

```
    CREATE LOADER ENTRIES:  (NAME1,...);
```

where NAME1,... is a load module name that is to be loaded into the partition via the LOAD macro and be made available as entry points for the execution of any loader call. This allows programs in the LINKPAC or RAM areas to be objects of a CALL LOCAL. The names specified in the LIST must be in the JOBLIB PDS. To add this phrase to the dictionary, the following PLAN command must be executed:

```
    ADD PHRASE:  CREATE LOADER ENTRIES, PRO
    'DFJLLIST';
```

The maximum number of names in the list is 75. Use of this command destroys any entries defined by previous use of the command.

<u>Programs that reference blank COMMON may not be operands of this command</u>.

10.10.0 USE OF IN-CORE DIRECTORY

A PLAN utility program (DFJCRDIR) allows the user to build an in-core PDS directory of names of frequently loaded modules. This utility must be invoked by the PLAN command:

```
    CREATE CORE DIRECTORY:  (NAME1,...);
```

NAME1,... is a load module name that is placed in the in-core PDS directory to decrease load time for those modules. The names in the list must be entries in the PLANLIB PDS.

Use of this command will replace the previous directory. The maximum number of entries is 75 names.

This facility is added to the PLAN language dictionary (PFILE) by executing the following command:

```
    ADD PHRASE:  CREATE CORE DIRECTORY,
    PROGRAM 'DFJCRDIR';
```

10.11.0 PARAMETER PASSING

If the arguments in a parameter list are external names, the called program and calling program must be compiled by the same level FORTRAN compiler.

10.12.0 OVERLAY STRUCTURE

The System/360 OS PLAN System provides a local overlay structure that provides the mechanism for common usage of multiple purpose control sections. This type of processing is typified by an application in which the mainline serves only to provide linkage to logic segments that perform specific functions, and provides the basic hardware routines.

The following logic module is considered appropriate for an application of the type listed above. It is assumed in the example that a command would initially load the example module and define the first local task to be completed by entries in the load list.

```
      EXTERNAL ARG1,ARG2,...ARGN
    1 CALL LOCAL(0,0,ARG1,ARG2,...,ARGN)
      GO TO 1
      END
```

The local module would then be written in the following form:

```
    SUBROUTINE NAME (ARG1,ARG2,...ARGN)
    CALL ARG1(X,Y,Z)
    CALL ARG2(P,Q,R)
        .
        .
        .
    RETURN
```

Return from the LOCAL immediately loads the next module indicated in the pop-up loader until the loader is found to be empty. At that time control is given to PSCAN for processing a new command. The logic module shown in the above example would incorporate all multiple-use subroutines required by the local modules.

Note that the module issuing the CALL LOCAL or CALL LCHEX must complete non-PLAN file processing (EOF) before issuance of the call or must not allow the called module to use the file. It is not true of PLAN DYNAMIC, PERMANENT, and SEQUENTIAL file support.

The use of CALL LOCAL in a source program suggests detailed knowledge of an installation's core storage boundaries. There must be room enough for all load modules that are implied by any sequence of CALL LOCALs without intervening RETURNS. Since core use is an installation variable, it is not good practice to use CALL LOCAL in general purpose modules. This call is designed for root modules containing shared subroutines to use in invoking a hierarchical overlay scheme. An example is shown below.

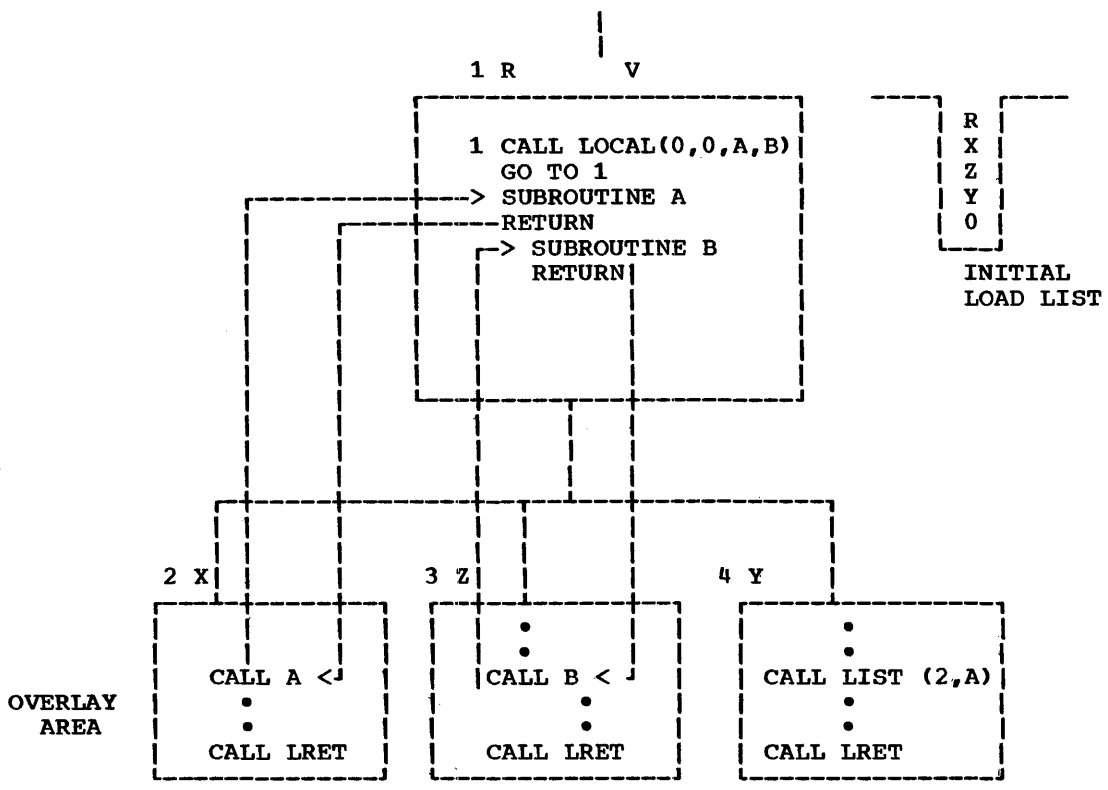


Figure 24. OS overlay structure

10.13.0 PLAN SYSTEM CHECKPOINT

The following regulations govern execution and control of the checkpoint facility within the OS version of PLAN (CALL LCHEX):

1. Checkpoints can be reloaded only within the limits of the phrase from which they were written. This means that any checkpoint that has not been reloaded when the end of the phrase is encountered -- that is, when the pop-up list is found to be empty -- is destroyed. No warning message is issued.

2. If the checkpoint return (*) is encountered while in local mode, the local processing is terminated and the checkpoint is reloaded.

3. Any input/output error while reading or writing the checkpoint data set results in a phrase abort and PLAN level error recovery is initiated. This action is

also true when insufficient space is available in the checkpoint data set.

4. The user may specify, in the DCB BLOCKSIZE parameter of the PLCHKPT DD card, the record size (in bytes) to be used when writing checkpoints. If no blocksize is specified, a blocksize of 512 is assumed.

5. There is no logical restriction on the number or level of checkpoints that a user may execute. A physical limit based on the size of the checkpoint data set may produce a real limit or error condition as outlined in 2 above.

6. Checkpoint restarts are executed in a reverse order from which they are written, that is, last in-first out.

7. After a checkpoint is taken, the status of all data sets, except system data sets (those data sets processed by CALL PLINP, CALL PLOUT, CALL GDATA, and CALL FIND), must not be altered until the checkpoint is restarted. This is a user responsibility and no check is made by PLAN to prevent such an alteration. If a data set status is altered while a checkpoint is in effect, the results are unpredictable.

8. COMMON is not protected between the time that a checkpoint is taken and the restart is loaded. It is the user responsibility to save and reload those parts of COMMON that might be destroyed and that must be present for continued execution of the checkpointed module.

9. Floating-point registers are not restored when a checkpoint is restarted.

10. The length of the PLAN PROGRAM/COMMON area must not be altered during the time a checkpoint is in effect.

10.14.0 USER-EXIT PROGRAMMING

The PSCAN user-exit program must be written to expect the standard System/360 FORTRAN subroutine linkage conventions.

10.15.0 COMMUNICATION ARRAY SPECIFICATION

The size of COMMON that is protected from overlay by PLAN system modules is the greater of (1) the size of PSCAN COMMON as defined at assembly time, or (2) the contents of Switch Word 9. PSCAN will give an error diagnostic and abort if an attempt is made to store values beyond these limits.

10.16.0 PERMANENT FILE SUPPORT

The OS version of PLAN provides support for files established outside of PLAN with the following characteristics:

1. File contains fixed length records.

2. File may be organized as a sequential or direct access file.

3. No secondary allocation is provided.

4. Track overflow feature may not be used.

5. No keys are allowed.

6. There may be no control characters.

7. The file may contain no truncated records.

The logical drive number (NDR) and the logical file number (ID(1)) must be equivalenced to the data set name. The DDNAME "PLFSynnn" will establish a name/number equivalence between PLFSynnn and NDR/ID(1), where y corresponds to NDR and may range from 0-7, and nnn corresponds to ID(1) and may range from 1-255.

10.17.0 DYNAMIC FILE SUPPORT (OS PLAN)

The NALLO parameter provided with CALL FIND is used to optimize space allocation. The basic unit of allocation for an OS PLAN file is 1350 words.

The positions of the NDR parameter other than the units position are not interrogated by OS PLAN. Each logical file can contain up to 147 discontiguous allocations. Thus, if normal allocation is allowed as the file is written, the maximum file size is restricted to 220,500 32-bit words. If the NALLO parameter of the CALL FIND subroutine is utilized, the maximum file size is 49,150,350 32-bit words.

Each logical drive may contain a maximum of 149 discontiguous free areas. This means that in cases of extreme discontiguous allocation a file may be destroyed.

10.18.0 IOCS DEVICE PARAMETERS

Under System/360 OS PLAN, INPUT and LIST correspond to units defined as DD names defined in the JCL for the PLAN job. The value specified for INPUT or LIST, corresponds to the device specified as the PLAN input device PLINPnnn in the job description deck. Unit nnn specified for LIST, corresponds to the device specified as the PLAN output device PLOUTnnn.

10.19.0 SEQUENTIAL FILE SUPPORT

The following steps outline the manner in which certain special conditions are handled on the OS/360 version of the SEQUENTIAL I/O subroutines (PLINP/PLOUT/PEOF/PCCTL).

Two subroutines are provided under OS PLAN that allow specification of page length and status switching (CLOSE) for PLINP/PLOUT data sets.

CALL PPAGL(NOD,N) is a subroutine used to specify the number of lines to be used as the page length for those data sets containing printed output. If N is 0, a default of 60 is used. The maximum value of N is 32,767.

A call to PPAGL sets the current line count to the page length specified. It also forces the next carriage control operation to be a skip to 1 unless overridden by an intervening call to PCCTL.

CALL PENDF(NOD) is a subroutine that may be used to close a sequential data set. If a data set is in output status, an EOF is written after the last record. Both PLINP and PLOUT data sets are repositioned to the beginning of this data set.

1. Maximum record size for any input/output record is 32,760 characters.

2. Records may be blocked within the limits of the facility for processing on the specified device. Truncated records are accepted if the character count is a multiple of the logical record length.

3. A PLINP/PLOUT call to an invalid device (missing DD card) is ignored.

4. In order to effect carriage control, that is, for PCCTL to be functional, the DCB RECFM parameter must be FA or FBA.

5. The DCB RECFM parameter must be F, FA, FB, or FBA.

6. If the device is a printer, the DCB RECFM parameter must be FA.

7. The following items define PCCTL functions:

   a. If the device is a reader, PCCTL will control stacker selection. DCB=(RECFM=F, BUFNO=1) must be used.
   b. If the device is a punch, RECFM must be FA for PCCTL to control stacker selection.
   c. If RECFM is FA or FBA, PCCTL will cause the correct ASA control character to be inserted as the first character of the record.

8. The following items are specifications for the PEOF routine:

   a. (1) Logical EOF is set when a "UREND" is read by CALL PLINP. The logical EOF will be reset by the next CALL PLINP to the data set.
      (2) The line count is zero for output data sets (CALL PLOUT) using RECFM FA or FBA.
   b. Physical EOF is set when:
      (1) EOF is read by a CALL PLINP.
      (2) A call PLINP is issued to a device not capable of input.
      (3) A CALL PLOUT is issued to a device not capable of output.
      (4) A CALL PLOUT is issued to a data set in input status (a CALL PLINP had previously been issued).
      (5) A CALL PLINP is issued to a data set in output status (a CALL PLOUT had previously been issued).

9. The following specifications pertain to the carriage tape simulation functions on an output device (CALL PCCTL):

   a. The maximum page length is 32,767 lines.
   b. Default page length is 60 lines.
   c. If RECFM is FA or FBA, a line count is maintained and an automatic eject (skip to carriage channel 1) is set when the line count reaches zero.
   d. Maintenance of the line count is suspended when a PCCTL CALL is issued for a skip to channels 2-12.
   e. Maintenance of the line count is resumed when a CALL PCCTL is issued for a skip to channel 1.

A PLAN utility program (DFJPLENG) allows the user to set the page length to be used on an output file that is to contain data to be printed. This utility must be invoked by the standard PLAN command.

```
SET PAGE LENGTH, NOD xxx, PGL yyyyy;
```

where xxx is a number up to three digits equivalent to the NOD argument for the subroutines PLINP and PLOUT and yyyyy is a number up to five digits to be used as the page length for the specified NOD.

10.20.0 PROGRAMMING RESTRICTIONS

The following System/360 FORTRAN statement should not be used because of its detrimental effect on the execution of PLAN. Alternate facilities are listed for each option. To avoid overriding the PLAN processor or endangering another user's job, the statement should not be executed.

CALL DUMP	This statement creates a premature end to the PLAN execution. Therefore, the CALL PDUMP, followed by a CALL LRET, should be used.

10.21.0 PERMANENT FILE SORT/MERGE

CALL GSORT(ID) and CALL GMERG(ID,JD,KD) provide the identical function for PERMANENT files as provided by CALL PSORT and CALL PMERG do for DYNAMIC files.

11.0.0 APPENDIX D: SYNTAX OF THE PLAN LANGUAGE

This appendix shows the optional and required elements of a PLAN statement. The first section shows the requirements for language definition and the second section shows the syntax of language use. Capitalized entries, left parenthesis, and right parenthesis are standard (nonvariable) items. Lowercase entries are replaced with variable information. Items in brackets are optional items. This material is presented in outline form to allow successively more detailed presentations of various items and options. Items enclosed in braces may be entered more than once.

Figure 25 is a graphic representation of the syntactical organization of the PLAN language.

11.1.0 LANGUAGE DEFINITION SYNTAX

```
I. ADD PHR: name[,definition];
   A. name is one to five words
   B. definition
      1. [LEVEL n,]
      2. [PROGRAM'program list',]
      3. [VERB['program list'],]
      4. [EXIT 'program list',]
      5. [data definition,]
```

The following abbreviations are used in the syntactical entries:

```
aex = arithmetic expression
aop = arithmetic operand
cap = communication array position
chk = check entry definition
dan = data name
dav = execution-defined data value (numer-
      ic, logical, literal)
I   = mode (I=integer)
lex = logical expression
lop = logical operand
nuv = numeric value
prl = program list
P±n = scale factor
saop= special arithmetic operand
sdv = standard data value (numeric,
      logical)
slv = standard literal value
Um  = user exit
←   = arithmetic operator
$n  = formula number
•   = logical operator
†   = relational operator
```

[(cap)sdv,]

[(cap)dan,]

[(cap)dan sdv,]

[I(cap)dan,]

[I(cap)sdv,]

[I(cap)dan sdv,]

[P±n(cap)sdv,]

[P±n(cap)dan,]

[P±n(cap)dan sdv,]

[I P±n(cap)sdv,]

[I P±n(cap)dan,]

[I P±n(cap)dan sdv,]

[Um(cap)dan,]

[Um(cap)dan sdv,]

[Um I(cap)dan,]

[Um I(cap)dan sdv,]

[Um P±n(cap)dan,]

[Um P±n(cap)dan sdv,]

[Um I P±n(cap),]

[Um I P±n(cap)dan sdv,]

[(aex)dan,]

[(aex)dan sdv,]

[I(aex)dan,]

[I(aex)dan sdv,]

[P±n(aex)dan,]

[P±n(aex)dan sdv,]

[I P±n(aex)dan,]

[I P±n(aex)dan sdv,]

[Um(aex)dan,]

[Um(aex)dan sdv,]

[Um I(aex)dan,]

[Um I(aex)dan sdv,]

[Um P±n(aex)dan,]

```
[Um P±n(aex)dan sdv,]

[Um I P±n(aex)dan,]

[Um I P±n(aex)dan sdv,]

[UM][I][P±n](cap)[dan][sdv]     [slv][{chk}]
[{lex}] [{aex}]
```

The following entries show valid syntactical entries for the phrase-defined formula area. Abbreviations used are defined above. Note that none of the preceding ADD PHRASE entries may follow any of the entries below.

```
[$n] [dan]=aex,

[$n] [dan]:  lex,

[$n] [dan]:  (lex) [?=aex],

[$n] [dan]:  (lex)[?:lex],

[$n] [dan]:  (lex) [?$n],

[$n] [dan]:  (lex)[?=aex!=aex],

[$n] [dan]:  (lex)[?:lex!:lex],

[$n] [dan]:  (lex)[?=aex!:lex],

[$n] [dan]:  (lex)[?:lex!=aex],

[$n] [dan]:  (lex)[?$n!=aex],

[$n] [dan]:  (lex)[?$n!:lex],

[$n] [dan]:  (lex)[?=aex!$m],

[$n] [dan]:  (lex)[?:lex!$m],

[$n]:$n,
```

The following entry is the valid form for arithmetic operands:

```
    dan {←dan} {←nuv}
```

The following entries are valid forms for special arithmetic operands:

```
    +
    -
    "LITERAL"
    'LITERAL'
    @LITERAL@
```

The following entries are the valid forms for arithmetic expressions (aex) in phrase entries. Braces define the acceptability of multiple entry of the enclosed items.

```
=aop {←aop}

=saop
```

The following entries are valid forms for logical operands (lop) in phrase entries (execution or ADD PHRASE):

```
dan{•dan}

(aex↑aex)

(dan="slv")

(dan=+)

(dan=-)
```

Logical expressions (lex) may be written in the following valid form:

```
:  lop{•lop}[?:lop{•lop}]
:  lop {•lop}[?:lop{•lop}!:lop{•lop},]
:  lop{•lop}[?=aex]
:  lop{•lop}[?:lop{•lop}!=aex]
:  lop{•lop}[?=aex!:lop{•lop}]
```

Note that there are other combinations of the elements shown above. In addition, parentheses may be used within logicals to show order of evaluation.

<u>11.2.0 LANGUAGE USE SYNTAX</u>

General format of execution-time PLAN statements is shown below:

```
I. Command, data section;
   , data section:
   Command;

   A. Command
      phrase
      {verb phrase}8 phrase

   B. data section
      [ ]dav[ ][,]
      [ ]dan[ ][,]
      [ ]dan[ ]dav[ ][,]
      [$n][ ]dan[ ]=[ ]aex[ ]
      [$n] [ ] dan[ ]:[ ]lex[ ]
      [$n] [ ]=[ ]aex[ ]
      [$n] [ ] :[ ]lex[ ]
      [ ] dan[ ] (aex) [ ] [,]
      [$n]   [ ]  dan[ ] (aex)[ ]dav [ ] [,]
      [$n]   [ ]  [$ ]  dan[ ](aex)   [ ]=[ ]
      aex[ ]
      [$n]   [ ] dan[ ] (aex)[ ]:[ ] lex [ ]
      [,]
```

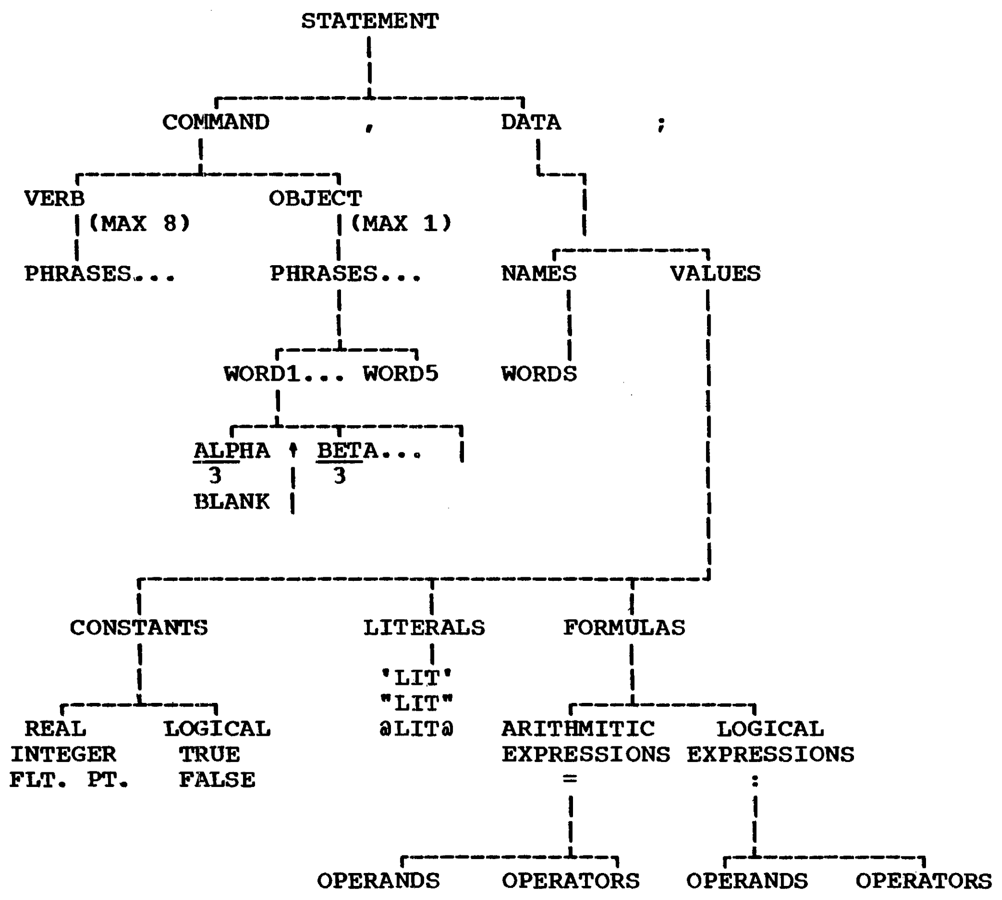


Figure 25. PLAN execution-time statement syntax

12.0.0 APPENDIX E: PLAN SYSTEM FILES LAYOUT

12.1.0 PFILE LAYOUT

The PLAN language definition file (PFILE) is generated and maintained by the PHRAS logic module and is utilized by PLAN (loader) and PSCAN for temporary system save areas. PFILE is required to be present before a PLAN execution is permitted.

