Honeywell

PL/I USER'S GUIDE

SERIES 60 (LEVEL 66)/6000

SOFTWARE

Honeywell

PL/I USER'S GUIDE

SERIES 60 (LEVEL 66)/6000

SUBJECT:

User's Guide for PL/I in a GCOS Environment

SOFTWARE SUPPORTED:

Software Release PL 1.0

DATE:

November 1975

ORDER NUMBER:

DE04, Rev. 0

PREFACE

This manual describes the use of PL/I in a GCOS environment for Series 60 (Level 66)/6000 systems. The manual includes information on the execution of a PL/I program, file generation and access, compiler processing, loader functions, required control cards, and internal representation of PL/I data. Also, examples are included which are complete and executable. These examples contain the control cards and data necessary for execution; in addition sample output listings produced from the execution of some of these programs are also given.

File No.: IP33, 1733

CONTENTS

ILLUSTRATIONS

TABLES

SECTION I

INTRODUCTION

This manual describes the ways in which the general facilities of the General Comprehensive Operating Supervisor (GCOS) are applied to the specific tasks of compiling, loading, and executing PL/I programs.

DESCRIPTION OF THE MANUAL

The scope and structure of the User's Guide are described in the following paragraphs and then a list of related manuals is given.

Scope of the Manual

This User's Guide is a self-contained and complete introduction to the use of PL/I for the Series 60 (Level 66)/6000 (hereafter referred to as Series 60). Therefore, it contains some basic information about GCOS to aid the programmer using this system for the first time. The necessary control cards, for example, are described and illustrated. Whenever a basic Series 60 concept is discussed, a reference is given to the manual that contains the detailed description. However, the information given in this manual about these concepts is sufficient for their initial use.

Many examples are included in this manual. These examples are complete and executable; they contain all the control cards and data necessary for their execution. Also included are sample output listings produced from the execution of some of these programs.

Structure of the Manual

The sections of the User's Guide are ordered to provide, first the basic information about the use of the Series 60, then the details on the use of PL/I, and finally guidelines and examples.

After the introductory material, the control cards required to compile and execute a PL/I program are given and the use of the standard input and output files is described. The sections that cover this material are:

II Execution of a PL/I Program
III System Input/Output Files

Next, detailed descriptions are given for the two large system programs involved in compilation and execution, namely: the compiler and the loader. The characteristics of the compiler are described and the compiler output listing is explained and illustrated. The relevant loader control cards are given and the overlay capability is described. The sections are:

IV Compiler
V Loader

Next, the use of external files is described. For each type of organization, the method of attachment and an example of file access are given. The sections are:

VI External Files
VII CONSECUTIVE and INTERACTIVE Organization
VIII INDEXED Organization
IX REGIONAL Organization

Next, some details of the compiler program are given. The method of linking PL/I programs with programs written in other languages is described. The internal representation of PL/I data is described and storage layout rules for variables given. The sections are:

X Linking PL/I and Other Languages
XI Internal Representation of PL/I Data

Next, examples of the creation, modification, and use of the INCLUDE file are given. The section is:

XII INCLUDE Files

Next, a detailed description of debugging PL/I programs is presented. The messages printed upon the abnormal termination of a job are discussed and the methods for locating different types of PL/I variables in a memory dump are described and illustrated. The section is:

XIII Debugging PL/I Programs

Next, a series of hints on the effective use of PL/I are given. Methods for optimizing PL/I programs are suggested and some common programming errors illustrated. The sections are:

XIV Efficiency Considerations
XV Common Programming Errors

Finally, a sample problem is programmed in PL/I in two ways. The first program illustrates how a programmer can use PL/I to solve a problem quickly for his own use. The second program illustrates the use of PL/I for the development of a routine for a production environment. The section is:

XVI Solution of a Problem in PL/I

In addition to these sections, appendixes are included in the User's Guide to give reference material in tabular form. The appendixes are:

A	Restrictions in Series 60 PL/I
B	Comparison of Series 60 PL/I and Standard PL/I
C	Memory Limits
D	Character Conversion Tables
E	Internal Representation of PL/I Data Tyes
F	External Names
G	Format of the INCLUDE File
H	Error Messages
I	ON Codes

Related Manuals

Additional information on the PL/I language and the Series 60 is available. The PL/I language is described in another Honeywell publication, as follows:

> The PL/I Reference Manual (Order Number DE05) describes the Series 60 PL/I language. Each feature of the language is explained by an example, and the rules of the language are given in definitions that are informal but complete.

The aspects of the Series 60 environment discussed in this manual are described in other Honeywell publications, as follows:

> The General Comprehensive Operating Supervisor (GCOS) manual (Order Number DD19) describes the functions of GCOS.
>
> The Control Cards Reference Manual (Order Number DD31) describes the control cards used in the execution of the activities of a job.
>
> The General Loader manual (Order Number DD10) describes the general-purpose loader used to initiate an execution activity.
>
> The File and Record Control manual (Order Number DD07) describes file processing.
>
> The Indexed Sequential Processor manual (Order Number DD38) describes the processor used for creating, accessing, and maintaining files with indexed sequential organization.

GCOS FUNCTIONS

The General Comprehensive Operating Supervisor (GCOS) consists of a set of control programs and processing programs that monitor the current status of all system resources and jobs in the system and allocate optimum resources to each job.