PFILE is defined as a logical file containing a minimum of 14 (17 on the 1130) and a maximum of 268 (205 on the 1130) records. Records in PFILE are fixed in length at 512 bytes on System/360. On the 1130 each record is 320 (16-bit) words in length. The following table lists the contents of PFILE.

ITEM NAME	SIZE IN RECORDS	RECORD DISPLACEMENT	DESCRIPTION
PFLDRSV	5	0	Loader save and error stack area
PFSYMT4	1	5	Level 4 symbol table save area (128 words)
PFINPUTB	1	6	Card image residual save area (20 words)
PFSYMT3	1	7	Level 3 symbol table save area (128 words)
PFPWVTAB	1	8	Phrase-verb validity table (512 bytes)
PFSYMT2	1	9	Level 2 symbol table save area (128 words)
PFINPUTA	1	10	Current statement image save area (114 words)
PFSYMT1	1	11	Level 1 symbol table save area (128 words)
PFPAVTB	1	12	Phrase entry availability table (512 bytes)
PFPSASV	*		PSCAN save area
PFPETAB	1-255	13 or 16	Phrase entry table

*NOTE: This area is used in 1130 PLAN for saving portions of the PSCAN module. It consists of 3 sectors on the 1130 but is not present under OS or DOS PLAN.

The following section describes the functions of each of the areas listed in the above table of contents:

PFLDRSV This area is used for temporary storage of the 1130 PLAN loader. The use of this area is initiated by:

1. CALL LSAV

2. CALL PSORT

3. CALL PMERG

On all PLAN systems the area is used as a temporary stack area for diagnostics awaiting processing by the system error module when a stacked mode of operation is indicated.

PFSYMT4 This area is used to store the level 4 symbol table. The symbol table must be saved for use in initializing the symbol table of a blank-level command following a level 4 command.

PFINPUTB The image of the card, to the right of the semicolon terminating a command, is saved in this area for processing as the start of the following command. (Hexadecimal 00 indicates the end of the image.)

PFSYMT3	This area is used to store the level 3 symbol table. The symbol table must be saved for use in initializing the symbol table of a blank-level command following a level 3 command or the symbol table for a level 4 command following this level 3 command without intervening commands of level 3 or higher.
PFPWVTAB	This table is used as an expedient to determining phrase validity. There are 256 entries corresponding to the 256 possible phrase check sums. A zero entry indicates no valid phrase has the check sum; a nonzero entry is a pointer to the phrase entry table.
PFSYMT2	This area is used to store the level 2 symbol table for use in initializing the symbol table of a blank-level command following a level 2 command or the symbol table of a level 3 command following this level 2 command without an intervening command of level 2 or level 1.
PFINPUTA	This area is used to store the length and the EBCDIC image of the current phrase. PSCAN places the command in this area for access by PHRAS. The subroutine INPUT reads the statement image from this area and places it in memory.
PFSYMT1	This area is used to store the level 1 symbol table for use in initializing the symbol table for a blank-level command following this level 1 command or the symbol table for a level 2 command following this level 1 command without an intervening level 1 command.
PFPAVTB	There is one entry in this table for each record in the phrase entry table. The entry provides information as to the available room within each record for the addition of new phrase definitions.
PFPETAB	This portion of the PFILE contains the language description elements. Each command is entered with header information followed by up to seven tables of phrase definition data. The length of this section is variable up to a maximum of 255 records, a function of the number of commands that must be added into the language dictionary.

The following section describes the detail layout of the variable (maintained) portions of PFILE. Those portions that are merely temporary storage areas are not described.

12.1.1 PFPWVTAB (PHRASE-VERB VALIDITY TABLE)

This section has 256 entries corresponding to the 256 possible phrase check sums. The word check sum of each word in the phrase is calculated as:

```
KSUM = L1*4 + L2*2 + L3
  L1 = First letter in EBCDIC in low-
       order eight bits
  L2 = Second letter in EBCDIC in low-
       order eight bits
  L3 = Third letter in EBCDIC in low-
       order eight bits
```

Only the low-order eight bits of the word check sum are saved. The phrase check sum is formed by the "exclusive or" of succeeding word check sums. The following example illustrates the calculation of the phrase check sum for the phrase "DUMP PLAN":

Word Check Sum Calculations

```
    D       11 0001 0000     310
    U       01 1100 1000     1C8
    M       00 1101 0100     0D4
           101 1010 1100     5AC      AC

    P       11 0101 1100     35C
    L       01 1010 0110     1A6
    A       00 1100 0001     0C1
           101 1100 0011     5C3      C3

    DUM        1010 1100              AC
    PLA        1100 0011              C3
               0110 1111              6F
```

The 256 entries accessed by the phrase check sum have the following format. Each entry contains 16 bits. The term "record/64" in the following discussions means 64 bits on System/360 and 80 bits on the 1130 System. This grouping is one sixty-fourth of a disk record.

```
         | |           |          |
Contents |V|     A     |    B     |
         └─┴───────────┴──────────┘
Bit       0 1 2       7 8        15
```

V Verb Control
0 if no verb phrase has this check sum
1 if a verb phrase has this check sum

A	The number of records/64 from the beginning of the sector indicated by B to the first phrase entry in the chain.
B	Those bits contain the relative sector address (1-255) of the first phrase entry in the chain of phrases with equal check sums. The field is zero if no valid phrase has this check sum.

12.1.2 PSYMT 1,2,3,4 (SYMBOL TABLES)

This section is made up of 255 bytes of information, including 126 (16-bit) words containing the symbol table entries. The format of the table is shown in the following chart:

```
         | | | | |                |
Contents |0|R|L|0|       E        |
         └─┴─┴─┴─┴────────────────┘
Byte      0 1 2 3 4            255
```

R	The relative byte (8-bit) address of the first table entry. The tables are built from left to right. The rightmost entry wraps around to the left end. The last (rightmost) value entered is preceded to the right by a zero entry.
L	The level of the symbol table is indicated as the level minus one. Thus, the indicator occupies the second and third bits and ranges from 0-3.
E	Each symbol is entered in compressed form from the phrase. The table is initialized from the symbol table of the next higher level. The format of the compressed symbol is shown in the chart below. The symbol allows expeditious detection of undefined symbols. Note that the symbol table entry is the same as 1 and 2 of Table 3.

```
         |        |        |        |  |
Contents |Letter 1|Letter 2|Letter 3|0 |
         └────────┴────────┴────────┴──┘
Bit       0      4 5      9 10   14 15
```

The letters are compressed into five bits through the following code compression:

LETTER	COMPRESSED CODE
A-I	1-9
J-R	11-19
S-Z	22-29
blank	0

12.1.3 PFPAVTB (PHRASE AVAILABILITY TABLE)

This section of PFILE contains a maximum of 256 entries corresponding to the number of records in PFPETAB. Each entry is a half-word (16 bits). The entry format is shown in the following table:

```
      |       |        |
Entry |   B   |   L    |
      └───────┴────────┘
Bit    0     7 8     15
```

B	The number of records/64 to the beginning of the first phrase entry or available space entry in the sector. The value of 7FFF (hexadecimal) indicates that the entire sector is available; 8000 (hexadecimal) indicates the end of the table.
L	The number of records/64 in the largest contiguous, available block that begins in this sector. This entry is used as a test for the possible addition of the current phrase into this sector.

12.1.4 PFPETAB (PHRASE ENTRY TABLE)

The available space entries and the phrase entries in the phrase entry table are packed across sector boundaries. The first records/64 of the phrase entry table must be initialized when PLAN is invoked. If it is not, the ADD PHRASE command is set and PHRAS is loaded to add it to PFILE. The format of the PFILE header is shown below in hexadecimal.

```
|    |     |     |     |     |     |     |
|    |  P  | F   | I   | L   | E   |  •  |
|0001| D7  | C6  | C9  | D3  | C5  | 4B  |
└────┴─────┴─────┴─────┴─────┴─────┴─────┘
 0-15 16-23 24-31 32-39 40-47 48-55 56-63
```

Note that bits 16 to 63 contain the EBCDIC representation of PFILE. On the 1130 System, bits 64-79 are included but unused.

The first word (32 bits) of each phrase (or available space) entry provides data as to the size of the entry and pointers to the next item in the chain. The format of this portion of the entry is provided below:

```
| |   | |   |     | |       |     |
|T| L |X|000|  S  | V|    Z  |  SA |
└─┴───┴─┴───┴─────┴─┴───────┴─────┘
 0 1 3 4 5 7 8 15 16 17  23  24  31
```

T	This bit determines whether this is a phrase entry or an available-space entry. 0 = Phrase entry 1 = Available space (The following fields, except S, are meaningless if this is an available-space entry.)
L	These bits (in a phrase entry) define the level of the phrase according to the following table:

000 Level 1
001 Level 2
010 Level 3
011 Level 4
100 Blank level

X The presence of this bit indicates a level zero phrase.

S These eight bits define the number (<128) of records/64 in this entry. No phrase may result in an entry of greater than 128 records/64. The appropriate diagnostic is issued if such an attempt is made.

V This bit (in a phrase entry) defines whether the phrase is a verb or an object phrase.
0 = Object phrase
1 = Verb phrase

Z This six-bit (<64) field defines the number of records/64 (within the sector) that precede the first word of the chained-to (phrase with equal check sums) entry. This entry and the following entry allow direct access of the chained phrase.

SA This eight-bit field (<256) defines the sector address, relative to the first record of the phrase entry table minus one word, of the first word of the next chained-to phrase. This field is zero if this phrase is the last of a chain.

Note that all phrases of equal check sum (as defined under phrase-verb validity table) make up the links of the phrase chain.

Following the phrase entry header, as defined above, are up to eight tables. Each table is ended with 80xx (hexadecimal), where xx is the number of 16-bit half-words in the following table. The last table is terminated with 7FFF (hexadecimal). Trailing tables of zero length are not required, nor is the table length indication (8000) entered.

12.1.5 TABLE 1 (PHRASE NAME)

One word (32 bits) is required for each word in the phrase name. There is a maximum of five double-words used. Letters are coded in EBCDIC code.

```
|         |         |         |          |
|Letter 1|Letter 2|Letter 3|(Not Used)|
└────────┴─────────┴─────────┴──────────┘
 0      7 8     15 16    23 24       31
```

Note that the next table (80xx) or last table (7FFF) indicator is placed in the next half-word.

12.1.6 TABLE 2 (CONSTANT INITIALIZATION DATA VALUES)

This table contains all constant (default or initialization) values. There are four formats for this entry that depend upon the format of the phrase definition. In the following table definitions, the example phrase entry is given, followed in order by the general form of the table entry, the description of the table, and the table entry representing the example phrase entry. Note that there is one entry required for each literal character count plus one for each succeeding group of four literal characters.

1. Constant Value: I(35)10,

```
| | |     |      |
|0|0| S   |  V   |
└─┴─┴─────┴──────┘
 0 1 2 15 16 47
```

S This 14-bit (<16,384) field defines the subscript relative to the beginning of the switch area.

V This 32-bit field defines the initialization value as defined in the phrase entry.

```
| | | |   |   |   |   |   |   |   |   |
|0|0|2|D |0 |0 |0 |0 |0 |0 |0 |A |
└─┴─┴─┴──┴──┴──┴──┴──┴──┴──┴──┴──┘
 0 4 8 12 16 20 24 28 32 36 40 44
```

2. Symbolic Subscript: I(M)DATA3,

```
| |        | |        |          |
|1|   C    | 0|   S    |    V     |
└─┴────────┴─┴────────┴──────────┘
 0 1    15 16 17   31 32       63
```

C This 15-bit field contains the compressed data name in symbol table code that is to be initialized. The symbol is stored in the same compressed code as defined for the symbol table entries.

S This 15-bit field contains the subscript relative to the data name into which the initialization value is stored.

V This 32-bit field defines the initialization value as defined in the phrase entry.

```
| | | |  |         |            |
|9|0|3| 7|  0001   |  00000003  |
└─┴─┴─┴──┴─────────┴────────────┘
 0 4 8 12 16     28 32          63
```

3. Implied DO: I(30,36,2)15,...

```
| | |      |       |       |     |
|0|1|  S   |   D   |   I   |  V  |
L_L_L______L_______L_______L_____J
 0 1 2   15 16   31 32   47 48 79
```

S This 14-bit (<16,384) field contains the subscript associated with the data value relative to the beginning of the switch area.

D This 16-bit field contains the displacement (range) for the implied DO. The value must be a multiple of field I. This value is computed from the first two specified implied DO parameters.

I This 16-bit field contains the increment for the implied DO.

V This 32-bit field contains the initialization value as defined in the phrase entry.

```
| | | | |          |     |        |
|4|0|2|8 |   0006  |0002 |0000000F|
L_L_L_L__L_________L_____L________J
 0 4 8 12 16      31 32 47 48    79
```

4. Symbolic Subscript and Implied DO: (M+2,10,2)NAME1,...

```
| |     | |     |     |     |
|1| CS  | 1|  D  |  I  |  V  |
L_L_____L_L_____L_____L_____J
 0 1  15 16 17  31 32 47 48 79
```

CS This field contains the compressed data name of the starting position to be initialized. The symbol is stored in the same compressed code as defined for symbol table entries.

V This 32-bit field contains the initialization value defined in the phrase entry.

D This 16-bit (<65,536) field contains the displacement from the first position to be initialized to the final position to be initialized.

I This 16-bit field contains the increment between succeeding values to be initialized.

```
| |    |     |       |          |
|B| C2E| 800A|  0002 | 00000001 |
L_L____L_____L_______L__________J
 0     16    32      48        79
```

12.1.7 TABLE 3 (SYMBOL TABLE)

1. Symbol with Constant Subscript and Scale Value: P+2(15)ABC...

```
|     | |    | |    | |        |
| S   | 0|  E  | I|  P  | G|   SUB   |
L_____L_L____L_L____L_L________J
 0 14 15 16-17 18 19-21 22 23   31
```

S This 15-bit field contains the compressed data name to be defined. The format is as defined above for symbol tables.

E This field defines the user-exit number to be associated with this symbol.
00 = No exit
01 = User exit 1
10 = User exit 2
11 = User exit 3

I This field defines the mode for the variable.
0 = Real (floating-point)
1 = Integer (fixed-point)

P This three-bit (<8) field contains the scale factor to be associated with this symbol.

G This one-bit field determines the sign of the scale factor.
0 = Positive
1 = Negative

SUB This nine-bit (<512) field contains the subscript of the value to be entered in the symbol table relative to the first position of the communication array.

```
| | | |  |  |  |  |     |
|0|8|8| 6| 0| 8| 0|  F  |
L_L_L_L__L__L__L__L_____J
 0 4 8 12 16 20 24 28 31
```

2. Symbol with Constant Subscript and No P-value: IU2 (25)VALUE...

```
|       |   |       |   |       |
|  S    | 1 |   E   | I |  SUB  |
L_______L___L_______L___L_______J
 0   14  15  16   17  18  19    31
```

S This 15-bit field contains the compressed data name in the mode indicated for symbol table entries.

E This two-bit field defines the user-exit number to be associated with this data name.

I This one-bit field determines the mode of storage.
0 = Real (floating-point)
1 = Integer (fixed-point)

SUB This 13-bit (<8192) field contains the subscript associated with the data name relative to the switch area.

```
| | | | | | | |        |
|C|8|5| B| A| 0| 1|   9  |
└─┴─┴─┴──┴──┴──┴──┴──────┘
 0 5 8 12 16 20 24 28   31
```

3. Symbols with Symbolic Subscript: (M+2-N)ABC...

The symbolic subscript is indicated by setting SUB to zero. The subscript defining expression is then appended to the symbol table entry in EBCDIC code with a prefixed left parenthesis and a terminating comma.

```
|          |                  |
| 08860000 |  4DD44EF260D56B  |
└──────────┴──────────────────┘
 0          32               87
```

12.1.8 TABLE 4 (PROGRAM LIST)

The program list table is made up of one entry per program in the list.

1. Program Name: M0798,...

```
|                            |
| 8-CHARACTER EBCDIC NAME    |
| (RIGHT-PADDED WITH BLANKS) |
└────────────────────────────┘
0                          63

|D|4|F|0|F|7|F|9|F|8|4|0|4|0|4|0|
└─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┘
 0    16 17 31 32            60
```

2. Checkpoint Return (asterisk)

```
|                   |
| 5C40404040404040  |
└───────────────────┘
 0                64
```

3. Left Parenthesis (EBCDIC)

```
|                   |
| 4D40404040404040  |
└───────────────────┘
 0                64
```

4. Right Parenthesis (EBCDIC)

```
|                   |
| 5D40404040404040  |
└───────────────────┘
 0                64
```

12.1.9 TABLE 5 (DATA CHECK ENTRIES)

1. Test, Abort, Generate PLAN Literal: (5)*,...

```
| |   |           |          |
|0| * |    SUB    |   CTL    |
└─┴───┴───────────┴──────────┘
 0 1 2 3        15 16       31
```

* This two-bit field contains the condition code.
00 = *
01 = *R
10 = *T
11 = *F

SUB This 13-bit (<8192) field contains the subscript relative to the switch area of the PLAN word to be tested.

CTL If this field is nonzero, there is a suffix section, as defined under 4 and 5, starting at field "F".

```
|    |    |    |       |
|  0 |  0 |  0 |   5   |
└────┴────┴────┴───────┘
 0    4    8   12   15
```

2. Test, Abort, Generate PLAN Literal; Symbolic Subscript: (M)NAME*R,...

```
| |    |         | |           |          |
|0| *  | -0-     | 0|   SYM    |   CTL    |
└─┴────┴─────────┴─┴───────────┴──────────┘
 0 1  2 3      15 16 17      31 32       47
```

* (See above)

SYM This 15-bit field contains the compressed data name in the format as defined for symbol tables.

CTL (See above)

```
|    |    |    |      |   |   |   |   |
|  2 |  0 |  0 |    0 | 3 | C | 2 | E |
└────┴────┴────┴──────┴───┴───┴───┴───┘
 0    4    8    12    16  20  24  28
```

3. Same conditions as above:
Same as previous example, plus: ,*F

```
| |   |       | |       |       |       |
|0| * | -0-   | 1| SYM  |  SUB  |  CTL  |
└─┴───┴───────┴─┴───────┴───────┴───────┘
 0 1 2 3    15 16 17  31 32   47 48   63
```

* (See above)

SYM (See above)

CTL (See above)

SUB This 15-bit (<32,768) field contains the subscript relative to the data name that is to be checked.

```
| | | |  |  |  |  |  |  |  |  |  |
|6|0|0| 0| B| C| 2| E| 0| 0| 0| 2|
└─┴─┴─┴──┴──┴──┴──┴──┴──┴──┴──┴──┘
 0 4 8 12 16 20 24 28 32 36 40 44
```

Note: In the following examples the formats defined in 1, 2, or 3 above remain the same as a function of conditions except for bit 0 and the last 15-bit field. Bit 0 will indicate whether the literal to be processed is implicit (1) or explicit (0). The last 15-bit field will contain function information for the literal processing.

4. Process Implicit Literal: ( )*TZ(9)

Note: Z in the above example is a user-given function code and will be reflected in the F field below according to the following table.

```
If Z = A (Abort) then F = 00
     = C (Continue)     = 01
     = P (Phrase)       = 11
     = b (List)         = 10
```

```
| |                   |   |     |
|1| SAME AS 1, 2 OR 3 | F | SUB |
└─┴───────────────────┴───┴─────┘
 0 1
```

F See above table.

SUB This 14-bit (<16,384) field contains the subscript relative to the start of the communication array that contains the literal to be processed.

5. Process Explicit Literal: ( )*TZ'LITERAL'

```
| |                   |   |     |      |
|0| SAME AS 1, 2 OR 3 | F | L   |   Q  |
└─┴───────────────────┴───┴─────┴──────┘
 0 1                 n 0 1   15 16   n
```

F Same as example 4.

L This 14-bit field contains the length of the literal in 16-bit words.

Q This variable-length field contains the literal in EBCDIC packed format.

12.1.10 TABLE 6 (PHRASE-DEFINED EXPRESSIONS)

This table is made up of two sections. The following three examples define the format of the possible first-section entries:

1. Value with Scale Factor: P+3(7) A=A*.017453...

```
|     |   |      | |       |       |
| 1 0 | I |   P  |G|   S   |   T   |
└─────┴───┴──────┴─┴───────┴───────┘
0   1 2   3    5 6 7    15 16     n
```

I This field designates the storage mode of the data value.
0 = Real (floating-point)
1 = Integer (fixed-point)

P This three-bit (<8) field designates the scale factor to be applied to the result of the expression before storage.

G This bit designates the sign of the scale factor.
0 = Positive
1 = Negative

S This nine-bit (<512) field contains the subscript associated with the data value relative to the first position of the communication array.

T This variable-length field contains the text of the phrase-defined expression terminated with a comma. The text is compressed to eliminate meaningless blanks and characters.

```
|  |  |   |   |   |   |   |   |       |
|8C|07| C1| 7E| C1| 5C| 4B| F0| F1... |
└──┴──┴───┴───┴───┴───┴───┴───┴───────┘
0  8  16  24  32  40  48  56  64
```

2. Values without Scale Factors: I(12) I=I*12...

```
|    | |     |       |
| 11 |I|  S  |   T   |
└────┴─┴─────┴───────┘
0  1 2 3   15 16     n
```

I See above.

S This 13-bit (<8,192) field contains the subscript of the data value relative to the start of the systems switch area.

T See above.

```
|   |   |   |   |   |   |   |   |   |
| E0| 16| 7E| C9| 7E| C9| 5C| F1| F2|
└───┴───┴───┴───┴───┴───┴───┴───┴───┘
0   8   16  24  32  40  48  56  64
```

3. Value with Symbolic Subscript: (m+5) A,:(B>0)

```
|    |       | |      |     |
| 00 |   S   | 0|   C  |  T  |
└────┴───────┴──┴──────┴─────┘
0  1 2    15 16 17   31 32   n
```

S This 14-bit (<16,384) field contains the subscript relative to the data name into which the result of the expression evaluation is stored.

C This 15-bit field contains the compressed data name in the symbol table code.

T See above.

```
|  |  |   |   |   |   |   |   |   |   |
|00|02| 04| 00| 7A| 4D| C2| 6E| F0| 5D|
└──┴──┴───┴───┴───┴───┴───┴───┴───┴───┘
 0  8  16  24  32  40  48  56  64  72
```

The second portion of this table contains the expression area in compact literal form (excess blanks and characters eliminated). This portion of the table is introduced with a dollar sign ($).

12.1.11 TABLE 7 (USER-EXIT LIST)

This table is in a format identical to Table 4 and contains the program list defined following the keyword EXIT. The table, when present, always contains three entries.

12.1.12 TABLE 8 (VERB PROGRAM LIST)

This table is in a format identical to Table 4 and contains the program list defined following the term VERB at phrase definition time.

12.2.0 PLAN FILE CONTROL BLOCKS

The following charts provide the content of the PLAN DYNAMIC file control blocks. Note that because of the integer word size differences (16-bit versus 32-bit), the 1130 PLAN system has a different format from that of the System/360 OS or DOS PLAN. The table given below provides the format for the System/360 OS/DOS PLAN.

```
ID(1)                            ID(2)
|    |         |        |       |          |
|  0 |   TTR   |   D    |   N   |    S     |
└────┴─────────┴────────┴───────┴──────────┘
 0    1         20    23 24   31 32       63
```

TTR	This 19-bit field contains the TTR of the FDR for this file.
D	This three-bit (<8) field contains the logical drive code for this file.
N	This 8-bit (<256) field contains the file identification number. This field is originally set by the user before issuance of the CALL FIND. All other fields within ID(1) are set as a result of CALL FIND or CALL WRITE operations.
S	This 32-bit field contains the current size of the file in words.

The following chart defines the 1130 DYNAMIC file control block:

```
ID(1)                         ID(2)
|      |   |     |      |      |      |   |   |
|0  R  | P |  N  |  D   |  F   |  S   | A | C |
└──────┴───┴─────┴──────┴──────┴──────┴───┴───┘
 0    4 5 7 8 15 16  19 20  31 32  47 48    56
```

R	(reserved)
P	This three-bit field contains the file priority.
N	(see above)
D	(see above)
F	This twelve-bit (<640) field contains the physical address of the first record in the last extent accessed.
S	(see above)
A	This eight-bit (<50) field contains the relative allocated segment number of the first segment of the last extent accessed.
C	This eight-bit (<50) field contains the number of contiguous segments in the current extent.

The following charts define the format of the PERMANENT (GDATA, RDATA, WDATA) file control blocks. The file ID number is set by the user before issuing the CALL GDATA. All other fields are defined as a result of the CALL GDATA and are modified by CALL RDATA.

System/360 OS/DOS PLAN

```
ID(1)                                      ID(2)
|           |       |        |        |        |
|    00     |  7F   |   D    |   N    |   S    |
└───────────┴───────┴────────┴────────┴────────┘
 0         7 8    15 16    23 24    31 32    63
```

D	This eight-bit (<8) field indicates the logical drive code as 0-7.
N	This eight-bit (<64) field contains the number of the file.
S	This 32-bit field contains the size of the file in 32-bit words.

IBM 1130 PLAN

```
ID(1)                          ID(2)
|       |    | |     |       |     |      |
|       |    | |     |DISK   |     |(NOT  |
|7F(HEX)| N  |1|LDR  |ADDRESS|  S  | USED)|
|_______|____|_|_____|_______|_____|______|
 0     7 8 15 16 17 20 21   31 32 46 47  63
```

LDR   This four-bit field contains the logical drive code.

13.0.0 APPENDIX F: PLAN SYSTEM DIAGNOSTIC PROCESSING

This appendix contains a discussion of the control of diagnostic processing and lists diagnostic messages generated by various PLAN components through linkage to the error processor PERRS. The format of PLAN system diagnostics is shown below:

```
DFJ000 001-100 TEXT
       101-200 TEXT
       201-300 TEXT
       301-400 TEXT
       401-450 TEXT
CCCnnn *A* mmmmm SEQ=yyy ID=ccccc
PG=xxxxxxxx DIAGNOSTIC
```

The segments of the diagnostic message underlined in the above example are variable. Functions defined by the variable data are:

TEXT	This field of up to five lines contains the current input statement. It is printed only if the long-form diagnostic is requested (see "Switch Words", 4.3.22). Character positions are printed to the left of the text.
CCC	This three-character field is DFJ if the diagnostic is generated by PLAN and *** if generated by the user.
nnn	This three-digit number is the error number assigned by the call to the error routines as calling parameter N1. In PLAN error diagnostics, this number is merely a diagnostic modifier (index).
A	This character specifies the action taken following generation of the literal. R indicates that execution of the current command is terminated. PLAN error recovery is initiated. C indicates that following generation of the diagnostic, the execution of the current command is continued. E indicates that the current execution of PLAN is terminated. O indicates a pause for operator intervention.
mmmmm	This is a five-digit modifier (ECODE) that provides additional information about the error. This parameter is provided as N2 in the call to the PLAN error subroutines.
SEQ=yyy	This field provides the statement sequence of this PLAN statement relative to the beginning of the PLAN job stack.
ID=ccccc	This five-character field provides the identification field (cc. 76-80) of the last card of the current PLAN statement.
PG=xxxxxxxx	This field provides the name of the program in execution at the time the call to the error routine is issued.
DIAGNOSTIC	This field contains the literal text of the diagnostic message and is limited to 76 characters.

13.1.0 PLAN ERROR PROCESSING

Since the PLAN system is a monitor which supervises the execution of other problem programs, it must have the ability to detect abnormal conditions.

There are four types of errors the PLAN system can detect and these are:

- Phrase Definition Errors
- Command Errors
- Execution Errors
- User-Defined Errors

1. Phrase definition errors are detected by the PLAN system module "PHRAS" when a phrase is being entered into the PLAN language dictionary.
2. Command errors are detected by the PLAN system module "PSCAN" while processing commands.

3. Execution errors are detected by the PLAN system mainline while a problem program is in execution.

4. User-defined errors are the result of a programmed call to one of the error subroutines (ERROR, ERRET, ERREX, ERRAT).

Each type of error discussed is detected by a different module and at a different point in time. The technique used to produce a diagnostic in this environment may be described as follows: When an error is detected by any component of the system, the type of error is recorded and a generalized diagnostic processing module is called to produce the required error message. The PLAN system module that produces diagnostic messages is "PERRS".

The PLAN system offers the user several options in processing errors. Several terms are defined below that are used in describing these options.

SHORT FORM. The diagnostic message is produced without printing the phrase that caused it.

LONG FORM. The phrase that caused the diagnostic is printed with the diagnostic message.

IMMEDIATE MODE. The error processing module "PERRS" is invoked at the time the error occurs, even if a checkpoint is required.

STACKED MODE. A condensed version of the error is recorded in the error message stack which will be processed the next time "PERRS" is invoked by the system.

ERROR MSG STACK. An area on PFILE is reserved exclusively for recording errors in a condensed form. This gives the system the ability to delay calling the diagnostic processor "PERRS" until the program area is available.

ERROR MSG QUEUE. DYNAMIC file 255 on PLAN DYNAMIC drive 0 is reserved as a queue area for diagnostic messages. This gives the system the ability to post-list diagnostic messages by writing the messages on the file as they occur and then dumping the file on command.

USER-ERROR EXIT. The PLAN system has the ability to call a user-error processing module in the cases where the normal PLAN mode of diagnostic presentation is not appropriate for the application.

13.2.0 SPECIFYING ERROR PROCESSING MODE

The mode of error processing by the PLAN system is controlled by the PLAN Switch Words 11, 12, and 13. These switch words can be set by any PLAN command. The standard error processing mode is as follows:

1. Errors are stacked.
2. Error message format is short.
3. No error messages are queued for post-listing.
4. No user-error processing module will be called.
5. Messages are printed on the standard PLAN output device.
6. Errors detected by the PLAN DYNAMIC file routines cause a phrase abort.
7. Errors detected by the PLAN PERMANENT file routines cause a phrase abort.

Switch Words 11-13 are normally set by the following operands of the PLAN JOB command:

1. NERM
2. DEVICE
3. UMOD
4. SHORT
5. LONG
6. STACK
7. IMM
8. DFI
9. PFI

NERM specifies the number of error messages to be written on the error message queue file before they are dumped on the error message device.

DEVICE specifies the sequential file device code (NOD argument for PLINP/PLOUT subroutines) to which the diagnostic messages are to be written.

UMOD specifies the EBCDIC name of a user-error processing module to be called by the error processor "PERRS" when an error is processed.

SHORT specifies that the SHORT form of the diagnostic is to be used when an error message is produced.

LONG specifies that the LONG form of the diagnostic message be used when an error message is produced.

STACK specifies that the system is to optimize error message processing by using the error message stack in PFILE to record messages until "PERRS" can be called without a checkpoint.

IMM specifies that "PERRS" is to be invoked at the time the error occurs.

DFI specifies that a phrase abort condition is not to occur on certain error conditions detected by the DYNAMIC file support subroutines routines (see "DYNAMIC File Routines", 5.11.0).

PFI specifies that a phrase abort condition is not to occur on certain error conditions detected by the PERMANENT file support subroutines (see "PERMANENT File Routines", 5.11.3).

If both SHORT and LONG are specified, the LONG-form option is used. If both STACK and IMM are specified, the IMMEDIATE option is used.

Use of the operands PFI and DFI requires the application program to process the error conditions that would normally abort the PLAN statement. If these operands are specified and the required programming is not present, unpredictable results can occur. What generally takes place is the following: When the error is detected the file control block is closed, and on the next reference to the file, an error message indicating an unopened file control block is issued. This masks the real reason for the error condition.

13.3.0 STANDARD ERROR PROCESSING

Normally, the PLAN system will process errors at SHORT form and in a stacked mode. The reason for using this technique is that the size of the PLAN error processing module is such that if the program area is not free, a checkpoint is required to load and execute "PERRS". Delaying the call to "PERRS" until the program area is free eliminates the need for a checkpoint and so improves performance. The error message stack has a finite limit on the number of messages it can contain, and in cases where the stack overflows, a checkpoint is forced and "PERRS" empties the stack.

13.4.0 POST LISTING OF ERRORS

Some applications may require that error messages be suppressed until end of job. An example of this is a compiler, such as FORTRAN or COBOL, where the error messages are listed at the end of the compilation. The PLAN system provides this facility to the user as a standard option. In order to use this facility the PLAN system must have available PLAN DYNAMIC drive 0. DYNAMIC file 255 is used as an error message queue file. To invoke this facility the user must specify a value in system Switch Word 11.

The value in this switch word is used by the error processor "PERRS" to determine the number of error messages to write on the error message queue file (drive 0, file 255) before dumping the file on an output device.

The message records on this file are written as 21-word or 124-character records. The first word of the record is an integer from -3 to +12, and is used as an argument for the PCCTL subroutine to effect carriage control for the data line that is contained in words 1-24 (characters 4-123). The data portion must be alphameric data in the A4 format. The data area of records produced by "PERRS" contains the PLAN system diagnostic message text. The user may write records directly to this file from an application program by using the PLAN subroutine EWRIT (see "Error Interface Subroutines", 5.11.6).

The PLAN error message queue file is dumped on the diagnostic device under the following conditions:

1. The number of diagnostics messages added to the queue file exceeds NERM.

2. The subroutine ERLST is called.

3. The end of PLAN input (/*) is read by PSCAN.

4. A level 0 phrase is processed.

5. A level 1 phrase is processed.

13.5.0 USER-ERROR EXIT PROCESSING

If a user module name is specified in system Switch Words 11 and 12, by specifying UMOD'NAME', the PLAN error processor PERRS creates an array in ERASABLE COMMON that describes the error and then invokes the named module through the PLAN LOCAL facility. This array is in the following format:

BYTE	CONTENTS
0-7	Program name issuing diagnostic call
8-11	Error number (N1 from error subroutine call)
12-15	Error code (N2 from error subroutine call)
16-20	ID from cc. 76-80
21	hexadecimal FF=system error, 0=user error
22	hexadecimal FF=abort, 0=continue
23	(unused)
24-27	Sequence
28-31	Length of literal in characters
32-107	Literal text
108-111	Character count of phrase
112-561	Phrase text

A program written as a user-error processor may not use the following PLAN subroutines ERROR, ERRAT, ERREX, ERRAT, ERLST, LREPT, LCHEX, LREPT, and PUSH. Any error detected while a user-error processing module is in control causes cessation of all error processing.

The UMOD and the NERM or DEVICE specifications are mutually exclusive. Therefore, the automatic PLAN facility for post-listing of errors (as described in 13.2.0) is not available, if a user-error processing module is used. The same effect may be produced, however, by using the subroutine EWRIT to create an error message queue file. A dump of the file may be forced by using the LIST subroutines to place the name PEDMP into the pop-up list. This module will force a dump of the error message queue file and will also terminate the current statement.

13.6.0 PHRAS DIAGNOSTICS

The following diagnostics are generated from errors detected by PHRAS (the ADD PHRASE processor). ECODE (m) for all diagnostics generated by PHRAS is a pointer to the position at which the error condition was detected, except as otherwise noted. Position one is the first character of the command. The format of the descriptions of the diagnostics is:

- DIAGNOSTIC NUMBER(n), ACTION CODE, DIAGNOSTIC •
  REASON

- 21 *C* PHRASE TO DELETE CANNOT BE FOUND •
  A phrase that is to be deleted is not currently in PFILE. This can result from a DELETE PHRASE or an ALTER PHRASE. If it results from an ALTER PHRASE, the ADD PHRASE aspect of the command is not suppressed.

- 22 *R* NO ROOM TO ADD PHRASE •
  There is no contiguous vacant area in PFILE large enough to allow the current phrase to be added. PFILE must be reorganized, reestablished, or expanded.

  Usually, some space can be gained by reorganizing the file without changing its size. This is accomplished by deleting the phrases and then re-adding them.

  Additional phrases may be provided for by enlarging PFILE if it is currently smaller than the maximum size. PFILE must be at least 14 sectors (17 sectors on 1130 PLAN) and not more than 268 (205 on 1130 PLAN) sectors in length. This will also require that the phrases for the system be re-added

- 23 *R* PHRASE ALREADY DEFINED •
  An attempt to add a phrase that already exists in PFILE has been made. If the phrase to be added is a replacement for the existing phrase, the existing phrase must be deleted (DELETE PHRASE or ALTER PHRASE, see 4.5.1 and 4.5.3) before the new phrase can be added.

- 24 *R* INVALID FORMAT IN PROGRAM LIST •
  A program list defined with the ADD (ALTER) PHRASE is found to contain invalid syntax. This can result from an unrecognizable numeric or special character

- 25 *R* INVALID FORMAT IN USER-EXIT PROGRAM LIST •
  This error may result from:
  a. A program name not starting with an alphabetic character
  b More than three programs in the list

  (Note that errors in the user-exit program list may also be diagnosed as error number 24.)

- 26 *R* KEYWORD ENTRY NOT TERMINATED BY COMMA OR SEMICOLON •
  A keyword (symbol table entry, PROG, VERB, EXIT, or LEVEL) has been collected, but the keyword and associated data was not terminated with a comma or semicolon.

- 27 *R* LEVEL NUMBER GREATER THAN 4 •
  The number collected following the specification word LEVEL is greater than 4.

- 28 *R* NO SYMBOL DEFINED AFTER EXECUTION-DEFINED SYMBOL SUBSCRIPT EXPRESSION •
  A symbolic subscript expression requires a symbol (name) to be defined. The required symbol has not been found.

- 29 *R* CONSTANT SUBSCRIPT ZERO OR LESS THAN -15 •
  A constant subscript has been encountered that does not describe a valid location

in the system switch words or communication array.

- 30 *R* IMPLIED DO SUBSCRIPT NOT FOLLOWED BY SINGLE-VALUED CONSTANT •
The value following an implied DO subscript was not found to be a single-valued constant, that is, numeric, +, or -. This error can result from an implied DO subscript followed by:

  a. A literal default, that is, "ABC"
  b. No default value

- 31 *R* SYMBOL SUBSCRIPT GREATER THAN 8176 OR 511 WITH P-VALUE •
A constant subscript that defines a symbol exceeds the maximum allowable value of 8176 without scale values (P values) or 511 with scale values.

- 32 *R* EXECUTION-DEFINED SYMBOL FOLLOWED BY IMPLIED SYMBOL •
A symbol that is implied follows a symbol associated with a symbolic (execution-defined) subscript. There may not be an implied symbol after a symbolic subscript.

- 33 *R* PHRASE DEFINITION INVALID •
A phrase is not defined properly, that is the phrase name is syntactically incorrect. This can be caused by:

  a. Failure to end the phrase definition with a comma
  b. Use of nonalphabetic characters within the phrase definition

- 34 *R* SUBSCRIPT FOR DATA VALUE GREATER THAN 16,368 •
A communication array subscript greater than 16,368 has been encountered.

- 35 *R* INVALID CHARACTER •
The ECODE pointer indicates a character that is invalid in a phrase definition. This error can result from an error within the phrase further to the left that was undetectable at that phase of the scan.

- 36 *R* BCD LEFT PARENTHESIS IN LOGICAL EXPRESSION •
All characters in a logical expression must be punched in the EBCDIC code.

- 37 *R* USER-EXIT NUMBER GREATER THAN 3 •
User exits must be 1, 2, or 3.

- 38 *R* FORMULA NUMBER USED BEFORE FORMULA AREA •
A conditional exit includes a formula number, but a $n introducing the expression area has not been encountered.

- 39 *R* FORMULA NUMBER ZERO OR GREATER THAN 1024 •
The valid range for formula numbers is from 0 to 1024 in a phrase definition.

- 40 *R* UNDEFINED FORMULA NUMBER IN FORMULA AREA •
A transfer type formula has been encountered that references a nonexistent formula number. ECODE is set to the formula number found to be in error.

- 41 *R* MULTIPLE DEFINITION OF FORMULA NUMBER IN FORMULA AREA •
More than one formula is identified with the same number within this phrase.

- 42 *R* INVALID FORMAT IN FORMULA AREA •
Formula numbers must be followed by:
  a. Another formula number
  b. Expression
  c. Symbol
  d. Semicolon
  e. Comma

- 43 *R* P-VALUE GREATER THAN 7 •
A scale factor greater than plus seven or less than minus seven has been encountered.

- 44 *R* KEYWORD 'PROGRAMS' NOT FOLLOWED BY PROGRAM LIST •
A program specification has been encountered, but a program list is missing. This can result from the next significant character not being a quotation mark, double quote, or commercial at sign.

- 45 *R* INVALID FORMAT IN RELATIONAL EXPRESSION •
A syntax error has been encountered in a relational expression. Possible reasons for this error are:
  a. Unbalanced parentheses
  b. A semicolon invalid within (not at end of) an expression

- 46 *R* PROGRAM NAME CONTAINS TOO MANY CHARACTERS •
The maximum allowable length for a program name is eight characters on System/360 or five characters on the 1130.

- 47 *R* SEMICOLON IN LITERAL OR EMPTY LITERAL •
A semicolon is an invalid literal character. This diagnostic may result from failure to include the terminal quotation mark of a literal. The phrase terminating semicolon may then appear to be within the literal. A zero-length literal is invalid.

- 48 *R* INVALID FORMAT IN SYMBOLIC SUBSCRIPT EXPRESSION •
The indicated position contains a character that forms an invalid context for a subscript (arithmetic) expression. These conditions include:
  a. Adjacent arithmetic operators

b. Unmatched parenthesis
c. Invalid characters
d. Expression does not end with comma

• 49 *R* USER EXITS NOT ALLOWED ON NEGATIVE SUBSCRIPTS •
An attempt has been made to define a user exit to store data in the switch area.

• 50 *R* INVALID FORMAT IN LOGICAL OR ARITHMETIC EXPRESSION •
ECODE points to a character that may not be contained in the context of a logical or arithmetic expression. These conditions include:
a. Adjacent arithmetic operators
b. Unmatched parenthesis
c. Invalid characters
d. Expression does not end with comma

• 51 *R* INVALID FORMAT IN SUBSCRIPT EXPRESSION •
The indicated position contains a character that forms an invalid context for a subscript (arithmetic) expression. These conditions include:
a. Adjacent arithmetic operators
b. Unmatched parenthesis
c. Invalid characters
d. Expression does not end with comma

• 52 *R* EXPRESSION SUBSCRIPT GREATER THAN 8176 OR 511 WITH P-VALUE •
The symbolic subscript that is associated with a phrase-defined expression is greater than 8176 (if a scale factor is not defined) or greater than 511 (if a scale factor is defined).

• 58 *R* NUMBER OUTSIDE ALLOWABLE FLOATING-POINT RANGE •
A number has been given that cannot be represented in the floating-point representation of the PLAN system. The maximum limit is 2^{127} and the minimum limit is 2^{-128} on the 1130. On System/360 the maximum limit is 16^{63} and the minimum limit is 16^{-63}.

• 64 *R* PHRASE ENTRY TOO LARGE •
The total phrase size is greater than 1024 bytes and will not be added, or one of the eight internal phrase tables is longer than 512 bytes. ECODE is either the total size of the phrase or the PFILE internal table number (see 12.1.5 to 12.1.12) that is too large.

• 65 *R* ILLEGAL SYMBOL - CANNOT BE 'E' •
A data name has been defined to be E. E is not allowed because of syntactical confusion with the exponential indicator E.

• 66 *R* INVALID FORMAT IN IMPLIED DO SUBSCRIPT •
A syntactical error has been encountered. Reasons for this diagnostic may be:
a. The increment (I_3) is negative.
b. The limit (I_2) is negative.
C. The limit divided by the increment is not a whole number.
d. (I_2) or (I_3) is not a numeric constant.

• 68 *R* LEG OF CONDITIONAL EXPRESSION NOT EXPRESSION OR FORMULA NUMBER •

The TRUE action leg or FALSE action leg of a conditional expression is not an expression (example: ?=B*100) or a formula number (example: ?$5).

• 70 *R* CHECK-ENTRY SUBSCRIPT GREATER THAN 8176 •
The constant subscript that is associated with a check entry is greater than 8176.

• 71 *R* INVALID FORMAT IN CHECK-ENTRY LITERAL •
A check entry must be in the following format when the literal option is exercised:

```
*A'LITERAL'
*C'LITERAL'
*RC(SUBSCRIPT)
```

The following condition may have been detected:
a. A semicolon within the literal
b. Quotation marks unmatched
c. A subscript greater than 16,383

• 72 *R* UNBALANCED PARENTHESIS IN PROGRAM LIST •
An unequal number of left and right parentheses have been found in a program list.

• 80 *C* UNREFERENCED FORMULA NUMBER IN FORMULA AREA **UPDATE NOT SUPPRESSED** •
The formula area has been found to contain a formula number that is not referenced in another expression. ECODE defines the formula number that is unreferenced.