GCOS performs the following functions:

Input media conversion
Resource allocation
Execution
Termination
Output media conversion

Each of these functions is considered in the following paragraphs.

Input media conversion handles input data in two categories: system related data and program data. The system related data consists of control cards that define peripheral devices, processors, and storage requirements for the activities belonging to a job. The program data consists of the information to be processed by the program.

Resource allocation is based on the information obtained from the control cards. If the resources required for a job activity are not currently available in the system, the job activity is suspended.

Execution of the activity under the supervision of a dispatcher begins when all the necessary resources are secured. The dispatcher queues activities with an attached priority and processes activities from the queue in the order of their priorities.

Termination follows the completion of a job activity. Errors and accounting information about the job activity are written on the SYSOUT file, the file is closed, and all resources allocated to the job activity are released.

Output media conversion takes place when all the activities that constitute a job are processed sequentially through allocation, execution, and termination.

Compilation of PL/I Programs

The compilation of a PL/I program requires the execution of a large system program, namely the PL/I compiler. GCOS loads the PL/I compiler from a catalogued master file, allocates the necessary resources for the compiler, and passes control to the compiler. The compiler then accepts a PL/I source program and translates it to an object program if no uncorrectable errors are found.

Note that PL/I source programs must be prepared using either the BCD or ASCII character set. Section XV discusses special character considerations and Appendix D lists the graphic and punch-card representations of these character sets.

The PL/I compiler can be used to perform a simple syntactic check of a source program, to compile a source program, or to compile and optimize a source program. The amount of processing done by the compiler is specified by the use of options. The structure of the compiler and the compiler-directing options are described later, in the section on the "Compiler".

The compiler operates in batch processing mode. The size of the source program determines the amount of memory that is required. Approximately 80K is needed for the compilation of a small PL/I program and 100K for an average program.

Loading and Execution of PL/I Programs

The execution of a PL/I program requires the loading of that program and the necessary subroutine group. The object program, the called subroutines, and the run-time packages are linked and the object program is executed.

SECTION II

EXECUTION OF A PL/I PROGRAM

This section describes the control cards that are used to compile and execute a PL/I program. These control cards define the compilation parameters, loading and execution operations, peripheral device assignments and core storage requirements.

DECK SETUP

An example of a basic control card setup that compiles and executes a PL/I external procedure follows. In this example, the source program is on cards.

```
1      8         16
$      SNUMB     12345
$      IDENT     ZETA,X2233,STOP2
$      OPTION    PL1
$      PL1       LIST
         .
         .       PL/I Source Program
         .
$      EXECUTE
$      LIMITS    2,30K,-4K
$      ENDJOB
***EOF
```

CONTROL CARDS

A detailed description of the control cards is given in the Control Cards Reference Manual. A brief description for each card in the example is given here.

SNUMB Control Card

The $ SNUMB control card provides an identifying name for the job. The format of the $ SNUMB card is:

```
1      8         16
$      SNUMB     identifier
```

where: identifier is a 1- to 5-character alphanumeric name identifying the job.

IDENT Control Card

The $ IDENT control card supplies the account number for the job and the name of the user. The format of the $ IDENT card is:

```
1      8       16
$      IDENT   account-no,name
```

where: account-no is the account number

name is a 1- to 12-character name identifying the user.

OPTION Control Card

The $ OPTION control card sets all the options required for loading a program. This card is described later, in the section on the "Loader".

PL1 Control Card

The $ PL1 control card specifies the compilation activity. Options that direct the compilation can be given on this card. The format of the $ PL1 control card and the options that can be requested are described later, in the section on the "Compiler".

EXECUTE Control Card

The $ EXECUTE control card specifies the activity of loading and executing the program produced as a result of the compilation activity. In response to this card, the loader brings the program into memory and links library subroutines to the object program. Normal termination of the loading initiates the execution of the object program. Further discussion of the $ EXECUTE control card is given later, in the section on the "Loader".

LIMITS Control Card

The $ LIMITS control card modifies standard activity resource limits. The format of the $ LIMITS card is:

```
1      8       16
$      LIMITS  time,storage-1,storage-2,print-lines,I/O-time
```

where: time is a decimal integer that specifies the maximum processor run-time for the activity in hundredths of an hour.

storage-1 is a decimal integer followed by "K" that specifies the number of 1024 word blocks requested by a slave program that can be shared with the loader.

storage-2	is a decimal integer, followed by "K" and preceded by a minus sign, that specifies the number of 1024-word blocks to be added to the size of the General Loader to allow extra space for load tables.
print-lines	is a decimal integer that specifies the maximum number of print lines to be written on SYSPRINT.
I/O-time	is a decimal integer that specifies the maximum amount of I/O time in hundredths of an hour.

The $ LIMITS card is not required for PL/I compilation since the following standard limits are defined:

Time	Storage-1	Storage-2	Print-lines	I/O-time
.15	90K	0	12000	None

ENDJOB Control Card

The $ ENDJOB control card indicates the end of the job. The format of the $ ENDJOB card is:

```
1      8       16
$      ENDJOB
```

SECTION III

SYSTEM INPUT/OUTPUT FILES

The file handling capability of PL/I is general and flexible. Four types of file organization can be generated and accessed by PL/I programs, namely:

CONSECUTIVE
INDEXED
REGIONAL
INTERACTIVE

The basic concepts of file handling are described later, in the section on "External Files". The section on "External Files" is followed by three sections that give the details of file attachment and use.