13.7.0 EXECUTION-TIME DIAGNOSTICS

The following errors are detected during execution of logic modules operating within the PLAN environment. All 100 series errors result in a PLAN "Phrase Abort" and subsequent level error recovery. The format of the definitions for this section is:

• NUMBER *ACTION CODE* DIAGNOSTIC •
PROGRAM INDICATED
ECODE MEANING
REASON FOR ERROR

• 101 *R* PROGRAM NAMED NOT IN PLAN LIBRARY •

Program: Program name not found
ECODE: Unused.
Reason: The named program was not in the search of the PLAN library

• 102 *R* INVALID COMMON DEFINITION ENCOUNTERED •
Program: Program name.
ECODE: Unused.
Reason: The length of COMMON for the named program is less than 640 FORTRAN (32-bit) words.

• 103 *R* PROGRAM TOO LARGE FOR AVAILABLE MEMORY •
Program: Program name.
ECODE: Unused.
Reason:
a. The size of the named program exceeds the size of the available area for program loading.
b. This message may be given by PSCAN if a program to be loaded as a user-exit program would overlay PSCAN (1130 only).

• 104 *R* PROGRAM NAME IN INVALID FORMAT •
Program: '••••••••' (Unpredictable)
ECODE: Unused.
Reason: An invalid program name has been found in the pop-up list.

• 105 *R* PROGRAM FORMAT INVALID •
Program: Program name.
ECODE: Unused.
Reason: The named program is in overlay, scatter mode or contains TESTRAN symbol cards on OS PLAN or is not in core image format on 1130 PLAN.

• 110 *R* CHECKPOINT PROCESSING INVALID •
Program: Last program entered.
ECODE: Unused.
Reason:
a. An * encountered without a checkpoint being in effect.
b. A checkpoint call when either there is no checkpoint file or insufficient room to write the complete checkpoint.
c. A program to be checkpointed has been overlaid by COMMON (DOS only).
d. LCHEX subroutine is not in the calling module (DOS only).

• 111 *R* OVER 50 NAMES IN POP-UP LIST •
Program: Last program entered.
ECODE: Unused.
Reason: An attempt to place more than 50 names in the pop-up list has been made.

• 112 *R* LOCAL PROCESSING INVALID •
Program: Program issuing CALL LOCAL.
ECODE: Unused.
Reason:
a. There is not room to load the program called as a LOCAL.
b. LOCAL caller overlaid by program named (DOS only).
c. LOCAL caller overlaid by COMMON (DOS only).
d. LOCAL subroutine not in the calling module (DOS only).
e. LOCAL caller is itself a PLAN LOCAL (1130 only).

• 113 *R* LSAV OR LRLD PROCESSING INVALID •
Program: Program issuing loader call.
ECODE: Unused.
Reason: A second CALL LSAV has been processed without an intervening CALL LRLD or a call to a loader function has been processed without the loader in memory. On System/360 all calls to LSAV or LRLD are invalid.

Note: In all 120-130 series diagnostics ID(1) is set to a closed status. Any further attempt to read or write to the file without reopening the file will result in a phrase abort, and PLAN level error recovery will be invoked.

• 120 *R* UNOPENED FILE CONTROL BLOCK ON CALL READ/WRITE •
Program: Last program entered.
ECODE: File number.
Reason: ID(1) in the file control block is in a closed status.

• 122 *R* INVALID DRIVE CODE OR FILE CONTROL BLOCK ON CALL FIND/RELES •

Program: Last entered.
ECODE: Unpredictable.
Reason:
a. File number is zero.
b. Drive code is negative or greater than 7 (DOS/OS) or greater than 4 (1130).

• 123 *R* INVALID FILE CONTROL BLOCK ON CALL READ/WRITE •
Program: Last program entered.
ECODE: Unpredictable.
Reason:
a. ID(1) has been altered.
b. The file specified by ID(1) has been released because of an allocation request for a higher-priority file.
c. The file specified by ID(1) was automatically released because a phrase of higher priority than the file was processed. This can apply only to ID control blocks that reside in COMMON through phrase boundaries.
d. The file control block for PFND1 support may not have

been located on an even word boundary.

• 124 *R* INVALID KDIS/KOUNT ON CALL READ/WRITE •
Program: Last program entered.
ECODE: File number.
Reason: KDIS or KOUNT is negative of KDIS+KOUNT exceeds maximum file size (32,767 on 1130).

• 125 *R* DYNAMIC DRIVE NOT MOUNTED •
Program: Last entered.
ECODE: File number.
Reason: A DYNAMIC drive required by a CALL FIND/READ/WRITE/RELES is not available to the system.

• 126 *R* INSUFFICIENT SPACE FOR ALLOCATION ON CALL FIND/WRITE •
Program: Last entered.
ECODE: File number.
Reason: a. On a CALL FIND insufficient space is available to satisfy the NALLO argument.
b. On a CALL WRITE insufficient space is available for secondary allocation.

• 128 *R* PACK ID NOT EQUAL ON VALIDITY CHECK •
Program: Last entered.
ECODE: File number.
Reason: (1130 only) A request for pack verification has resulted in a test failure.

• 129 *R* PFIND NOT IN PLAN LIBRARY •
Program: Last program entered.
ECODE: File number.
Reason: PFIND was not found at PLAN initialization time, and a subsequent call to FIND, READ, WRITE, FINDL, PFSPC, or RELES has been processed (on 1130 PLAN only).

• 130 *R* UNOPENED FILE CONTROL BLOCK ON CALL RDATA/WDATA •
Program: Last program entered.
ECODE: File number.
Reason: ID(1) in the file control block was not initialized.

• 132 *R* INVALID DRIVE CODE OR FILE CONTROL BLOCK ON CALL GDATA •

Program: Last entered.
ECODE: Unpredictable.
Reason: a. File number is zero.
b. Drive code is negative or greater than 7 (DOS/OS) or greater than 4 (1130).
c. File name is not in PLAN library.

• 133 *R* INVALID FILE CONTROL BLOCK ON CALL RDATA/WDATA •
Program: Last program entered.
ECODE: Unpredictable.
Reason: ID(1) has been altered

• 134 *R* INVALID KDIS/KOUNT ON CALL RDATA/WDATA •
Program: Last program entered.
ECODE: File number.
Reason: KDIS or KOUNT is negative or KDIS + KOUNT exceeds the maximum file size (32,767 on 1130).

• 135 *R* PERMANENT DRIVE NOT FOUND •
Program: Last program entered.
ECODE: File number.
Reason: The DYNAMIC drive is not defined or cannot be found on this system.

• 140 *R* INVALID RECORD LENGTH ON CALL PSORT/PMERG •
Program: PSRTA
ECODE: File number.
Reason: Word 1 of the sort control list is minus or greater than 512.

• 141 *R* INVALID SORT CONTROL FIELD COUNT ON CALL PSORT/PMERG •
Program: PSRTA
ECODE: File number.
Reason: The number of sort fields is specified as negative, zero, or greater than 99 or extends beyond the end of COMMON.

• 142 *R* INVALID SORT CONTROL FIELD ON CALL PSORT/PMERG •
Program: PSRTA
ECODE: File number.
Reason: a. Word 1 of the sort control field is out of range (-6 to +6).
b. Boundary alignment of displacement is invalid for type of sort.
c. The sort field extends beyond the length of the record.
d. The number of element specified is not a positive integer.

• 143 *R* INSUFFICIENT FILE SPACE TO EXECUTE PMERG FUNCTION •
Program: PMRGA
ECODE: Merge file number.
Reason: The required space for the output file of the merge is not available.

• 144 *R* INSUFFICIENT WORK AREA IN MANAGED AREA FILE FOR PSORT FUNCTION •
Program: Program calling PSORT.
ECODE: File number.
Reason: Self-explanatory

Note: This can also result from a call to PMERG on 1130.

• 145 *R* MERGE FILE OUT OF SEQUENCE ON CALL PMERG •
Program: PMRGA
ECODE: File number.
Reason: Self-explanatory.

• 146 *R* UNOPENED FILE CONTROL BLOCK ON CALL PSORT/PMERG •
Program: Program calling PSORT/PMERG
ECODE: File number.
Reason: The file control block specified is found not to be properly opened.

• 147 *R* FILE TO SORT DOES NOT EXIST •
Program: PSRTA
ECODE: File num.
Reason: Specified file cannot be found on the drive specified in the file control block.

• 148 *R* DRIVE CONTAINING FILE NOT AVAILABLE •
Program: PSRTA
ECODE: File number.
Reason: The drive specified by the file control block is not available (1130 only).

• 150 *R* INVALID RECORD LENGTH ON CALL GSORT/GMERG •
Program: Last entered.
ECODE: Record length.
Reason: Word 3 of the sort control list is minus or greater than 512 (System/360 only).

• 151 *R* INVALID SORT CONTROL FIELD COUNT ON CALL GSORT/GMERG •
Program: Last program entered.
ECODE: Sort field count.
Reason: The number of sort fields is specified as negative, zero, or greater than 98 (System/360 only).

• 152 *R* INVALID SORT CONTROL FIELD ON CALL GSORT/GMERG •
Program: Last program entered.
ECODE: Sort control field sequence.
Reason:
a. Word 1 of the sort control field is out of range (-6 to +6) (System/360 only).
b. Boundary aligment of displacement is invalid for type of sort.
c. The sort field extends beyond the length of the record.
d. The number of elements specified is not a positive integer.

• 153 *R* INSUFFICIENT FILE SPACE TO EXECUTE GMERG FUNCTION •
Program: Last program entered.
ECODE: Merge file number.
Reason: The required space for the merged file is not available (System/360 only).

• 154 *R* INSUFFICIENT WORK AREA IN MANAGED AREA SAVE FILE FOR GSORT FUNCTION •
Program: Program detecting the error.
ECODE: File number.
Reason: Self-explanatory (System/360 only).

• 155 *R* MERGE FILE OUT OF SEQUENCE ON GMERG •
Program: Program detecting the error
ECODE: File number.
Reason: Self-explanatory (System/360 only).

• 156 *R* UNOPENED FILE CONTROL BLOCK ON CALL GSORT/GMERG •
Program: Program calling GSORG/GMERG
ECODE: File number.
Reason: The file control block specified is found not to be properly opened (System/360 only).

• 171 *R* INVALID SAVED STATEMENT EXECUTION FILE •
Program: PSTSV
ECODE: File number.
Reason: The header of the indicated file is found not to be valid for a statement save file.

• 172 *R* STATEMENT TO EXECUTE NOT IN SAVE FILE •
Program: PSTSV
ECODE: The number of the statement to be executed from the save file.
Reason: A statement has been indicated for retrieval from the statement save file but cannot be found.

• 173 *R* PROGRAM ERROR IN SAVED STATEMENT RETRIEVAL •
Program: PSTSV
ECODE: The invalid value causing the error.
Reason: The saved statement file has been destroyed or overwritten.

• 180 *R* INVALID LITERAL FILE •
Program: PDIAG or PLITL
ECODE: The file number.
Reason: A file defined for literal processing cannot properly be opened by GDATA.

13.8.0 PSCAN DIAGNOSTICS

The following diagnostics are generated as a result of errors detected by PSCAN while processing the phrases and language definition file (PFILE).

• 201 *R* PHRASE SKIPPED •
ECODE: Unused.
Reason: PSCAN has caused the statement to be bypassed because of an error in a preceding command upon which this command is dependent.
Action: The next command is processed.

• 210 *R* LEVEL 0 PHRASE NOT ENCOUNTERED •
ECODE: Cursor.
Reason: A level 0 phrase was not encountered following the invoking of PLAN.
Action: Statements are skipped until a level 0 phrase is encountered.

• 220 *R* LEVEL 1 PHRASE NOT ENCOUNTERED •
ECODE: Cursor
Reason: The first recognizable command in a job stack depends logically on a statement that was not found. The preceding statement(s) may have resulted in a code 221 diagnostic.
Action: Statements are skipped until a level 1 phrase is encountered.

• 221 *R* UNDEFINED PHRASE •
ECODE: Cursor.
Reason: The command cannot be recognized in total or in part as a phrase defined in the systems dictionary. The statement scan is abandoned.
Action: The scan of this command is terminated.

• 222 *R* STATEMENT OVER 450 CHARACTERS •
ECODE: Cursor.
Reason: A semicolon may be mispunched or missing.
Action: Statement scan is terminated. Only that portion of the command read as the last record is printed as command text when long-form diagnostics are used. The position will be identified as positions 001-100.

• 223 *R* PLAN WORD FALSE •
ECODE: A subscript indicating the particular communication array location that was tested for not FALSE.
Reason: The tested location was found to be FALSE.
Action: Level error recovery and skipping is initiated.

• 224 *R* PLAN WORD NOT REAL •
ECODE: A subscript indicating the communication array location that was found to be TRUE or FALSE.
Reason: A word required to be real is TRUE or FALSE.
Action: Level error recovery and skipping is initiated.

• 225 *R* PLAN WORD NOT TRUE •
ECODE: A subscript indicating the communication array location that was found to be FALSE or REAL.
Reason: A word required to be TRUE is FALSE or REAL.
Action: Level error recovery and skipping is initiated.

• 226 *R* PLAN WORD NOT FALSE •
ECODE: A subscript representing the communication array that is found to be TRUE or REAL.
Reason: A word required to be FALSE is found to be TRUE or REAL.
Action: Level error recovery and skipping is initiated.

• 227 *R* UNDEFINED SYMBOL IN INPUT STREAM •
ECODE: A cursor pointing to the end of the symbol in question.
Reason: A symbolic data name has been misspelled, or a comma was omitted after the command in a statement. No symbol table entry can be found for the word in this statement or in any statement upon which this statement is dependent. Failure to terminate a command with a semicolon results in the next command being interpreted as data for the command that precedes it.
Action: The command is not executed, but the scan is completed.

• 228 *R* UNDEFINED SYMBOL IN EXECUTION-DEFINED SYMBOL EXPRESSION •
ECODE: The sequence number of the expression in the phrase definition.
Reason: A symbolic subscript expression contains an undefined symbol.
Action: The scan is completed and the level error recovery is initiated.

• 229 *R* UNDEFINED SYMBOL IN PHRASE-DEFINED EXPRESSION •
ECODE: The sequence number of the expression in the phrase definition.
Reason: A symbol used in a phrase-defined expression is found to be undefined.
Action: The scan is completed and the

level error recovery is initiated.

• 230 *R* OVER 8 VERBS IN INPUT STATEMENT •
ECODE: A pointer to the end of the ninth verb.
Reason: A command may not contain more than eight verb phrases and an object phrase.
Action: Statement scan is terminated.

• 231 *R* DITTO WORD IN COMMON NOT ALPHA •
ECODE: A pointer to the communication array word that is to be substituted in a command for a ditto mark.
Reason: Using the ditto character in a command depends on the definition of the preceding command. The word that is to be substituted is not alphabetic.
Action: The scan is terminated and level error recovery is initiated.

• 232 *R* EXECUTION-DEFINED SYMBOL SUBSCRIPT NOT POSITIVE •
ECODE: The sequence of the subscript expression within the phrase definition.
Reason: Evaluation of a symbolic subscript within the phrase definition has yielded a negative or zero result indicating an invalid communication array location.
Action: The scan is completed and level error recovery is initiated.

• 233 *R* EXECUTION-DEFINED SYMBOL SUBSCRIPT GREATER THAN 8176 OR 511 WITH P-VALUE •
ECODE: A number indicating the sequence of the symbolic subscript in the phrase definition.
Reason: The symbolic subscript expression, when evaluated, is found to be too large.
Action: The scan is completed and the level error recovery is initiated.

• 234 *R* INSUFFICIENT ROOM IN MANAGED ARRAY SAVE FILE •
ECODE: Number of additional words needed in PDATA file.
Reason: The file specified for saving the managed communication array is too small to allow saving of the context of the current managed array.
Action: The scan is completed and the level error recovery is initiated.

• 235 *R* MANAGED ARRAY DEFINITION TOO LARGE •
ECODE: The number of words in excess of the allowable size.
Reason: A communication array has been specified that cannot be accommodated by the current partition/machine size.
Action: The array is not saved or restored by PLAN data management, and the array is not initialized to FALSE at level 1 phrase time.

• 236 *R* INITIALIZATION VALUE SUBSCRIPT OUTSIDE OF COMMON •
ECODE: Value of subscript.
Reason: The CAP index for a default value is outside the current communication array.
Action: The value is not stored.

• 237 *R* DATA PLACEMENT FROM INPUT STREAM OUTSIDE OF COMMON •
ECODE: Input cursor.
Reason: The CAP index of an input value is outside the current communication array specification.
Action: The value is not written to the communication array.

• 239 *R* DATA PLACEMENT FROM PHRASE-DEFINED EXPRESSION OUTSIDE OF COMMON •
ECODE: Expression number.
Reason: The CAP index for storage of the results of an expression evaluation is outside the current communication array specification.
Action: The value is not written to the communication array.

• 240 *R* FIRST CHARACTER IN INPUT STREAM AFTER PHRASE NOT COMMA, COLON, OR SEMICOLON •
ECODE: A cursor to the unexpected character.
Reason: The character required to start/terminate data collection was not encountered.
Action: The scan is completed and the level error recovery is initiated.

• 241 *R* UNRECOGNIZABLE CHARACTER IN INPUT STREAM •
ECODE: A cursor to the unrecognizable character.
Reason: A character cannot be interrogated in this context. It may have resulted from an illegal multipunch.
Action: The scan is completed and the level error recovery is initiated.

• 242 *R* SEMICOLON IN LITERAL OR EMPTY LITERAL •
ECODE: A cursor pointing to the invalid semicolon.

Reason: Either the literal closure character is missing or a semicolon is present within the literal.
Action: The scan is completed and level error recovery is initiated.

• 243 *R* NUMBER OUTSIDE ALLOWABLE FLOATING-POINT RANGE •
ECODE: A cursor to the end of the offending constant.
Reason: A number larger than can be contained in a floating-point number has been encountered. Note that the limit is .17014117E39 on the 1130.
Action: The scan is completed and level error recovery is initiated.

• 244 *R* IMPLIED DO NOT FOLLOWED BY SINGLE VALUED CONSTANT •
ECODE: A pointer to the position processed when the error was detected.
Reason: A single logical or numeric value does not follow an implied DO definition.
Action: The scan is completed and level error recovery is initiated.

• 245 *R* OVER 1000 EXPRESSION GO-TO'S EXECUTED •
ECODE: A number indicating the sequence of the expression found to be in error or input cursor.
Reason: Only 1000 formula GO-TO's are allowed within any phrase. This limit has been exceeded.
Action: The scan is completed and level error recovery is initiated.

• 246 *R* CHECK-ENTRY SUBSCRIPT OUTSIDE OF COMMON •
ECODE: Subscript value.
Action: The indicated communication array location is not checked.
Reason: The CAP index requiring execution of a check is outside the current communication array specification.

• 247 *R* DATA RETRIEVAL OUTSIDE OF COMMON
Program: PSCAN
ECODE: A cursor to the input stream subscript.
Reason: An attempt has been made to access data outside the current communication array.
Action: A 1.0 is supplied for arithmetic calculations and 0.0 for relational calculations. The scan is completed and level recovery is initiated.

• 248 *R* DATA RETRIEVAL OUTSIDE OF COMMON IN EXECUTION-DEFINED SYMBOL EXPRESSION •
ECODE: The expression number.
Reason: An attempt has been made to access data outside the current communication array.
Action: A 1.0 is supplied for arithmetic calculations and 0.0 for relational calculations. The scan is completed and level recovery is initiated.

• 249 *R* DATA RETRIEVAL OUTSIDE OF COMMON IN PHRASE-DEFINED EXPRESSION •
ECODE: The expression number.
Reason: An attempt has been made to access data from a location outside the current communication array specification.
Action: A 1.0 is supplied for arithmetic calculations and 0.0 for relational calculations. The scan is completed and level recovery initiated.

• 255 *R* STATEMENT SAVE INVALID, PHRASE PUSHED FROM CHECK-ENTRY •
ECODE: CAP location being checked.
Reason: Implicit statement saving may not be combined with check entry pushed phrases.
Action: The statement is not saved; the PLAN error recovery is initiated, but the phrase is pushed.

• 263 *R* INVALID FORMAT IN INPUT STREAM EXPRESSION •
ECODE: A cursor to the offending position.
Reason: An input stream expression is found to contain improper syntax. Reasons for this diagnostic may be:
a. Arithmetic operators without an intervening constant or data name.
b. An arithmetic or logical operator immediately following a parenthesis.
c. An arithmetic or logical operator immediately followed by a right parenthesis.
d. Invalid characters.
Action: The scan is completed and level error recovery is initiated.

• 264 *R* INVALID FORMAT IN EXECUTION-DEFINED SYMBOL EXPRESSION •
ECODE: A number indicating the sequence of the expression in error.
Reason: A syntax error has been detected in the symbolic subscript defined at ADD PHRASE time. Reasons for this diagnostic may be:
a. Arithmetic operators without an intervening constant or data name.

b. An arithmetic or logical operator immediately following a parenthesis.
c. An arithmetic or logical operator immediately followed by a right parenthesis.
d. Invalid characters.

Action: The scan is completed and level error recovery is initiated.

• 265 *R* INVALID FORMAT IN PHRASE-DEFINED EXPRESSION •
ECODE: A number indicating the sequence of the expression in error.
Reason: A syntax error has been detected in the phrase definition of an expression. Reasons for this diagnostic may be:
a. Arithmetic operators without an intervening constant or data name.
b. An arithmetic or logical operator immediately following a parenthesis.
c. An arithmetic or logical operator immediately followed by a right parenthesis.
d. Invalid characters.
Action: The scan is completed and level error recovery is initiated.

• 266 *R* BCD LEFT PARENTHESIS USED IN INPUT STREAM LOGICAL EXPRESSION •
ECODE: A pointer to the erroneous parenthesis.
Reason: All logical expressions must be punched in EBCDIC code.
Action: The scan is completed and level error recovery is initiated.

• 268 *R* BCD LEFT PARENTHESIS USED IN PHRASE-DEFINED LOGICAL EXPRESSION •
ECODE: A number indicating the sequence number of the expression in error.
Reason: Logical expressions must be punched in EBCDIC code.
Action: The scan is completed and level error recovery is initiated.

• 269 *R INPUT STREAM EXPRESSION TOO COMPLICATED TO BE ANALYZED •
ECODE: A pointer to the position at which error was detected.
Reason: Too many levels of parenthesis have been encountered.
Action: The scan is completed and level error recovery is initiated.

• 270 *R* EXECUTION-DEFINED SYMBOL EXPRESSION TOO COMPLICATED TO BE ANALYZED •
ECODE: A number indicating the sequence of the expression found to be in error.
Reason: Too many levels of parenthesis have been encountered.
Action: The scan is completed and level error recovery is initiated.

• 271 *R* PHRASE-DEFINED EXPRESSION TOO COMPLICATED TO BE ANALYZED •
ECODE: A number indicating the sequence of the expression found to be in error.
Reason: Too many levels of parenthesis have been encountered.
Action: The scan is completed and level error recovery is initiated.

• 272 *R* INVALID FORMAT IN INPUT STREAM LITERAL RELATIONAL EXPRESSION •
ECODE: A pointer to the character processed when the error was discovered.
Reason: A syntax error in an alphabetic relational expression. This diagnostic may result from expressions of the nature:
a. 5="A"
b. A>"B"
c. B<"C"
Action: The scan is completed and level error recovery is initiated.

• 274 *R* INVALID FORMAT IN PHRASE-DEFINED LITERAL RELATIONAL EXPRESSION •
ECODE: A number indicating the sequence of the expression causing the error.
Reason: A syntax error in a phrase-defined relational. This diagnostic may result from expressions of the nature:
a. 5="A"
b. A>"B"
c. B<"C"
Action: The scan is completed and level error recovery is initiated.

• 275 *R* INVALID FORMAT IN INPUT STREAM SUBSCRIPT EXPRESSION •
ECODE: A pointer to the character processed when error was detected.
Reason: A syntax error in a symbolic subscript or a subscript expression evaluation yields a negative result. Reasons for this diagnostic may be:
a. Result of subscript expression is not positive.
b. A logical value was encountered during the evaluation
c. An Implied Do was encountered in the evaluation of a subscript expression.
Action: The scan is completed and level error recovery is initiated.

• 276 *R* INVALID FORMAT IN EXECUTION-DEFINED SYMBOL SUBSCRIPT EXPRESSION •
ECODE: A pointer to the character processed when the error was

detected.
Reason: A syntax error in symbol expression.
Action: The scan is completed and level error recovery is initiated.

• 277 *R* INVALID FORMAT IN PHRASE-DEFINED SUBSCRIPT EXPRESSION •
ECODE: A number indicating the sequence of the expression found to be in error.
Reason: A syntax error.
Action: The scan is completed and level error recovery is initiated.

• 278 *R* UNBALANCED PARENTHESES IN INPUT STREAM EXPRESSION •
ECODE: A pointer to the position at which the error was detected.
Reason: An unequal number of right and left parentheses are found in an expression.
Action: The scan is completed and level error recovery is initiated.

• 279 *R* UNBALANCED PARENTHESES IN EXECUTION-DEFINED SYMBOL EXPRESSION •
ECODE: A pointer to the position at which the error was detected.
Reason: An unequal number of left and right parentheses are found in an expression.
Action: The scan is completed and level error recovery is initiated.

• 280 *R* UNBALANCED PARENTHESES IN PHRASE-DEFINED EXPRESSION •
ECODE: A number indicating the sequence of the expression found to be in error.
Reason: An unequal number of left and right parentheses have been found, or a right parenthesis has been found with no preceding matched left parenthesis.
Action: The scan is completed and level error recovery is initiated.

• 281 *R* INVALID FORMAT IN INPUT STREAM CONDITIONAL EXPRESSION •
ECODE: A pointer to the position at which the error was detected.
Reason: A syntax error. Reasons for this diagnostic may be:
a ? or ! not followed by #, =, :, or $
Action: The scan is completed and level error recovery is initiated.

• 283 *R* INVALID FORMAT IN PHRASE-DEFINED CONDITIONAL EXPRESSION •
ECODE: A number indicating the sequence of the expression found to be in error.
Reason: A syntax error. Reasons for this diagnostic may be:
a ? or ! not followed by #, =, :, or $
Action: The scan is completed and level error recovery is initiated.

• 284 *R* INVALID FORMAT IN INPUT STREAM RELATIONAL EXPRESSION •
ECODE: A pointer to the position at which the error was detected.
Reason: A snytax error. Reasons for this diagnostic may be:
a. Unbalanced parenthesis
b. Invalid characters
Action: The scan is completed and level error revovery is initiated.

• 286 *R* INVALID FORMAT IN PHRASE-DEFINED RELATIONAL EXPRESSION •
ECODE: A number giving the sequence of the expression found to be in error.
Reason: A syntax error. Reasons for this diagnostic may be:
a. Unbalanced parenthesis
b. Invalid characters
Action: The scan is completed and level error recovery is initiated.

• 287 *R* INVALID END TO AN INPUT STREAM EXPRESSION •
ECODE: Input cursor.
Reason: An expression must end with a semicolon or comma.
Action: The scan is completed and level error recovery is initiated.

• 289 *R* INVALID END TO A PHRASE-DEFINED EXPRESSION •
ECODE: Sequence number of the expression in error.
Reason: An expression must end with a semicolon or comma.
Action: The scan is completed and level error recovery is initiated.

• 290 *R* LOGICAL EOF ENCOUNTERED IN PSCAN INPUT •
ECODE: Undefined.
Reason: A logical EOF has been set by a PSCAN CALL PLINP operation.
Action: The scan is completed and level error recovery is initiated.

• 291 *R* INVALID END OF PLAN JOB •
ECODE: Undefined.
Reason: (1130 only) A monitor control record has been processed. The record will not be processed by 1130 monitor.
Action: Monitor continues processing at the next record.

• 299 *R or C* ******************** •
ECODE: A pointer to the communication array upon which an unsuccessful test was made.
Reason: The text for this diagnostic is normally user-defined text from a phrase-defined check entry. If the asterisks are provided,

an error has been detected in the defined literal.
Action: The phrase is terminated.

13.9.0 1130 ONLY DIAGNOSTICS

The following messages are generated from 1130 PLAN during the initialization phase.

• 700 *C* PFILE FOUND ON PACK xxxx •
Program: PLAN
ECODE: None produced.
Reason: This message is produced every time that PLAN is executed; it lists the cartridge identification of the pack on which the language dictionary was found.
Action: Processing continues.

• 701 *E* XXXXX NOT FOUND IN LET OR FLET •
Program: PLAN
Reason: XXXXX (which may be PSCAN, PERRS, PFILE, or PDQZ0) is not in the library.

• 702 *E* PFILE IS TOO SMALL •

Program: PLAN
Reason: The PLAN dictionary is too small. It must contain a minimum of 17 sectors on 1130 PLAN or 14 sectors on OS or DOS PLAN.
Action: PLAN execution is inhibited.

• 703 *C* PFILE INITIALIZED ON PACK XXXX •
Program: PLAN
Reason: The PLAN dictionary was found to be uninitialized and has been initialized.
Action: Normal processing continues.

• 725 *O* MOUNT PACK xxxx ON LOG DR n PHY DR 0,1,2,3,4 •
Reason: A request has been made for a disk cartridge that is not available. (See "Pack Changing Procedures" under "DYNAMIC File Support", 8.6.0.)

• 799 *E* PLAN EXECUTION INHIBITED •
Program: PLAN
Reason: Given after message 702 or when PSCAN, PERRS, PFILE or PDQZ0 are not in LET/FLET.
Action: PLAN execution is inhibited.

13.10.0 DOS ONLY DIAGNOSTICS

The following messages are produced by DOS PLAN.

• 820 *E* DFJPFIL MISSING OR FORMATTED INCORRECTLY •
Reason: Language file is incorrectly defined.
Action: PLAN execution is inhibited.

• 821 *E* XXXXXXXX NOT FOUND IN CORE IMAGE LIBRARY •
Reason: DFJPSCAN, DFJPHRAS, or DFJPERRS not found in program library.
Action: PLAN execution is inhibited.

• 822 *E* XXXXXXXX IS AN INVALID PLAN RUN CONTROL CARD •
Reason: The xxxxxxxx field represents the first eight characters of the invalid control card.
Action: PLAN execution is inhibited.

• 890 *R* UNCORRECTABLE INPUT/OUTPUT ERROR •
Program: Last entered.
ECODE: DOS logical unit.
Action: Current statement execution is aborted and level error recovery is invoked.

• 899 *E* PLAN EXECUTION INHIBITED •
Reason: Provided by the preceding diagnostic.
Action: PLAN execution is inhibited.

13.11.0 OS ONLY DIAGNOSTICS

The following messages are generated from the DD card edit performed by OS/360 PLAN. The message form is DDNAME, TEXT.

• 901 *E* XXXXXXXX NOT FOUND IN THE PLANLIB PDS •
Program: PLAN
Reason: The named module could not be loaded by the PLAN system. The modules are DFJPERRS, DFJPSCAN, or DFJRETN.
Action: PLAN execution is inhibited.

• 902 *E* DDNAME, DOES NOT SPECIFY A DIRECT ACCESS DEVICE •
Program: PLAN
Reason: The unit parameter of the specified DD card is incorrect.
Action: PLAN execution is inhibited.

• 903 *E* DDNAME, DATA SET DOES NOT EXIST •
Program: PLAN
Reason: The data set named in the DD card does not exist on the specified volume.
Action: PLAN execution is inhibited.

• 904 *E* DDNAME, INVALID BLKSIZE SPECIFICATION •

Program:	PLAN
Reason:	The specified BLKSIZE parameter is either too large for the unit specified or not a multiple of LRECL.
Action:	PLAN execution is inhibited.

• 905 *E* DDNAME, INVALID DSCB SPECIFICATIONS •

Program:	PLAN
Reason:	The data set named in the specified DD card: a. Has a partitioned data set format b. Has RECFM other than F or FB c. contains keys d. was never closed
Action:	PLAN execution is inhibited.

• 906 *E* DDNAME, INVALID SPACE ALLOCATION •

Program:	PLAN
Reason:	The space parameter in the named DD card does not specify TRK or CYL allocations.
Action:	PLAN execution is inhibited.

• 907 *E* DDNAME, I/O ERROR WHILE FORMATTING •

Program:	PLAN
Reason:	Input/Output error.
Action:	PLAN execution is inhibited.

• 908 *E* DDNAME, IS AN INVALID PLAN DD CARD •

Program:	PLAN
Reason:	The numeric specification on a PLINPxxx, PLOUTxxx, PLANFILx, or PLFSxxxx DD card is invalid.
Action:	PLAN execution is inhibited.

• 909 *E* DDNAME, DATA SET INITIALIZED INCORRECTLY •

Program:	PLAN
Reason:	A PLANFILX, PLSYSTAB, or PLNUMTAB was specified with DISP=OLD, and is not formatted correctly.
Action:	PLAN execution is inhibited.

• 910 *E* DDNAME, INSUFFICIENT FILE SIZE •

Program:	PLAN
Reason:	PLSYSTAB or PLANFILX is not allocated sufficient space for correct execution.
Action:	PLAN excution is inhibited.

• 911 *E* DDNAME, NOT DEFINED IN A DD CARD •

Program:	PLAN
Reason:	PLANFILO, PLSYSTAB, or PLANLIB DD cards are missing.
Action:	PLAN execution is inhibited.

The following messages are generated from OS/360 PLAN during the initialization phase.

• 922 *E* XXXXXXX PARAMETER OR OPERAND IS INVALID •

Program:	PLAN
Reason:	The named parameter in the EXEC control card is invalid.
Action:	PLAN execution is inhibited.

• 940 *R* DDNAME I/O ERROR •

Program:	Current program in control.
Action:	Phrase abort.

• 941 *R* XXXXXXXXXXXXXXXX •

Program:	Current program in control.
Reason:	A program interrupt occurred in a problem program. The diagnostic message is the program interrupt.
Action:	PLAN level error recovery is initiated.

• 942 *R* INSUFFICIENT PROGRAM AREA FOR PLAN FUNCTION •

Program:	PLAN
Reason:	The area allocated for the program area is too small.
Action:	Phrase abort.

• 999 *E* PLAN EXECUTION INHIBITED •

Program:	PLAN
Reason:	This action results if any of the above 900-series error conditions listed occur.
Action:	PLAN execution is inhibited.

PLAN will ABEND during PLAN initialization with the following user codes:

• ABEND USER CODE 100 •

Program:	PLAN
Reason:	Missing or invalid PLINP/PLOUT DD card.
Action:	PLAN execution is inhibited.

• ABEND USER CODE 101 •

Program:	PLAN
Reason:	Unable to load one of the following PLAN modules: DFJLODER, DFJTRACE
Action:	PLAN execution is inhibited.

• ABEND USER CODE 102 •

Program:	PLAN
Reason:	No DD card supplied.
Action:	PLAN execution is inhibited.

• ABEND USER CODE 103 •

Program:	PLAN
Reason:	Insufficient core storage to initialize PLAN.
Action:	PLAN execution is inhibited.

14.0.0 APPENDIX G: COMPATIBILITY CONSIDERATIONS

The main body of this manual defines a specification for a PLAN system that operates compatibly across the 1130, DOS, and OS versions (except as specifically noted). Special support provided in each of these systems is detailed in Appendices A, B and C.

Compatibility across all three versions of PLAN is provided as long as the compatible PLAN specifications are adhered to and as long as program modules are written in a language for which a compiler is provided on all three monitor/operating systems. The only language meeting such a requirement is ASA BASIC FORTRAN IV. Statements included in BASIC FORTRAN are detailed below:

ASSIGNMENT STATEMENTS
  Variable = arithmetic expression

FORMAT STATEMENT
  Statement-number FORMAT (format-specification)
  (Note: Because of difference in parameter specification, care should be exercised here if compatibility is required. PLAN Sequential I/O should be examined here for this type of support.)

CONTROL STATEMENTS
  CALL subroutine-name [(argument[,argument]...)]
  CONTINUE
  DO statement-number control-variable = initial-value, test-value [,increment]
  END
  GO TO statement-number
  GO TO (statement-number, statement-number [,statement-number]...)variable
  IF (arithmetic-expresion) statement-number, statement-number, statement-number
  PAUSE [one-to-four decimal digits]
  RETURN
  STOP [one-to-four decimal digits]
    (Note: The appendices contain specific restrictions on the use of the STOP statement.)
  READ (data-set-number, format-statement-number) [list]
    (Note: See comment under FORMAT and special comment in Appendix A.)
  WRITE (data-set-number, format-statement-number) [list]
    (Note: See comment under READ.)
  DEFINE FILE data-set-number (number-of records, maximum-record-size, U, associated-variable) [data-set-number..]...
    (Note: See comment under READ.)
  FIND (data-set-number' relative-position)
    (Note: See comment under READ.)
  READ (data-set-number' relative-position) [list]
    (Note: See comment above under READ.)
  WRITE (data-set-number' relative-position) [list]

SPECIFICATION STATEMENTS
  COMMON name [,name]
  COMMON array-declarator [,array-declarator]
  DIMENSION array-declarator [,array-declarator]
  EQUIVALENCE (name [,name]...) [(name[,name]...)]...
  EXTERNAL subprogram-name [,subprogram-name]
  INTEGER name [,name]
  REAL name [,name]

SUBPROGRAM STATEMENTS
  FUNCTION function-name (argument [,argument])
  function-name (argument [,argument]...)= arithmetic-expression
  INTEGER FUNCTION function-name
  REAL FUNCTION function-name
  SUBROUTINE subprogram-name [(argument [,argument]...)]

CONSTANT AND VARIABLE TYPES
  INTEGER
  REAL

15.0.0 APPENDIX H: SUMMARY OF SYSTEM LIMITS

This appendix provides a summary of limits and restrictions defined throughout this manual. The material is duplicated here merely as a convenient reference for the reader.

1. Maximum number of names in pop-up list:
   50

2. Maximum PLAN statement length:
   450 characters including semicolon (excluding all leading blanks)

3. Maximum number of symbols in an ADD PHRASE:
   255

4. Allowable formula numbers:

   a. ADD PHRASE: 0-1024
   b. Input stream 0-32,767

5. Maximum formula execution branches:
   1000

6. Maximum communication array size
   a. 32,767 except on
      (1) 8K 1130 with 8K PSCAN/PERRS: 510
      (2) 16K 1130 with 8K PSCAN/PERRS: 4606
      (3) 32K 1130 with 8K PSCAN/PERRS: 12,798
      (4) 16K 1130 with 16K PSCAN/PERRS: 1024
      (5) 32K 1130 with 16K PSCAN/PERRS: 9216

7. Maximum recognized alphabetic characters in word:
   3

8. Maximum words in a phrase:
   5

9. Maximum phrases in a statement:
   45 words = 1 object phrase + 8 verb phrases

10. Maximum CAP index in ADD PHRASE:
    16,368

11. Maximum CAP value resulting from evaluation of ADD PHRASE expression:
    a. 8176
    b. 512 if a scale factor is defined

12. Maximum range of Implied Do:
    65,368

13. Maximum CAP index for implicit check entry literal:
    16,384

14. Maximum DYNAMIC/PERMANENT logical drives:
    a. 5 on 1130
    b. 8 on OS/DOS

15. Allowable DYNAMIC/PERMANENT file numbers per logical drive: 1-255

16.0.0 APPENDIX I: PLAN CHARACTER SET

The following chart defines the characters that are required for PLAN language definition. Characters not shown in this chart may be entered as a character within literal text.

CHARACTER	CARD CODE	HEXADECIMAL	DECIMAL
blank		40	64
.	12-8-3	4B	75
<	12-8-4	4C	76
(	12-8-5	4D	77
+	12-8-6	4E	78
|	12-8-7	4F	79
&	12	50	80
!	11-8-2	5A	90
$	11-8-3	5B	91
*	11-8-4	5C	92
)	11-8-5	5D	93
;	11-8-6	5E	94
¬	11-8-7	5F	95
-	11	60	96
/	0-1	61	97
,	0-8-3	6B	107
%	0-8-4	6C	108
_	0-8-5	6D	109
>	0-8-6	6E	110
?	0-8-7	6F	111
:	8-2	7A	122
#	8-3	7B	123
@	8-4	7C	124
'	8-5	7D	125
=	8-6	7E	126
"	8-7	7F	127

CHARACTER	CARD CODE	HEXADECIMAL	DECIMAL
A	12-1	C1	193
B	12-2	C2	194
C	12-3	C3	195
D	12-4	C4	196
E	12-5	C5	197
F	12-6	C6	198
G	12-7	C7	199
H	12-8	C8	200
I	12-9	C9	201
J	11-1	D1	209
K	11-2	D2	210
L	11-3	D3	211
M	11-4	D4	212
N	11-5	D5	213
O	11-6	D6	214
P	11-7	D7	215
Q	11-8	D8	216
R	11-9	D9	217
S	0-2	E2	226
T	0-3	E3	227
U	0-4	E4	228
V	0-5	E5	229
W	0-6	E6	230
X	0-7	E7	231
Y	0-8	E8	232
Z	0-9	E9	233
0	0	F0	240
1	1	F1	241
2	2	F2	242
3	3	F3	243
4	4	F4	244
5	5	F5	245
6	6	F6	246
7	7	F7	247
8	8	F8	248
9	9	F9	249

17.0.0 APPENDIX J: SYSTEM REQUIREMENTS

MACHINE CONFIGURATIONS

IBM 1130 PLAN

Minimum requirement for PLAN operation
1131 Central Processing Unit Model 2B

2501 Card Reader Model A1 or A2
#3630 (1130/2501 Coupling)
#8042 Attachment
#3054 Expansion Adapter
AND
1442 Card Punch Model 5
#4449 Attachment

OR

1442 Card Read Punch Model 6 or 7
#4454 Attachment

Optional Support
1131 Central Processing Unit Models 2C, 2D, 3B, 3C, or 3D

1132 Printer
#3616
#3854 Expansion Adapter

1403 Printer Model 6 or 7
#4424 or #4425 Attachment
#1865 Channel Multiplexor

2310 Disk Storage Model B1 or B2 (one or two)
#3201, #3202, #3203, #3204 disk control

1133 Multiplex Control Enclosure (required for 1403 or 2310)
#1865 Multiplexor
#7490 Storage Access Channel

SYSTEM/360 DOS PLAN

Central Processing Unit
2025, 2030, 2040, 2050, 2065, or 2075 (32K bytes or larger). (An 8K supervisor is assumed.)
Floating-Point Arithmetic
One I/O Channel (either multiplexor or selector)
One Card Reader (1442, 2501, 2520, or 2540)
(one 2400 series tape drive may be substituted)
One Card Read Punch (1442, 2520, or 2540)
(one 2400 series tape drive may be substituted)
One Printer (1403, 1404, 1443)
(one 2400 series tape drive may be substituted)
One 1052 Printer-Keyboard
One 2311 Disk Storage Drive
One 2841 Storage Control

SYSTEM/360 OS PLAN

Central Processing Unit
2030, 2040, 2050, 2065, or 2075 that provides a problem partition of 32K bytes or larger for PCP-MFT and a region of 44K bytes or larger for MVT
Floating-Point Arithmetic
One Console
Any direct access device and control unit supported by OS/360 with a storage capacity equal to or greater than one 2311 Disk Storage Drive (in addition to that required by System/360)
One or more input devices supported by QSAM
One or more output devices supported by QSAM

ALL VERSIONS OF PLAN

The availability of an 029 printing keypunch will prove to be an asset in the preparation of PLAN language statements.

PROGRAMMING SYSTEM REQUIREMENTS

PLAN operates under three IBM monitor or operating systems. It is written primarily in Assembly Language. Some Utility functions are programmed in BASIC FORTRAN IV.

IBM 1130 PLAN operates under the 1130 Disk Monitor System, Version 2.