A large number of PL/I programs, however, use only the system input/output files. Since knowledge of the general capability is not required for the use of the system input/output files, a brief description of these files is given in this section.

SYSTEM INPUT/OUTPUT FILE USE

The two system files are SYSIN, the system input file, and SYSPRINT, the system output file. These files need not be declared, opened, or closed. If the filename is omitted from a GET statement, the filename SYSIN is assumed; if the filename is omitted from a PUT statement, the filename SYSPRINT is assumed.

The system input/output files have the following description:

Filename	Attribute	Record Size
SYSIN	INPUT,STREAM	80 characters
SYSPRINT	OUTPUT,STREAM,PRINT	132 characters

System Input/Output File Codes

Files referenced in a PL/I program are attached to external files by a file code. The general rules for determining and using file codes are given later, in the section on "External Files". The file codes for the system input/output files are as follows:

Filename	Filecode
SYSIN	I*
SYSPRINT	P*

To provide a program with input data on the system input file, the file code I* is used.

System Input

The data for a PL/I program can be included in the deck that contains the control cards and program cards. To include the data, a $ DATA control card is used. The $ DATA control card writes files onto a temporary linked disk for input to a user activity. The $ DATA control card has the following format:

```
1       8         16
$       DATA      fc,options

where:   fc          is the 2-character code identifying the file.

         options     are described in the Control Cards Reference
                     Manual.
```

Deck Setup Including Program Data

When the input data for a program is included in the job deck, the deck setup of the previous section is modified to include a $ DATA control card, as follows:

```
1       8         16
$       SNUMB     12345
$       IDENT     ZETA,X2233,STOP2
$       OPTION    PL1
$       PL1
        .
        .         PL/I Source Program
        .
$       EXECUTE
$       LIMITS    2,30K,-4K
$       DATA      I*
        .
        .         Program Data
        .
$       ENDJOB
***EOF
```

EXAMPLE OF THE USE OF THE SYSTEM INPUT/OUTPUT FILES

The use of the system input/output files is illustrated in the program given in Figure 3-1. The program determines the largest and smallest items from a list of five items. The list of five items is read from the system input file and the minimum and maximum values are printed on the system output file.

```
1      8       16
$      SNUMB
$      IDENT
$      OPTION  PL1
$      PL1

EXAMPLE:  PROC OPTIONS(MAIN);

          DCL (N1, N2, N3, N4, N5, SMALL, LARGE) FIXED BIN;
          DCL (MIN, MAX) BUILTIN;

          ON ENDFILE (SYSIN) GOTO EXIT;

LOOP:     GET LIST (N1, N2, N3, N4, N5);
          SMALL = MIN(N1, N2, N3, N4, N5);
          LARGE = MAX(N1, N2, N3, N4, N5);
          PUT LIST (SMALL, LARGE) SKIP;
          GOTO LOOP;

EXIT:     END;

$      EXECUTE
$      LIMITS  2,30K,-2K
$      DATA    I*
4 3 5 6 1
7 2 9 3 6
50 60 20 10 80
-15 -20 -35 -5 -10
$      ENDJOB
***EOF
```

Figure 3-1. The Use of Standard Files

The compiler output listing obtained from the execution of the program of Figure 3-1 is reproduced in the next section of this manual to illustrate the different sections of an output listing.

The output from the program of Figure 3-1 follows the compiler output listing on the standard output file, as follows:

```
SNUMB = 7605T,  ACTIVITY # = 02  REPORT CODE = 01  RECORD COUNT = 000005

          1          6
          2          9
         10         80
        -35         -5
```

SECTION IV

COMPILER

This section describes the PL/I compiler, the files used by the compiler, the options that can be specified to adjust the behavior of the compiler, and the output listing produced by the compiler.

COMPILER PHASES

A PL/I source program is translated into an executable object program by the PL/I compiler in six phases. During these phases the compiler produces edited error messages or object programs, as required. The six phases are:

Description	Phase Name
Compiler control phase	1 COMMON
Syntax analysis phase	2 PARSE
Semantic analysis phase	3 SEMANT
Optimization phase	4 OPTIMIZER
Code generation phase	5 CODE GENERATION
Error message editing phase	6 DIAGNOSTIC

The programmer can determine the phases of the compiler that operate on his program by specifying options on the $ PL1 control card. The PARSE option directs the compiler to perform only the syntactic analysis phase; the CHECK option directs the compiler to perform only the syntactic and semantic analysis phases; and the OPTZ option directs the compiler to perform an optimization phase in addition to the usual phases of the compiler. The SEVERITY option directs the compiler to suppress error messages with level number less than the integer argument given with the option. A detailed description of the options recognized by the PL/I compiler is given later in this section.

The logical flow of the PL/I compiler is illustrated in Figure 4-1. If no options are specified, flow proceeds along the path straight down from the PARSE phase to the DIAGNOSTIC phase.

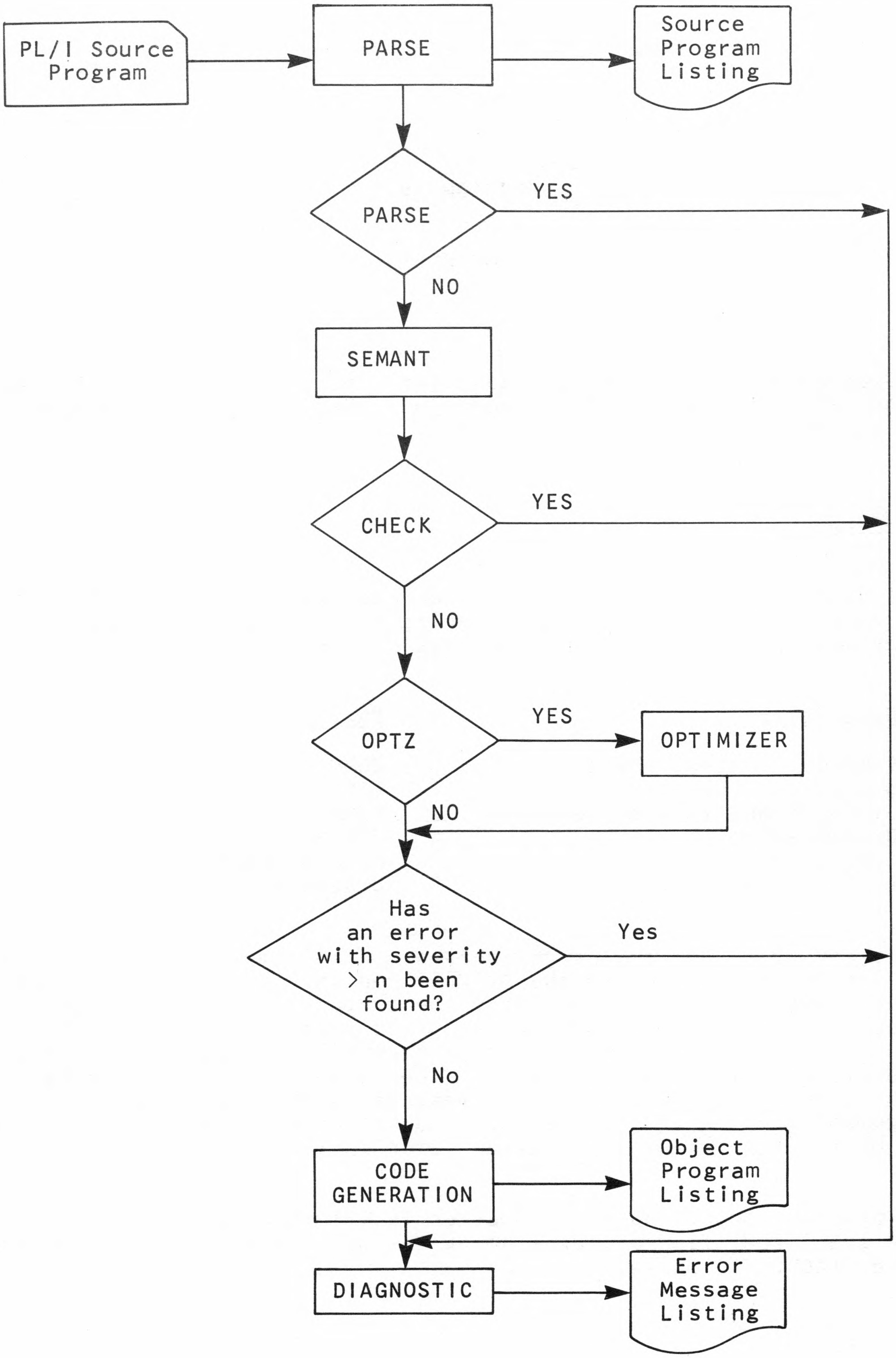


Figure 4-1. Logical Flow of the PL/I Compiler

Compiler Control Phase (COMMON)

The compiler control phase is in main storage throughout the compilation of a source program. This phase controls the execution of the other phases and performs the following actions:

Establish GCOS interfaces
Interpret compiler options
Determine overlay structure
Prepare output list
Determine storage space allocation
Prepare diagnostic message output
Prepare other services

Syntax Analysis Phase (PARSE)

The syntax analysis phase consists of two parts:

Lexical analysis
Parse

During lexical analysis, the compiler constructs a series of tokens to represent source language statements. During parse, the token string created by lexical analysis is used to create for the program a tree structure that represents the relationships that exist among the elements of the source program.

Semantic Analysis Phase (SEMANT)

The semantic analysis phase handles declaration and semantic conversion. The declaration process allocates storage for variables appearing in the program. The semantic conversion process analyzes the tree structure representing the source program and facilitates operator conversion and operand processing.

Optimization Phase (OPTIMIZER)

The optimization phase is an optional phase that can be requested by specifying the OPTZ option on the $ PL1 control card. The PL/I compiler produces reasonably efficient code without this phase. Two major optimizations are performed in this phase, namely:

Factoring of common sub-expressions
Moving invariant computations outside loops

This phase is usually requested for the final compilation of a production program.

Code Generation Phase (CODE GENERATION)

Two functions are performed by the code generation phase:

Allocation of storage space
Generation of the object code

The object code is the series of machine instructions generated from the tree representation of the program.