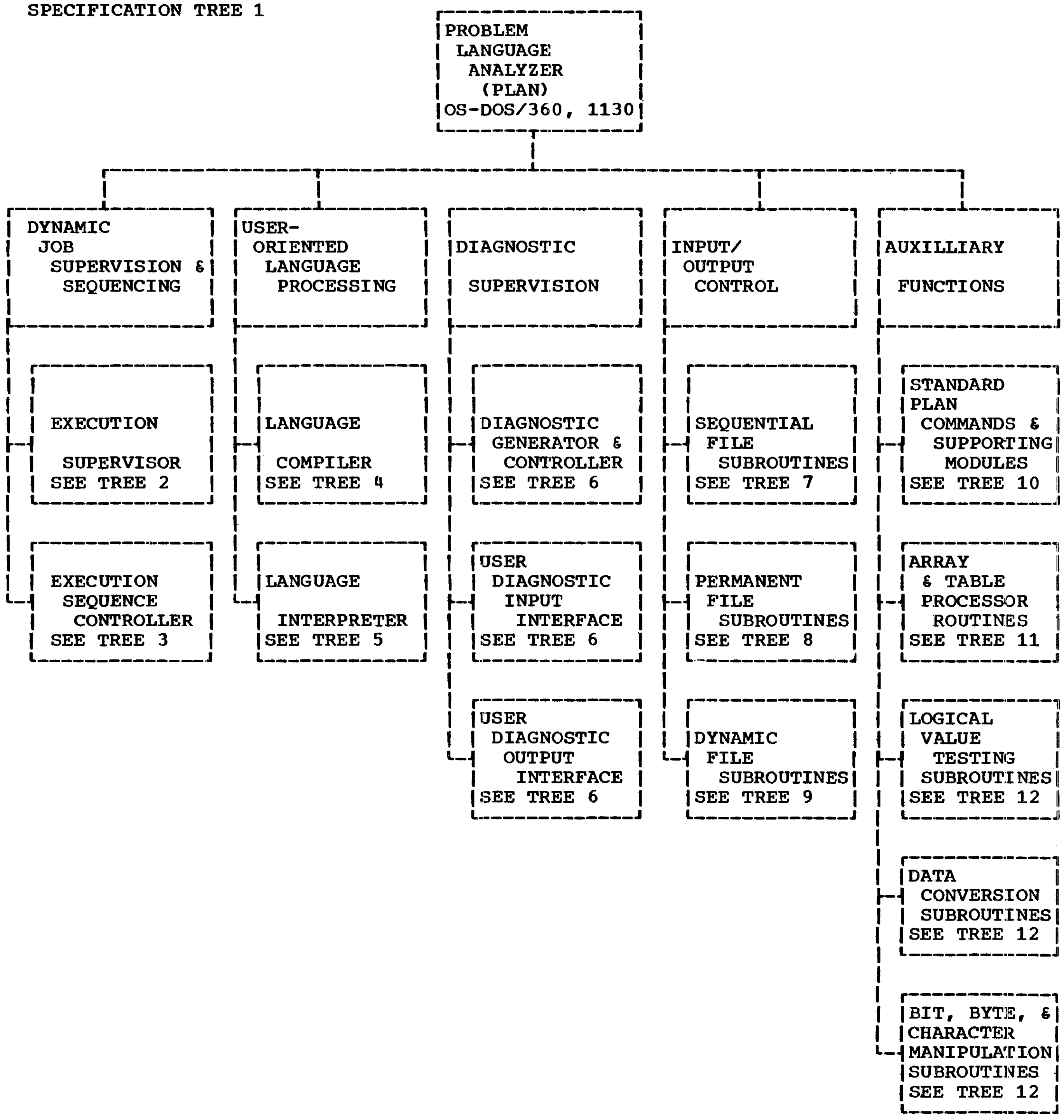
System/360 DOS PLAN operates under the IBM System/360 Disk Operating System.

System/360 OS PLAN operates under the IBM Operating System/360 using the MVT, PCP, or MFT options.

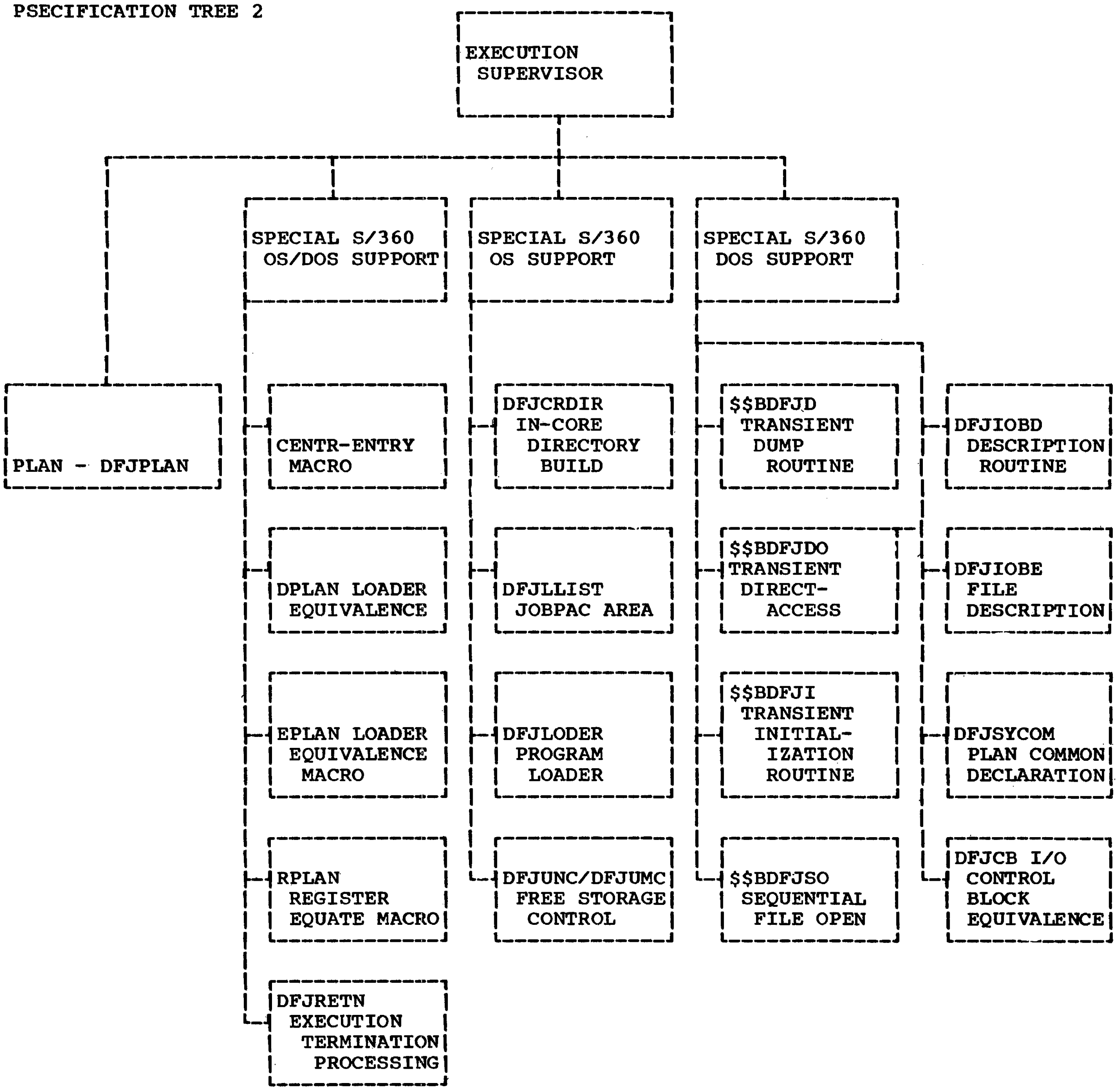
18.0.0 APPENDIX K: FUNCTIONAL ANALYSIS DIAGRAMS

The charts in this section represent the hierarchial structure of the PLAN system components. The first charts represent the functional areas and the subfunctional areas as described in the systems overview. Subsequent charts define the system components within the functional areas. The number on the lower line of the program component blocks define the major section where the function and use of the component is described.

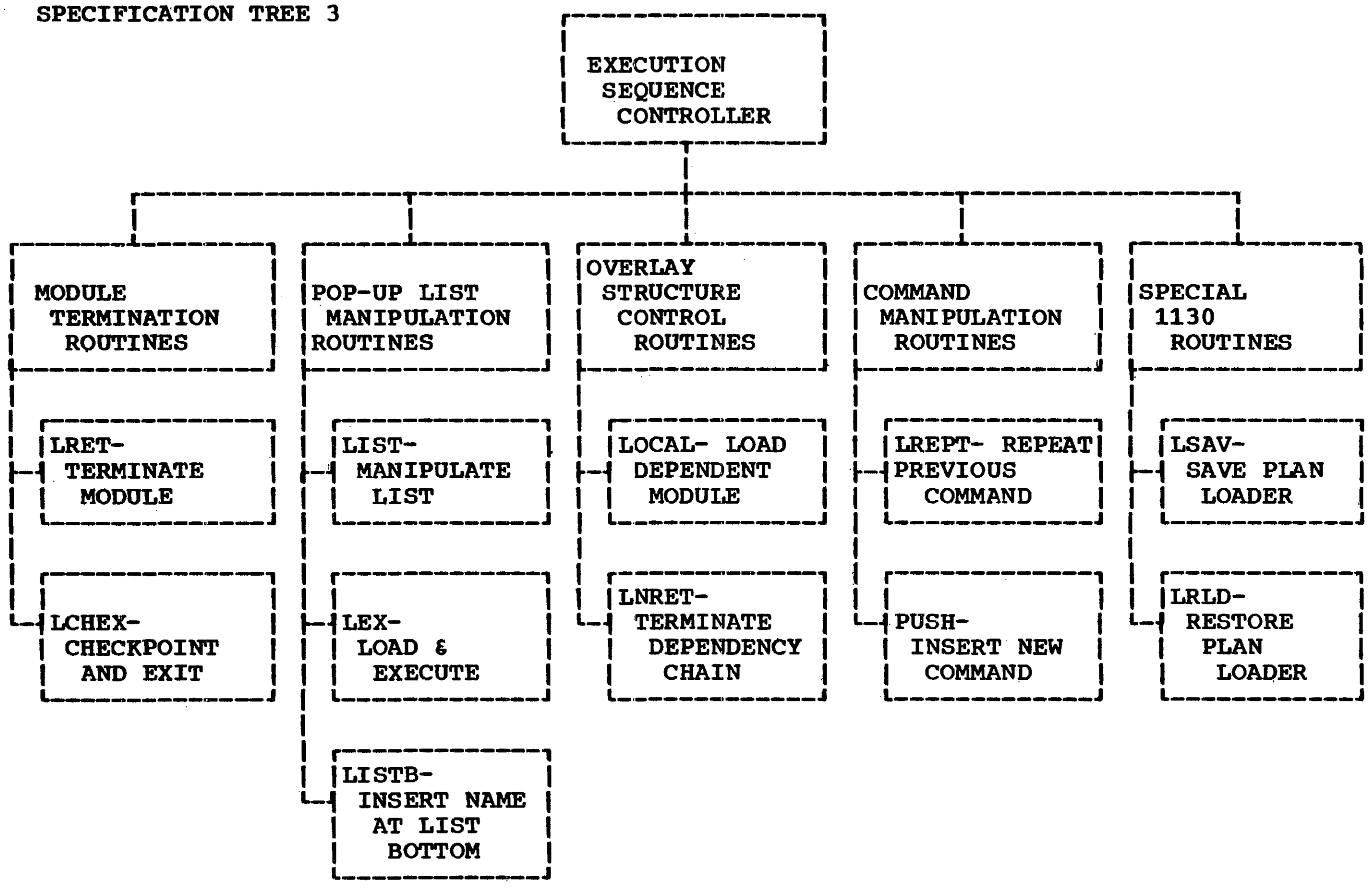
SPECIFICATION TREE 1


PROBLEM LANGUAGE ANALYZER (PLAN) OS-DOS/360, 1130
DYNAMIC JOB SUPERVISION & SEQUENCING
USER-ORIENTED LANGUAGE PROCESSING
DIAGNOSTIC SUPERVISION
INPUT/ OUTPUT CONTROL
AUXILLIARY FUNCTIONS
EXECUTION SUPERVISOR SEE TREE 2
EXECUTION SEQUENCE CONTROLLER SEE TREE 3
LANGUAGE COMPILER SEE TREE 4
LANGUAGE INTERPRETER SEE TREE 5
DIAGNOSTIC GENERATOR & CONTROLLER SEE TREE 6
USER DIAGNOSTIC INPUT INTERFACE SEE TREE 6
USER DIAGNOSTIC OUTPUT INTERFACE SEE TREE 6
SEQUENTIAL FILE SUBROUTINES SEE TREE 7
PERMANENT FILE SUBROUTINES SEE TREE 8
DYNAMIC FILE SUBROUTINES SEE TREE 9
STANDARD PLAN COMMANDS & SUPPORTING MODULES SEE TREE 10
ARRAY & TABLE PROCESSOR ROUTINES SEE TREE 11
LOGICAL VALUE TESTING SUBROUTINES SEE TREE 12
DATA CONVERSION SUBROUTINES SEE TREE 12
BIT, BYTE, & CHARACTER MANIPULATION SUBROUTINES SEE TREE 12

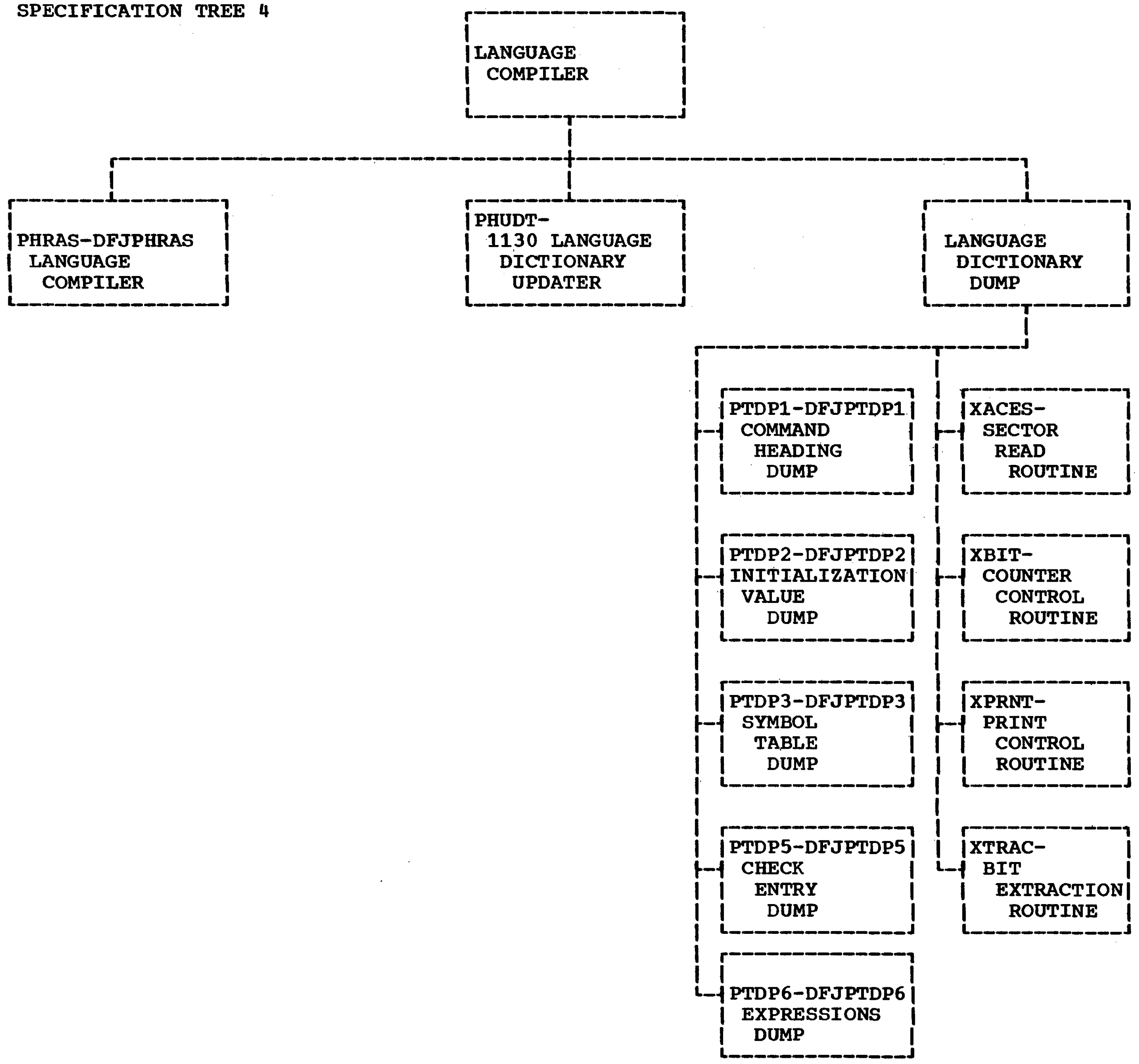
PSECIFICATION TREE 2


EXECUTION SUPERVISOR
SPECIAL S/360 OS/DOS SUPPORT
SPECIAL S/360 OS SUPPORT
SPECIAL S/360 DOS SUPPORT
PLAN - DFJPLAN
CENTR-ENTRY MACRO
DPLAN LOADER EQUIVALENCE
EPLAN LOADER EQUIVALENCE MACRO
RPLAN REGISTER EQUATE MACRO
DFJRETN EXECUTION TERMINATION PROCESSING
DFJCRDIR IN-CORE DIRECTORY BUILD
DFJLLIST JOBPAC AREA
DFJLODER PROGRAM LOADER
DFJUNC/DFJUMC FREE STORAGE CONTROL
$$BDFJD TRANSIENT DUMP ROUTINE
$$BDFJDO TRANSIENT DIRECT-ACCESS
$$BDFJI TRANSIENT INITIAL-IZATION ROUTINE
$$BDFJSO SEQUENTIAL FILE OPEN
DFJIOBD DESCRIPTION ROUTINE
DFJIOBE FILE DESCRIPTION
DFJSYCOM PLAN COMMON DECLARATION
DFJCB I/O CONTROL BLOCK EQUIVALENCE

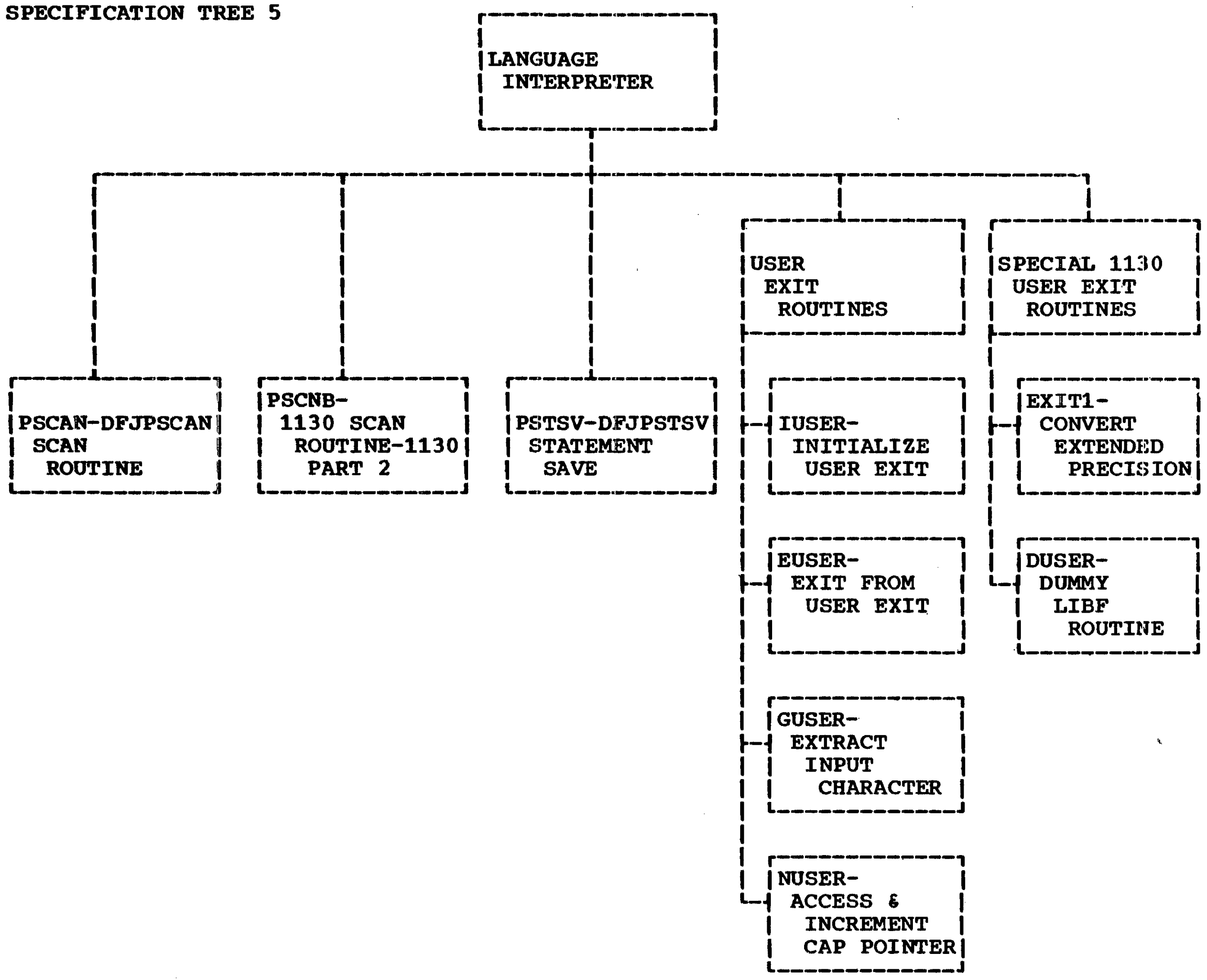
SPECIFICATION TREE 3


EXECUTION SEQUENCE CONTROLLER
MODULE TERMINATION ROUTINES
POP-UP LIST MANIPULATION ROUTINES
OVERLAY STRUCTURE CONTROL ROUTINES
COMMAND MANIPULATION ROUTINES
SPECIAL 1130 ROUTINES
LRET- TERMINATE MODULE
LCHEX- CHECKPOINT AND EXIT
LIST- MANIPULATE LIST
LEX- LOAD & EXECUTE
LISTB- INSERT NAME AT LIST BOTTOM
LOCAL- LOAD DEPENDENT MODULE
LNRET- TERMINATE DEPENDENCY CHAIN
LREPT- REPEAT PREVIOUS COMMAND
PUSH- INSERT NEW COMMAND
LSAV- SAVE PLAN LOADER
LRLD- RESTORE PLAN LOADER