Error Message Editing Phase (DIAGNOSTIC)

The error message editing phase produces edited error messages describing the errors detected in the compilation of the source program. At the completion of this phase, control returns to the compiler control phase and the compilation is completed.

FILES USED DURING COMPILATION

The PL/I compiler uses standard system files, implicitly generated by GCOS. The files used in a PL/I compilation are given in Table 4-1. The relationship of these files to the compiler is shown in Figure 4-2. Each file is then described in more detail.

Table 4-1. Files Used in a PL/I Compilation

File Code	File Name	Size	Type
A*	Alter file	variable	linked
B*	Object program file	2 links	linked
C*	Object deck file		
D*	Stranger option file		
K*	Compressed deck file		
P*	System output file		
S*	Source program file	variable	linked
*3	Work file	5 links	random
.L	Include file[a]		

[a] If an include file is used, it must be prepared by the user.

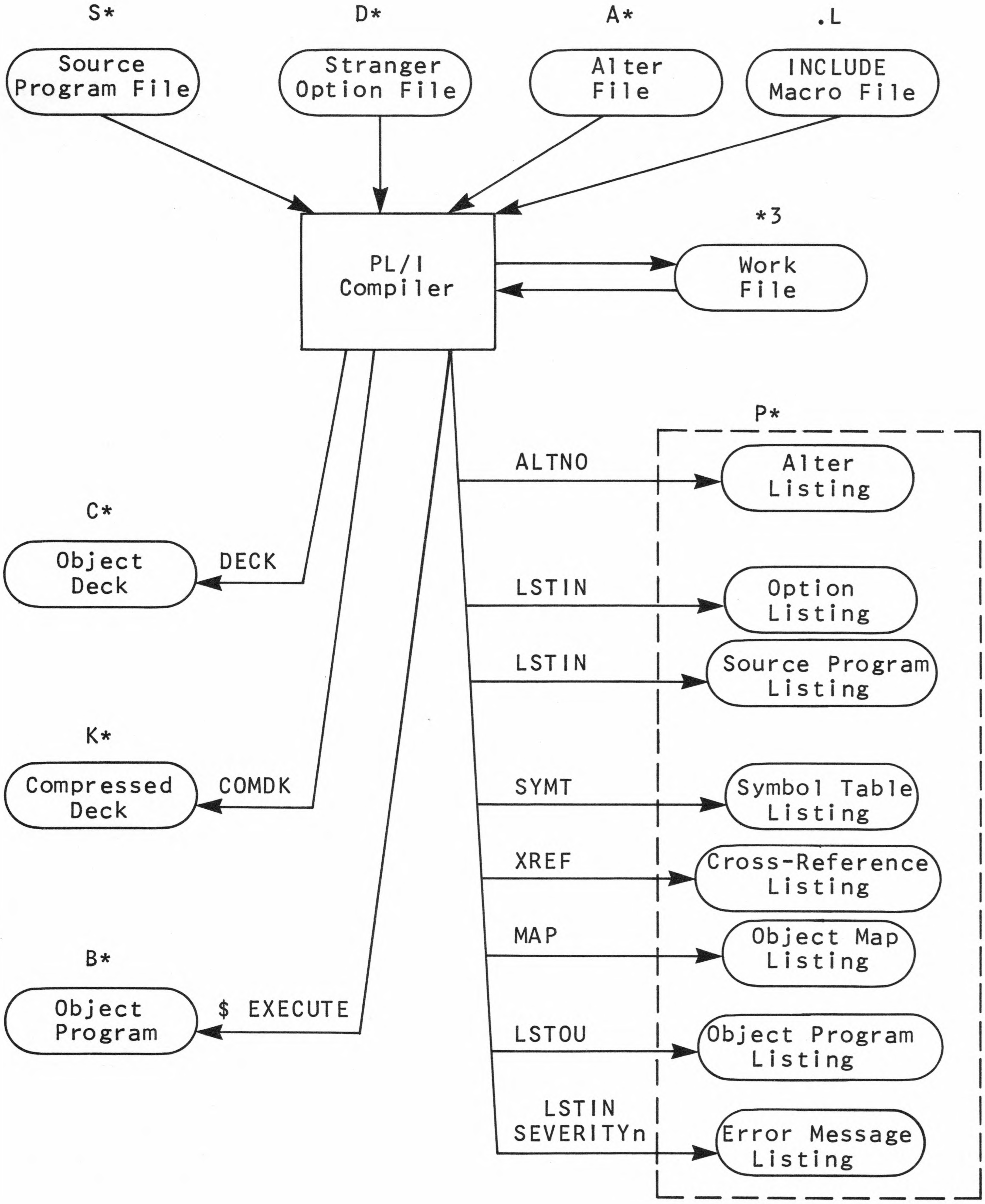


Figure 4-2. Files Used During Compilation

Alter File

The alter file (A*) is used to make partial modification of the PL/I source program to be compiled. A detailed description of the modification of a source program using $ ALTER cards is given in the File and Record Control manual.

The alter file is generated by the GCOS system input module when a $ UPDATE card is detected. The $ ALTER and source program cards are stored in the alter file by the system input module.