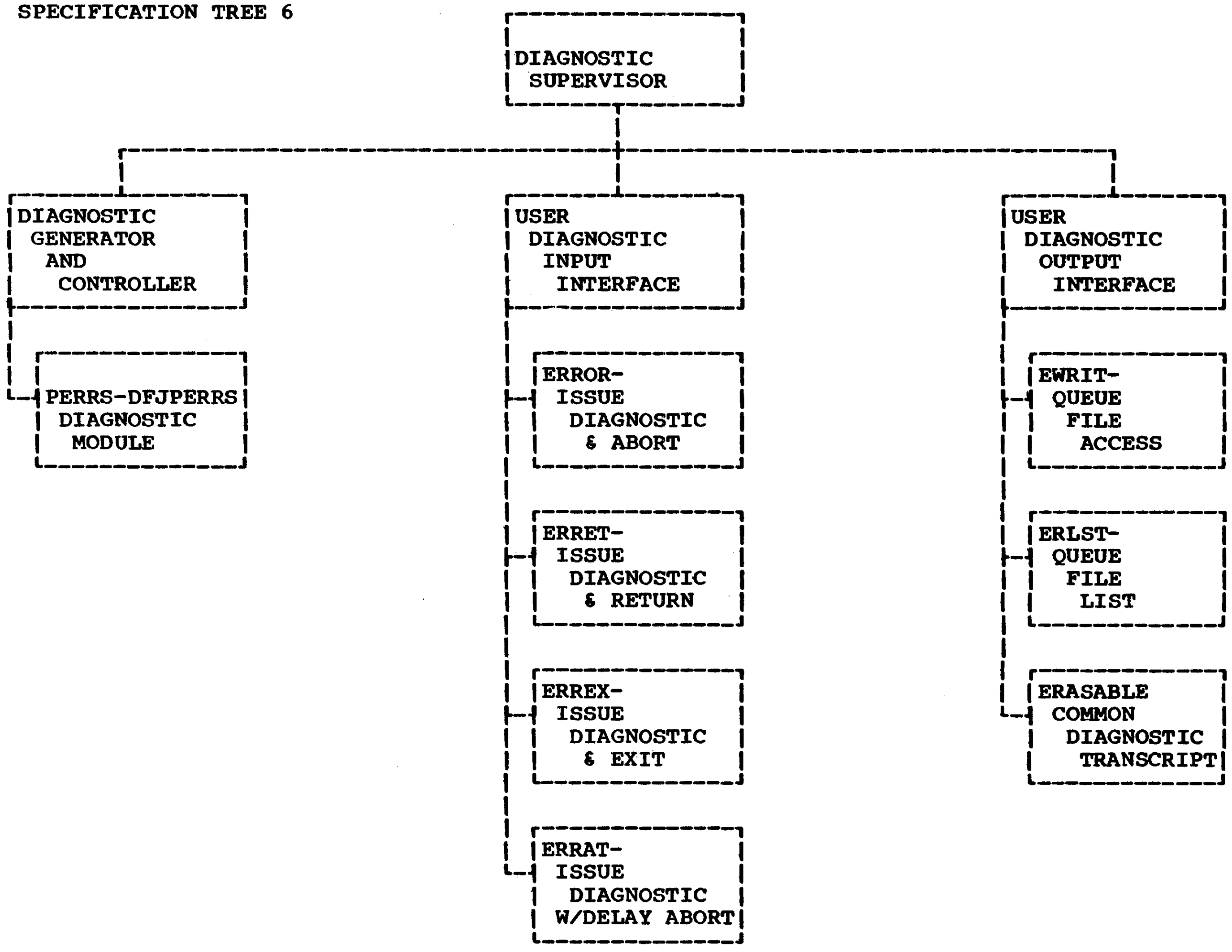
SPECIFICATION TREE 4


LANGUAGE COMPILER
PHRAS-DFJPHRAS LANGUAGE COMPILER
PHUDT- 1130 LANGUAGE DICTIONARY UPDATER
LANGUAGE DICTIONARY DUMP
PTDP1-DFJPTDP1 COMMAND HEADING DUMP
PTDP2-DFJPTDP2 INITIALIZATION VALUE DUMP
PTDP3-DFJPTDP3 SYMBOL TABLE DUMP
PTDP5-DFJPTDP5 CHECK ENTRY DUMP
PTDP6-DFJPTDP6 EXPRESSIONS DUMP
XACES- SECTOR READ ROUTINE
XBIT- COUNTER CONTROL ROUTINE
XPRNT- PRINT CONTROL ROUTINE
XTRAC- BIT EXTRACTION ROUTINE

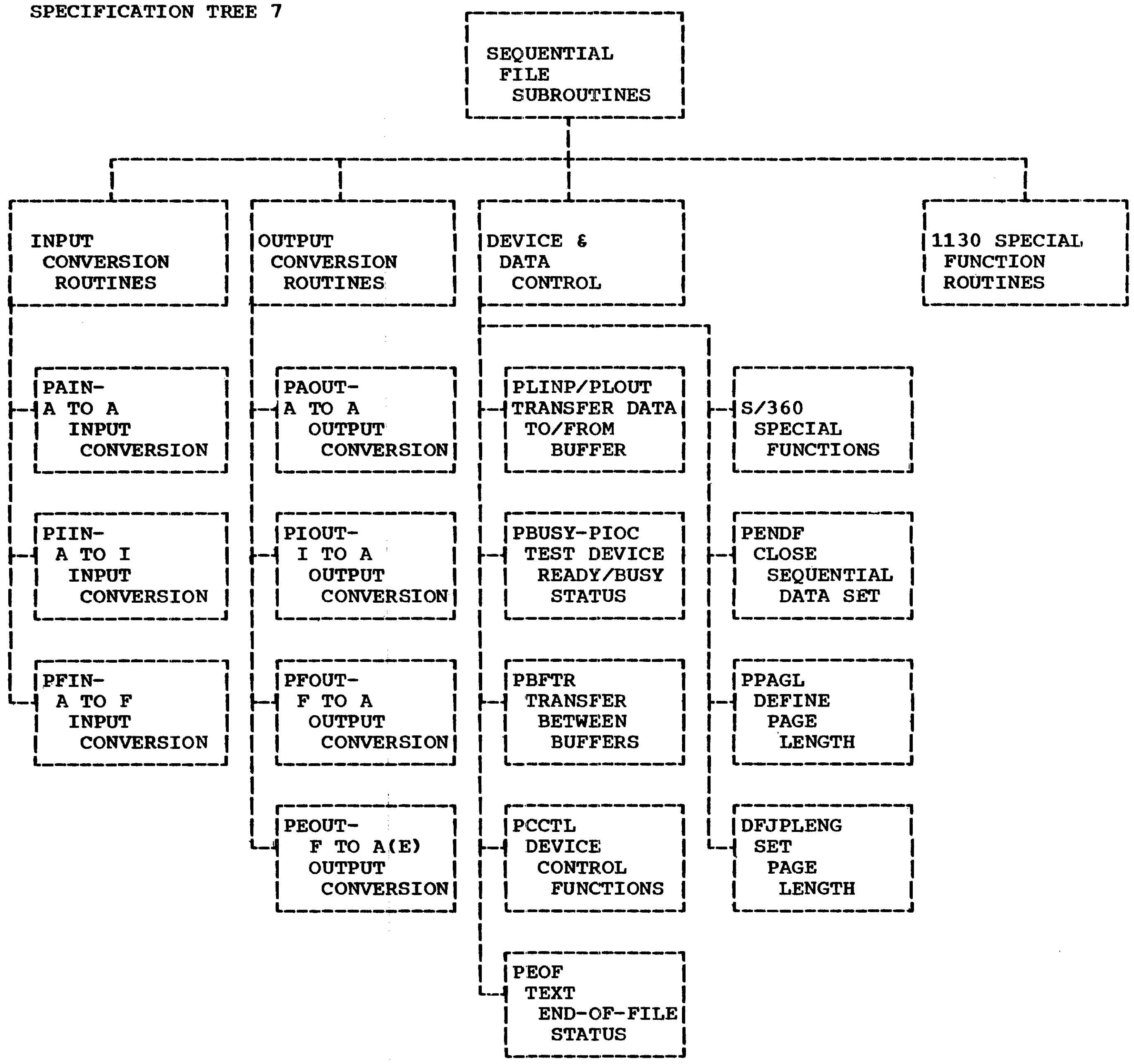
SPECIFICATION TREE 5


LANGUAGE INTERPRETER
PSCAN-DFJPSCAN SCAN ROUTINE
PSCNB- 1130 SCAN ROUTINE-1130 PART 2
PSTSV-DFJPSTSV STATEMENT SAVE
USER EXIT ROUTINES
SPECIAL 1130 USER EXIT ROUTINES
IUSER- INITIALIZE USER EXIT
EUSER- EXIT FROM USER EXIT
GUSER- EXTRACT INPUT CHARACTER
NUSER- ACCESS & INCREMENT CAP POINTER
EXIT1- CONVERT EXTENDED PRECISION
DUSER- DUMMY LIBF ROUTINE

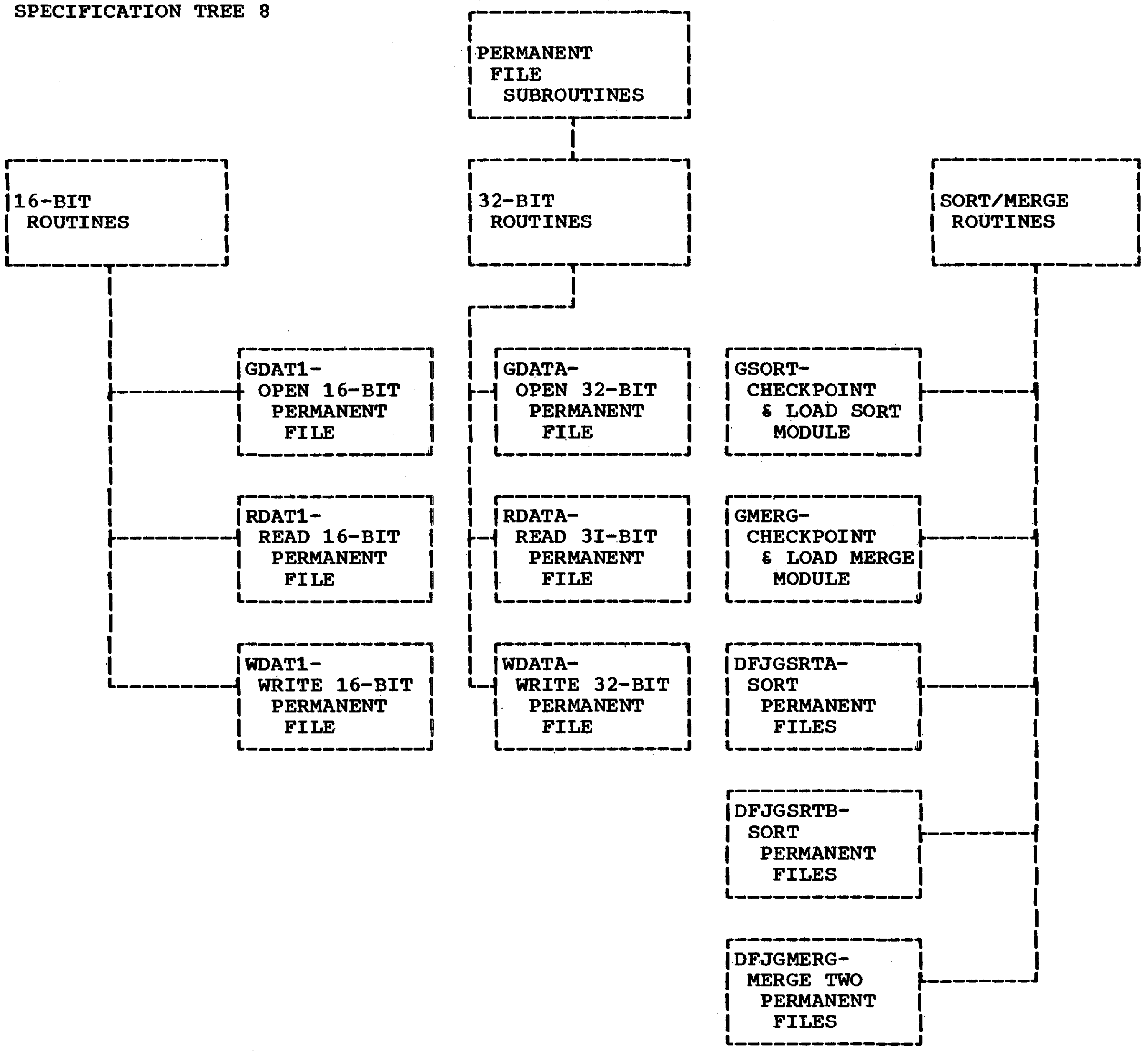
SPECIFICATION TREE 6


DIAGNOSTIC SUPERVISOR
DIAGNOSTIC GENERATOR AND CONTROLLER
PERRS-DFJPERRS DIAGNOSTIC MODULE
USER DIAGNOSTIC INPUT INTERFACE
ERROR- ISSUE DIAGNOSTIC & ABORT
ERRET- ISSUE DIAGNOSTIC & RETURN
ERREX- ISSUE DIAGNOSTIC & EXIT
ERRAT- ISSUE DIAGNOSTIC W/DELAY ABORT
USER DIAGNOSTIC OUTPUT INTERFACE
EWRIT- QUEUE FILE ACCESS
ERLST- QUEUE FILE LIST
ERASABLE COMMON DIAGNOSTIC TRANSCRIPT

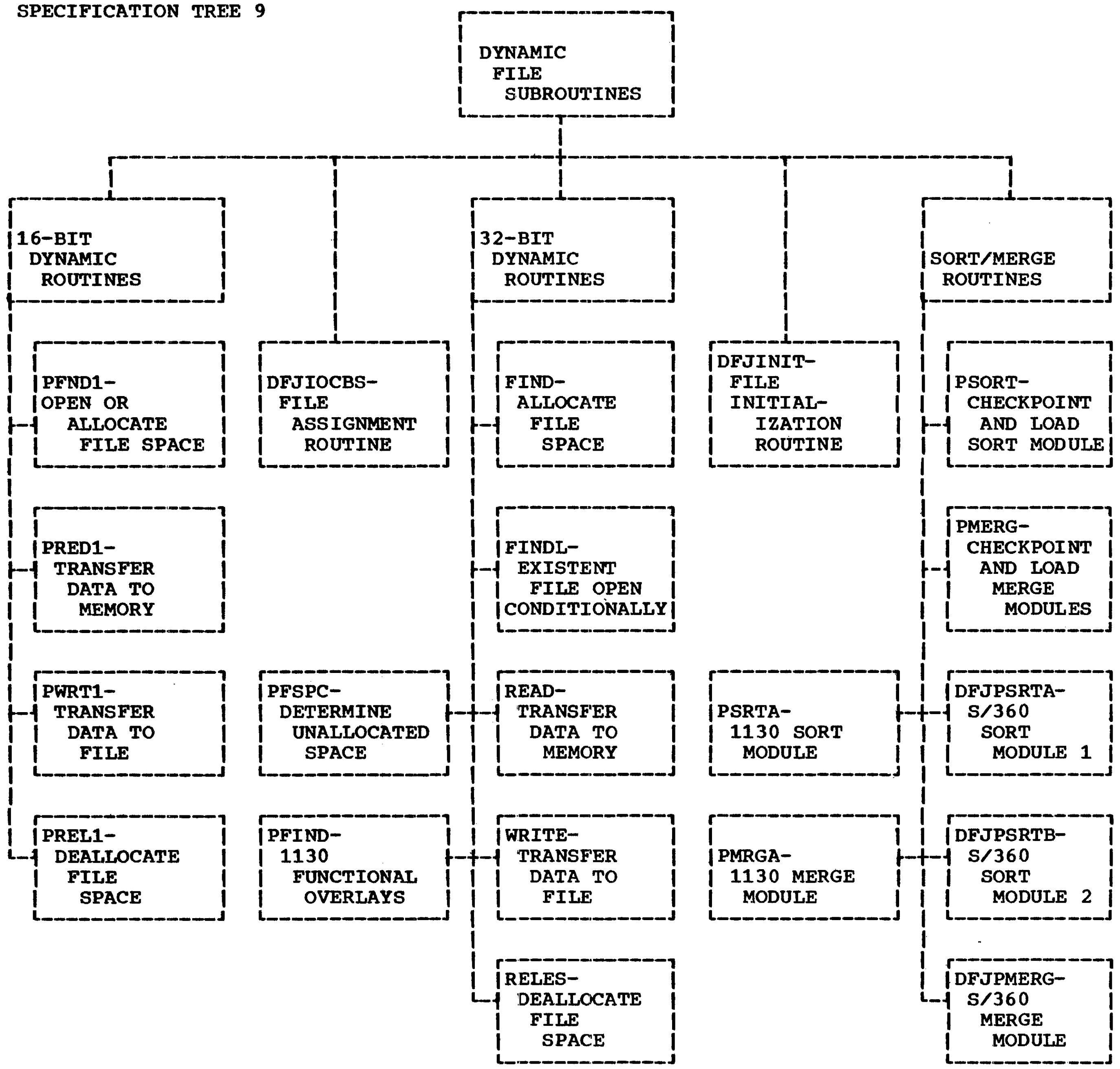
SPECIFICATION TREE 7


SEQUENTIAL FILE SUBROUTINES
INPUT CONVERSION ROUTINES
OUTPUT CONVERSION ROUTINES
DEVICE & DATA CONTROL
1130 SPECIAL FUNCTION ROUTINES
PAIN- A TO A INPUT CONVERSION
PIIN- A TO I INPUT CONVERSION
PFIN- A TO F INPUT CONVERSION
PAOUT- A TO A OUTPUT CONVERSION
PIOUT- I TO A OUTPUT CONVERSION
PFOUT- F TO A OUTPUT CONVERSION
PEOUT- F TO A(E) OUTPUT CONVERSION
PLINP/PLOUT TRANSFER DATA TO/FROM BUFFER
PBUSY-PIOC TEST DEVICE READY/BUSY STATUS
PBFTR TRANSFER BETWEEN BUFFERS
PCCTL DEVICE CONTROL FUNCTIONS
PEOF TEXT END-OF-FILE STATUS
S/360 SPECIAL FUNCTIONS
PENDF CLOSE SEQUENTIAL DATA SET
PPAGL DEFINE PAGE LENGTH
DFJPLENG SET PAGE LENGTH

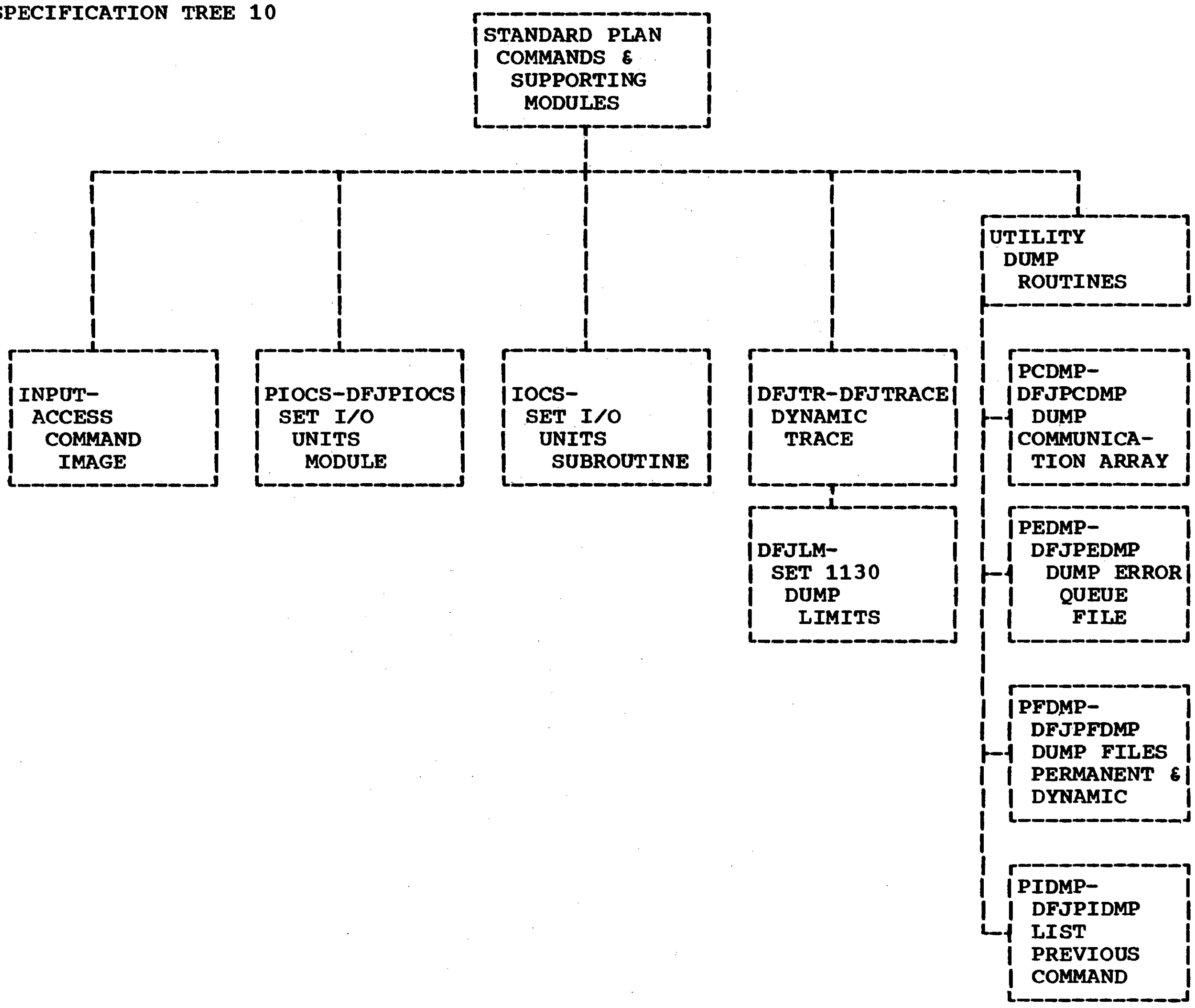
SPECIFICATION TREE 8


PERMANENT FILE SUBROUTINES
16-BIT ROUTINES
32-BIT ROUTINES
SORT/MERGE ROUTINES
GDAT1- OPEN 16-BIT PERMANENT FILE
RDAT1- READ 16-BIT PERMANENT FILE
WDAT1- WRITE 16-BIT PERMANENT FILE
GDATA- OPEN 32-BIT PERMANENT FILE
RDATA- READ 3I-BIT PERMANENT FILE
WDATA- WRITE 32-BIT PERMANENT FILE
GSORT- CHECKPOINT & LOAD SORT MODULE
GMERG- CHECKPOINT & LOAD MERGE MODULE
DFJGSRTA- SORT PERMANENT FILES
DFJGSRTB- SORT PERMANENT FILES
DFJGMERG- MERGE TWO PERMANENT FILES