Object Program File

The object program file (B*) is used to store the object program generated as a result of the compilation of the source program by the PL/I compiler.

The object program file is generated by ALLOC, a GCOS module, when a $ EXECUTE card occurs in the job.

Object Deck File

The object deck file (C*) contains the deck generated by the PL/I compiler. The object deck begins with the $ OBJECT card and ends with $ DKEND card.

The object deck file is generated by ALLOC, a GCOS module, when the DECK option is given on the $ PL1 control card.

Stranger Option File

The stranger option file (D*) contains card images of options specific to the PL/I compiler given on the $ PL1 control card. The PL/I compiler interprets the options given in this file.

The stranger option file is generated by the GCOS system input module when options that have no commonality with other language processors are given on the $ PL1 control card.

Compressed Deck File

The compressed deck file (K*) contains the source program in a compressed form. A detailed description of a compressed deck is given in the File and Record Control manual.

The compressed deck file is generated by ALLOC when the COMDK option is given on the $ PL1 control card. The compressed deck is stored in the format of an input file to the PL/I compiler and can be used as a source program input (S*).

System Output File

The system output file (P*) is used when the PL/I compiler, or any other processing program under GCOS, outputs reports or processing results. This file is usually allocated to the system output device and is fed to this device via SYSOUT, a GCOS module.

The maximum number of lines that SYSOUT can output for one activity is 30,000. Output beyond this limit can also be fed to the line printer by first allocating the system output file to a magnetic file or disk and then transferring its contents to the line printer by the Bulk Media Conversion (BMC) module.

Listings that the PL/1 compiler outputs on the system output file (P*) are described and and illustrated later in this section.

The system output file is generated by ALLOC when processing results are output.

Source Program File

The source program file (S*) contains the PL/I source program image used as input for the compilation.

The source program file is generated by the system input module when source cards are present in the job stream. The source program file (S*) may also be provided by a $ PRMFL card.

Work File

The work file (*3) is used to store intermediate results.

The work file is generated by ALLOC.

INCLUDE File

The INCLUDE file is prepared by the user as a file to store macro text when the %INCLUDE statement is used in the PL/I program. Appendix G gives a detailed description of INCLUDE files.

OPTIONS

By the use of options the programmer can direct the compiler in the translation of his program. For example, options can be requested that limit or extend the amount of processing done by the compiler, that request additional output listings, that change the form of input and output, and that suppress a class of error messages.

An option can be requested, negated, or omitted. An option is requested by giving the option name on the appropriate control card; for example, to request optimization the programmer specifies OPTZ. An option is negated by specifying the option name prefixed with the letter 'N'; for example, to negate the option OPTZ, the programmer specifies NOPTZ on the appropriate control card. If an option is omitted, a default assumption is made about the specification of the option.

There are two types of options, standard options and special options. The option names and the method of specifying options are given for both types in the following paragraphs.

Standard Options

The standard options allow a programmer to determine which phases of the compiler operate on his program and to specify the listings and decks produced by the compilation.

Standard options are specified on the $ PL1 control card. For example, the following $ PL1 control card illustrates the specification of the standard options ALTNO and COMDK.

```
1      8      16
$      PL1    ALTNO,COMDK
```

A detailed description of the control card formats used to specify the standard options is given later in this section.

The standard options recognized for the PL/I compiler are given in Table 4-2. A brief description of the meaning and the associated default assumption is given for each option. Following the table, a more detailed description is given for each of the options.

Table 4-2. Standard Options

Option	Meaning	Default
ALTNO	Produce a list of the source program with alter numbers.	NO ALTNO
CHECK	Suspend the code generation phase.	NO CHECK
COMDK	Produce a compressed deck of the source program.	NO COMDK
CSYM	Include in the symbol table the internal names created by the compiler.	NO CSYM
DECK	Produce a binary deck for the object program.	NO DECK
LIST	Assume that LSTOU, MAP, SYMT, and XREF are specified.	NO LIST
LSTIN	Produce the option list, expanded source program, storage requirements, external symbols, and compiler storage requirements.	LSTIN
LSTOU	Produce a list of the object program.	NO LSTOU
MAP	Produce a map that associates the line numbers of the source program with the relative address in the object program.	NO MAP
NLSTIN	Cancel the LSTIN option.	LSTIN
OPTZ	Optimize the object program.	NO OPTZ
PARSE	Suspend semantic analysis and code generation.	NO PARSE
SEVERITYn	Suppress error messages with a severity level less than the indicated number.	SEVERITY1
SNUMBER	Attach the line number of the source program to error messages.	NO SNUMBER
STAB	Generate a complete symbol table that can be used at execution time.	NO STAB
SYMT	Produce a list of names used in the source program and their attributes.	NO SYMT
XREF	Produce a cross reference table with the symbol table indicating lines of declaration and reference for each name.	NO XREF

STANDARD OPTION NAMES

The standard options recognized by the PL/I compiler are described in the following paragraphs.

ALTNO Option

The ALTNO option directs the compiler to produce a listing of the source program with attached alter numbers. The source program listing is an exact copy of the source program input to the compiler and contains star (*) option cards and %INCLUDE statements. This listing is useful for determining the line number in the program for altering the output compressed deck image, especially if the program contains the %INCLUDE statement or if the LONGFORM option is also requested.