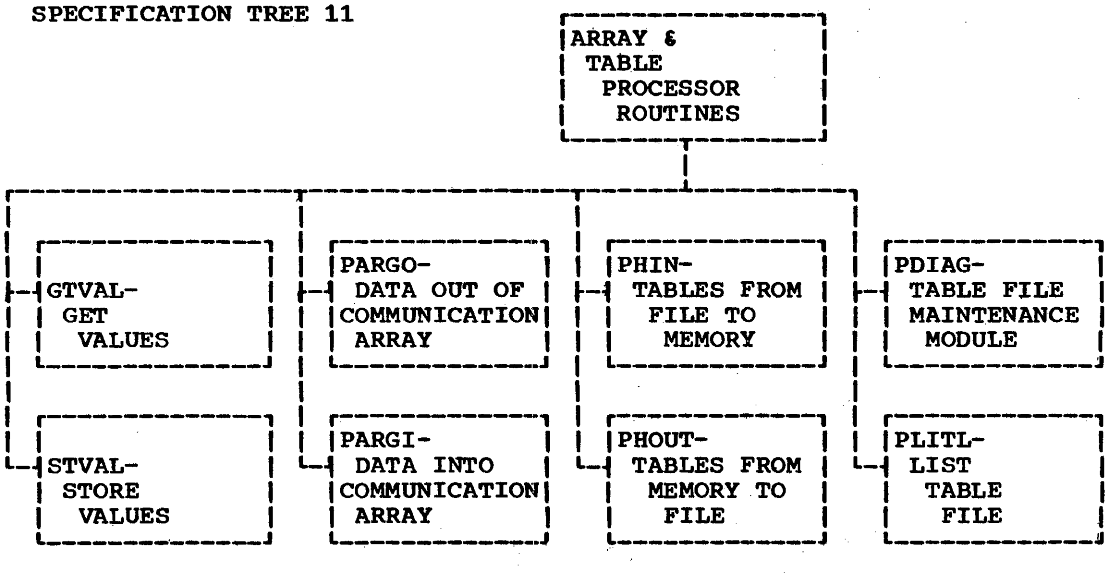
SPECIFICATION TREE 9


DYNAMIC FILE SUBROUTINES
16-BIT DYNAMIC ROUTINES
PFND1- OPEN OR ALLOCATE FILE SPACE
PRED1- TRANSFER DATA TO MEMORY
PWRT1- TRANSFER DATA TO FILE
PREL1- DEALLOCATE FILE SPACE
DFJIOCBS- FILE ASSIGNMENT ROUTINE
PFSPC- DETERMINE UNALLOCATED SPACE
PFIND- 1130 FUNCTIONAL OVERLAYS
32-BIT DYNAMIC ROUTINES
FIND- ALLOCATE FILE SPACE
FINDL- EXISTENT FILE OPEN CONDITIONALLY
READ- TRANSFER DATA TO MEMORY
WRITE- TRANSFER DATA TO FILE
RELES- DEALLOCATE FILE SPACE
DFJINIT- FILE INITIAL-IZATION ROUTINE
PSRTA- 1130 SORT MODULE
PMRGA- 1130 MERGE MODULE
SORT/MERGE ROUTINES
PSORT- CHECKPOINT AND LOAD SORT MODULE
PMERG- CHECKPOINT AND LOAD MERGE MODULES
DFJPSRTA- S/360 SORT MODULE 1
DFJPSRTB- S/360 SORT MODULE 2
DFJPMERG- S/360 MERGE MODULE

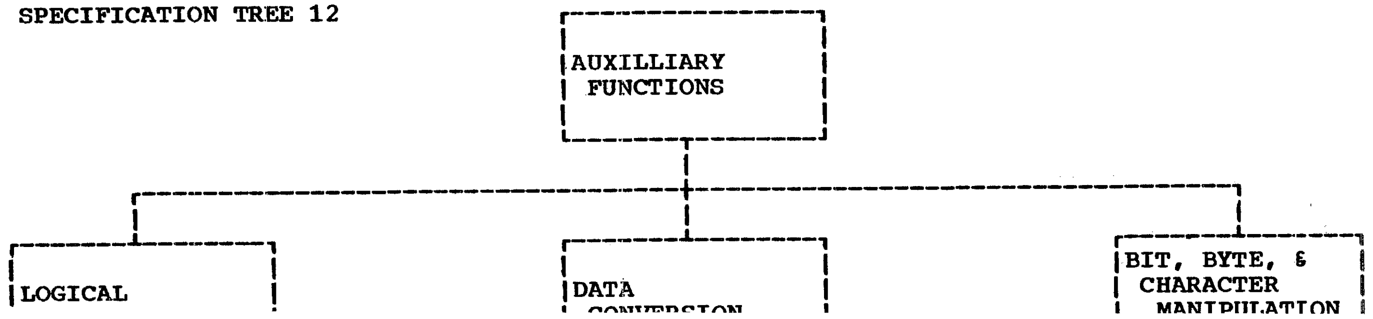
SPECIFICATION TREE 10


STANDARD PLAN COMMANDS & SUPPORTING MODULES
INPUT- ACCESS COMMAND IMAGE
PIOCS-DFJPIOCS SET I/O UNITS MODULE
IOCS- SET I/O UNITS SUBROUTINE
DFJTR-DFJTRACE DYNAMIC TRACE
DFJLM- SET 1130 DUMP LIMITS
UTILITY DUMP ROUTINES
PCDMP- DFJPCDMP DUMP COMMUNICATION ARRAY
PEDMP- DFJPEDMP DUMP ERROR QUEUE FILE
PFDMP- DFJPFDMP DUMP FILES PERMANENT & DYNAMIC
PIDMP- DFJPIDMP LIST PREVIOUS COMMAND

SPECIFICATION TREE 11


ARRAY & TABLE PROCESSOR ROUTINES
GTVAL- GET VALUES
STVAL- STORE VALUES
PARGO- DATA OUT OF COMMUNICATION ARRAY
PARGI- DATA INTO COMMUNICATION ARRAY
PHIN- TABLES FROM FILE TO MEMORY
PHOUT- TABLES FROM MEMORY TO FILE
PDIAG- TABLE FILE MAINTENANCE MODULE
PLITL- LIST TABLE FILE

SPECIFICATION TREE 12



COMMUNICATION ARRAY LAYOUT FOR: DATE:

DATA NAME		DATA NAME		DATA NAME		DATA NAME		DATA NAME		DATA NAME		DATA NAME		DATA NAME	
CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE
DEFAULT		DEFAULT		DEFAULT		DEFAULT		DEFAULT		DEFAULT		DEFAULT		DEFAULT	

The following definitions reflect context and usage within this manual and in other manuals related to the PLAN system.

Abort. An action resulting from detection by PLAN of an error. The current PLAN module is terminated, all programs in the pop-up list are skipped, and processing starts at the next equal- or higher-level command.

Arguments. A list indicating data that is to be made available to a new operation.

CAP. An abbreviation for communication array position. It represents the subscript to which a data name is equivalenced.

Check Entry. A definition within a phrase definition of tests to be made and actions to be made as a result of the test on data in the communication array.

Command. The phrase and modifying verbs indicating the indexed dictionary procedure to be followed.

COMMON. A FORTRAN or Assembler statement (source or control) that specifies variables and variable arrays that are to be placed in an area of core storage protected from overlay by program-loading operations.

Communication array. An array of data residing in COMMON used for transmittal of data from module to module and from the PLAN language interpreter to modules.

Constants. Data with a fixed numeric, logical, or alphmeric value.

Data. Input to or output from an operation.

Dictionary. An indexed data file containing the condensed syntactical text of a language. Indexed file of user's language.

Dimension. A FORTRAN source statement that defines the size of a data array for which storage is to be allocated.

DUMP COMMON. A PLAN command to print the contents of the communication array.

DUMP MANAGED. A PLAN command to print only the managed portion of the communication array.

DUMP NONMANAGED. A PLAN command to print only the nonmanaged portion of the communication array.

DUMP PERMANENT. A PLAN command to print the contents of a PLAN PERMANENT file.

DUMP DYNAMIC. A PLAN command to print the contents of a PLAN DYNAMIC file.

DUMP SWITCHES. A PLAN command to print the contents of the 15 PLAN switch words.

Elements. The syntactical portion of a statement required to define a data item.

EQUIVALENCE. A FORTRAN source statement that specifies variable names and array names that are to be set equal to the same location in memory. This statement affects only addressing; it does not effect any data transfer.

Evaluate. Determine a result on the basis of a series of procedural steps to be performed on data.

Extended precision. The size of a FORTRAN variable word greater than that defined by ASA standard FORTRAN.

FORMAT. A FORTRAN statement used to describe the physical organization of an input or output record. FORMAT is also used to describe the syntactical organization of data.

Interpreter. A program(s) that examines a language syntax and links to a series of programmed steps to effect a task solution without generating specific computer code for the task.

IOCS. A PLAN statement to allow the changing of input and output devices.

Language. A syntactically correct text stream that is input to a particular processor.

Loader. A computer program with the capability of retrieving a module from the program library and placing it in core storage in a form ready for execution.

Managed array. A portion of the communication array whose data content is maintained according to a language associated level (dependence) structure.

Module. A mainline program and its associated subroutines stored in the program library in a form appropriate for execution within PLAN.

Nonmanaged array. The portion of the communication array not in the managed array.

Operands. The data upon which operations are to be performed.

Operator. An arithmetic or logical symbol indicating an operation (add, and, or, multiply) that is to be performed on data.

Pack. The process of placing an EBCDIC character into a PLAN (32-bit) word.

Phrase. The name assigned to a language statement that becomes the identifiable dictionary index.

PLAN level error recovery. An action resulting from detection by PLAN of an error. The current PLAN module is terminated, all programs in the pop-up list are skipped, and processing starts at the next equal- or higher-level command.

Program. A group of FORTRAN or Assembly language source statements processed as one compilation or assembly.

Program list. Program names or program numbers maintained in a push-down, pop-up list where the top of the list always indicates the next program to be loaded.

Scan. The process of examining statement syntax to determine correctness and meaning.

Standard precision. The standard size of a FORTRAN variable word defined by ASA standard FORTRAN.

Statement. An entity input to a processor consisting of the alphameric text.

Store. The process of placing a program into the program library or data into a core storage facility.

Switch Words. A group of 15 32-bit words that provide communication of control information between PLAN and user modules and between user modules.

Unpack. The process of extracting EBCDIC characters from a PLAN word.

UREND. A logical end-of-file indicator for PLAN data files.

Variables. Data with changing numeric, logical, or alphameric values.

Verb. A phrase that modifies the meaning and syntax of another phrase.

99.0.0 INDEX

Items in this index are arranged alphabetically. Entries include module names, subroutine names, phrase names, PLAN terms, and language definer characters. Numeric references define the section in which pertinent explanations or examples of use may be found. The exact page of the reference may be found by locating the section reference number in the table of contents. Primary references are underlined.

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N. Y. 10601
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

READER'S COMMENT FORM

Problem Language Analyzer (PLAN) GH20-0594-1

Program Description Manual

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM. If you wish a reply, be sure to include your name and address.

COMMENTS

fold fold

fold fold

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A. FOLD ON TWO LINES, STAPLE AND MAIL.

YOUR COMMENTS PLEASE...

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

Please note that requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or the IBM branch office serving your locality.

fold fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N. Y.

BUSINESS REPLY MAIL
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY...

IBM Corporation
112 East Post Road
White Plains, N. Y. 10601

Attention: Technical Publications

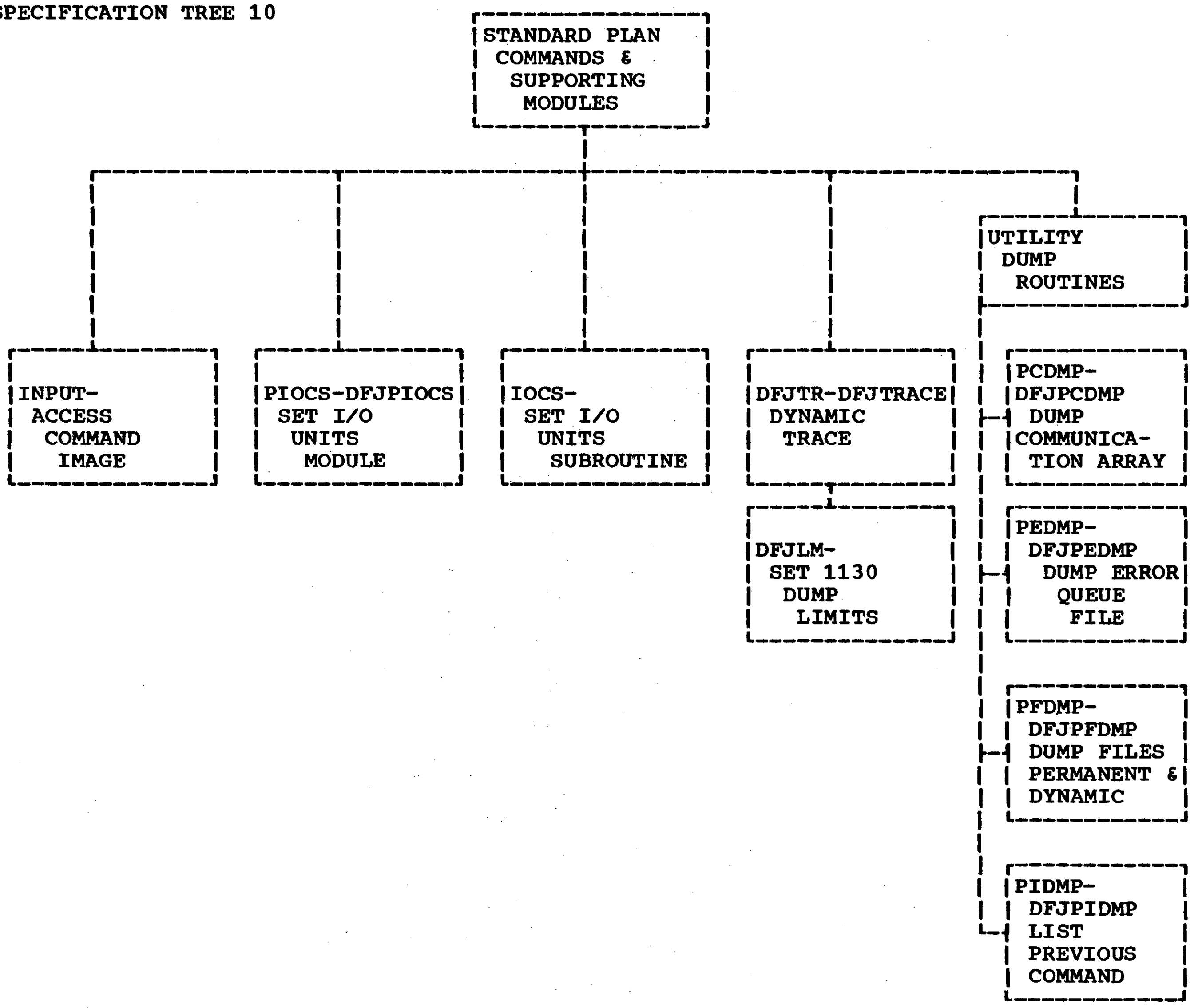
fold fold

IBM®

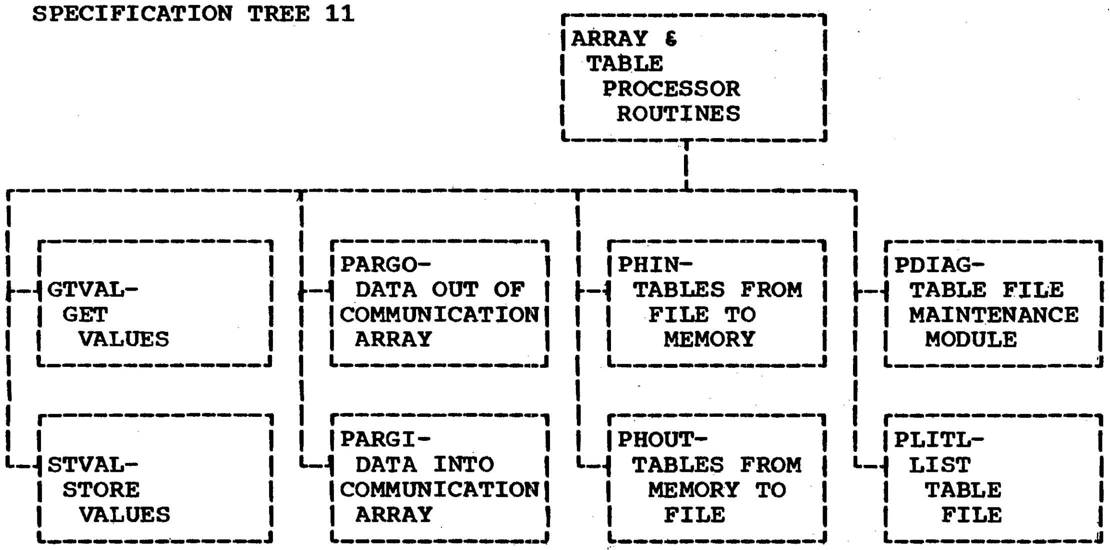
International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

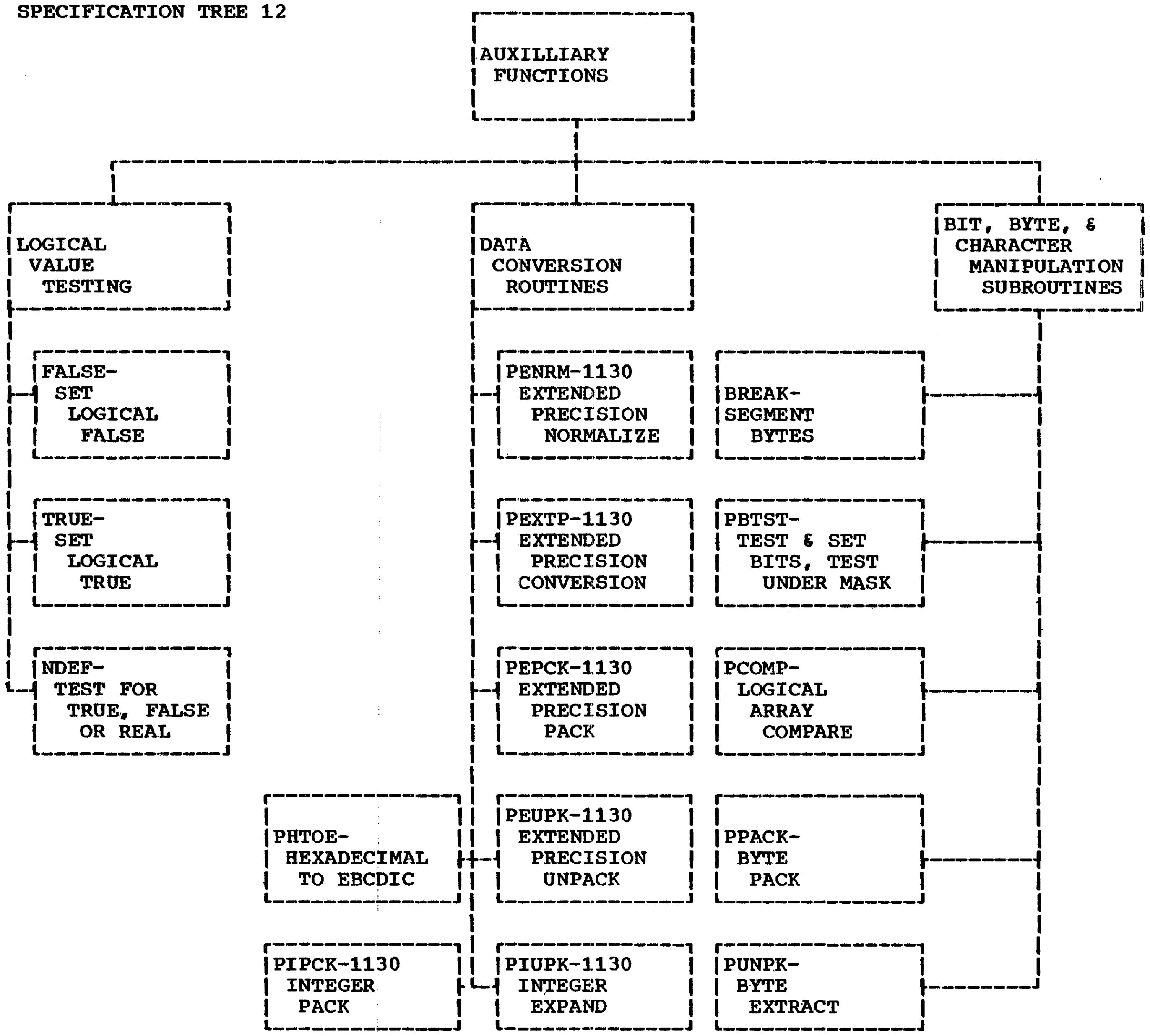
SPECIFICATION TREE 10


STANDARD PLAN COMMANDS & SUPPORTING MODULES
INPUT- ACCESS COMMAND IMAGE
PIOCS-DFJPIOCS SET I/O UNITS MODULE
IOCS- SET I/O UNITS SUBROUTINE
DFJTR-DFJTRACE DYNAMIC TRACE
DFJLM- SET 1130 DUMP LIMITS
UTILITY DUMP ROUTINES
PCDMP- DFJPCDMP DUMP COMMUNICATION ARRAY
PEDMP- DFJPEDMP DUMP ERROR QUEUE FILE
PFDMP- DFJPFDMP DUMP FILES PERMANENT & DYNAMIC
PIDMP- DFJPIDMP LIST PREVIOUS COMMAND

SPECIFICATION TREE 11


ARRAY & TABLE PROCESSOR ROUTINES
GTVAL- GET VALUES
STVAL- STORE VALUES
PARGO- DATA OUT OF COMMUNICATION ARRAY
PARGI- DATA INTO COMMUNICATION ARRAY
PHIN- TABLES FROM FILE TO MEMORY
PHOUT- TABLES FROM MEMORY TO FILE
PDIAG- TABLE FILE MAINTENANCE MODULE
PLITL- LIST TABLE FILE

SPECIFICATION TREE 12


AUXILLIARY FUNCTIONS
LOGICAL VALUE TESTING
DATA CONVERSION ROUTINES
BIT, BYTE, & CHARACTER MANIPULATION SUBROUTINES
FALSE- SET LOGICAL FALSE
TRUE- SET LOGICAL TRUE
NDEF- TEST FOR TRUE, FALSE OR REAL
PENRM-1130 EXTENDED PRECISION NORMALIZE
PEXTP-1130 EXTENDED PRECISION CONVERSION
PEPCK-1130 EXTENDED PRECISION PACK
PEUPK-1130 EXTENDED PRECISION UNPACK
PIUPK-1130 INTEGER EXPAND
PHTOE- HEXADECIMAL TO EBCDIC
PIPCK-1130 INTEGER PACK
BREAK- SEGMENT BYTES
PBTST- TEST & SET BITS, TEST UNDER MASK
PCOMP- LOGICAL ARRAY COMPARE
PPACK- BYTE PACK
PUNPK- BYTE EXTRACT

19.0.0 APPENDIX L: COMMUNICATION ARRAY LAYOUT CHART

The chart in this section may be copied and used for planning the utilization of the communication array layout.

COMMUNICATION ARRAY LAYOUT FOR: DATE:

DATA NAME		DATA NAME		DATA NAME		DATA NAME		DATA NAME		DATA NAME		DATA NAME		DATA NAME	
CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE	CAP	MODE
DEFAULT		DEFAULT		DEFAULT		DEFAULT		DEFAULT		DEFAULT		DEFAULT		DEFAULT	