CHECK Option

The CHECK option causes the compiler to suppress the code generation phase. Syntax analysis and semantic analysis are performed and any errors detected during these phases are reported.

The use of this option allows the programmer to save computer time during the initial compilations of his program when the probability of errors is high.

COMDK Option

The COMDK option directs the compiler to produce a compressed deck of the source program. Columns 73 through 76 of this deck contain the name of the deck, composed of four characters specified by the TITLE option or as described below. Columns 77 through 80 contain a sequence number.

CSYM Option

The CSYM option directs the compiler to add the internal names created by the compiler to the symbol table output listing. The attributes of each internal name are given in the listing. The CSYM option is recognized only when the SYMT option is also requested.

DECK Option

The DECK option directs the compiler to produce a binary deck for the object program.

LIST Option

The LIST option directs the compiler to proceed as if the LSTOU, MAP, SYMT, and XREF were requested. The LIST option is a convenient way to obtain a complete output listing.

LSTIN Option

The LSTIN option directs the compiler to produce the option listing, the expanded source program listing, the storage space and external symbol listing, and the storage capacity required at compile time. These listings are parts of the compiler output listing described and illustrated later in this section.

The LSTIN option is the only option with a positive default assumption. Therefore, if no options are specified, the output requested by the LSTIN option is produced.

LSTOU Option

The LSTOU option directs the compiler to produce a listing of the object program in the compiler listing. The object program is given in a format similar to that of assembly language.

The object program listing is described and illustrated later in this section.

MAP Option

The MAP option directs the compiler to produce the table that gives the association between the line number of the source program and the relative address of the generated object code for that line of source language.

The object program map listing is described and illustrated later in this section.

OPTZ Option

The OPTZ option directs the compiler to perform an additional optimization phase. In this additional phase, common sub-expressions are eliminated. The code produced by the compiler, without this additional phase, is quite efficient. The decision to request the OPTZ option is based on considerations of program size and frequency of execution.

PARSE Option

The PARSE option directs the compiler to suppress the semantic analysis and code generation phases. This option is useful for the initial compilations of programs that use the %INCLUDE statement in which syntactic errors can be especially serious.

SEVERITY Option

The SEVERITY option, with its associated integer 'n', directs the compiler to suppress the listing of any error messages with level number less than 'n'.

Error levels for PL/I error messages range from level 1 (warning level) to level 4 (fatal error). The error levels are categorized, as follows:

Level	Meaning
1	Warning. The program contains a construction that may be in error. The compilation is not affected by an error of level 1.
2	Correctable error. Errors of level 2 are corrected by the compiler and the compilation process continues unless the correction affects the process adversely.
3	Uncorrectable error. Errors of level 3 are not fatal, but cannot be corrected. The compilation process continues from the next logical point in the program, but code generation is suspended.
4	Fatal error. Errors of level 4 cause compiler termination.

For example, the option SEVERITY2 causes warnings (level 1) to be suppressed but allows error messages with a level greater than or equal to 2 to be listed.

SNUMBER Option

The SNUMBER option directs the compiler to include the corresponding source program statement number and line number in the information given by the error trace-back when an error occurs at execution time.

STAB Option

The STAB option directs the compiler to generate a complete symbol table for use at execution time. Variable names, label names, and entry names referred to in the source program are arranged so that execution time debugging can be performed conveniently.

SYMT Option

The SYMT option directs the compiler to produce the symbol table listing as part of the compiler output listing. The symbol table listing contains the names used in the source program with their attributes. The symbol table listing is described and illustrated later in this section.

XREF Option

The XREF option directs the compiler to include in the symbol table listing the line numbers on which each name is declared and referenced. The XREF option implies the SYMT option and automatically specifies it.

STANDARD OPTION CONTROL CARDS

Standard options are given on the $ PL1 control card, in the following way:

```
1      8      16                                       73     80
------------------------------------------------------------------
$      PL1    option,option,...                        (not used)
```

The options are separated by commas and can be given in any order. A control card is terminated by a blank column, so no imbedded blanks can be included in the option list. If the last nonblank character on the card is a comma, more options are assumed to follow. The additional options are given on a $ ETC control card, as follows:

```
1      8      16                                       73     80
------------------------------------------------------------------
$      PL1    option,option,                           (not used)
$      ETC    option,...                               (not used)
```

The format of the $ ETC control card is similar to that of the $ PL1 control card. If the $ ETC control card ends with a comma, another $ ETC control card is assumed to follow.

Each option name must be entirely contained on one control card and cannot be continued from one card to another.

The following example requests the options DECK, MAP, LSTOU, and SYMT on a single $ PL1 control card:

```
$      PL1    DECK,MAP,LSTOU,SYMT
```

The same request can be made on several cards, as follows:

```
$      PL1    DECK,
$      ETC    MAP,LSTOU,
$      ETC    SYMT
```

Special Options

The special options are used to name a program, to determine conventions for different versions of PL/I, to request statistical information about the compilation process, and to change the format of output listings.

Special options are given on star (*) control cards and are interpreted by the PL/I compiler. For example, the following star (*) control cards assign the name PRG1 to any output program decks, provide a listing title and request the special option SMESSAGE.

```
1              13                            69
---------------------------------------------------
*TITLE         ALPHA PROGRAM MAIN LISTING    PRG1
*OPTIONS       SMESSAGE
```

A detailed description of the control card formats used to specify the special options is given later in this section.

The special options recognized by the PL/I compiler on the *OPTIONS cards are given in Table 4-3 with a brief description of their meanings. Following the table, a more detailed description is given for each option.

Table 4-3. Special Options

Option	Meaning
FLOATBIN	Regard scaled arithmetic fixed-point constants as arithmetic floating-point constants.
IBMFORM	Process source in columns 2 - 72 only.
LONGFORM	Interpret the character '#' or "@" in column 1 as the continuation symbol.
SEC_SYMDEF	Create for each external entry name containing a '$' character a corresponding entry name to be used as a secondary symbol.
SHORT_CALL	Reduce the size of the object program by restricting the code generated for subroutine calling sequences.
SMESSAGE	Do not use the full printer for error message output. Limit it to 80 columns.
STATUS	Produce statistical data about the compilation.

SPECIAL OPTION NAMES

The special option names recognized by the PL/I compiler are described in this section.

FLOATBIN Option

The FLOATBIN option directs the compiler to regard any scaled arithmetic fixed-point constant in the source program as an arithmetic floating-point constant.

This option is provided so that programs written in the IBM DOS PL/I language can be compiled by the PL/I compiler.

IBMFORM Option

The IBMFORM option directs the compiler to process data in columns 2 through 72 of the input card. Column 1 and columns 73 through 80 are not processed when this option is requested. If this option is not requested, the entire card is processed.

LONGFORM Option

The LONGFORM option directs the compiler to interpret the character '#' or "@" in column 1 of the source program input card as a symbol indicating the continuation of output lines in the source program listing produced by the compiler.

Since the image of each input card with this continuation symbol is the logical continuation of the preceding card, the compiler tries, insofar as possible, to put the complete image on a single line in the program listing.

This option is provided to allow the programmer to produce a program listing that is easy to read and to understand.

SEC_SYMDEF Option

The SEC_SYMDEF option directs the compiler to create a corresponding entry name to be used as a secondary symbol for each external entry name that contains a '$' character.

The loader processing of the secondary SYMDEF occurs after the processing of the primary SYMDEF. The <u>General Loader</u> manual contains a complete description of this processing.

SHORT_CALL Option

The SHORT_CALL option directs the compiler to reduce the size of the object program by restricting the code generated for subroutine calling sequences. The object programs created when this option is specified are smaller in size but execute less efficiently than programs created when the option is not requested or is negated.

SMESSAGE Option

The SMESSAGE option directs the compiler to use only 80 columns of the printer line for the listing of error messages. This option is used to limit page width.

STATUS Option

The STATUS option directs the compiler to produce statistical data about the compilation of the source program in the compiler listing.

The compiling statistic list, produced as a result of requesting this option, is described and illustrated later in this section.

SPECIAL OPTION CONTROL CARDS

Special options are given on one or more star (*) control cards. The star control card is a PL/I control card and contains options that are specific to the PL/I compiler. The star control cards must be given first in the PL/I source program input deck.

The TITLE option card must be the first card if it is present. Its format is:

```
1          13                                     63         73      80
________________________________________________________________________

*TITLE     Listing heading of user's choice    YYMMDDXXXX(not used)
```

The listing heading replaces the standard main title line of the compilation listings. The date is optional and the date of compilation is entered if columns 63-68 are blank. The date is placed in columns 67-72 of the $ OBJECT card of the object deck (if any). Columns 69-72 of the *TITLE card are used for identification of output object and compressed source decks if such decks are requested by the appropriate options. The first 32 characters of the title are reproduced on the $ OBJECT card of the object deck (if any) beginning at column 16. If no TITLE card is present, the deck identification is four zero characters and blanks are entered on the $ OBJECT card.

The SUBTITLE option card must be the second card if it is present. The format is:

```
1          13                                                73      80
________________________________________________________________________

*SUBTITLE  Listing subheading                               (not used)
```

Columns 13-72 become the subtitle on the compilation listing. If there is no SUBTITLE card the subtitle consists of the standard subtitle line that includes the first text line image of the program.

The COPYRIGHT option card has the following format:

```
1           13                                               73      80
________________________________________________________________________

*COPYRIGHT  "any desired copyright message"                 (not used)
```

The given message will be printed in a box formed with asterisks on the option listing page of the compiler output listing. A typical message could be

"COPYRIGHT 1975 BY THE ABC WIDGET CO."

The format of the OPTIONS option card is as follows:

```
1           13                                                          73      80
*OPTIONS    option,...                                                  (not used)
```

The special options given in Table 4-3 are separated by commas and can be given in any order. More than one *OPTIONS card may be present.

Example of the Use of Options

The following program fragment illustrates the use of both standard and special options:

```
1      8    13 16                                                   69

$      SNUMB   12345
$      IDENT   ZETA,X2233,STOP2
$      OPTION  PL1
$      PL1     DECK,COMDK,LIST,
$      ETC     ALTNO
*TITLE      PROG1                                                   PRGA
*OPTIONS    SMESSAGE,STATUS
       .
       .       PL/I Source Program
       .
$      EXECUTE
$      LIMITS  2,30K,-4K
$      ENDJOB
***EOF
```

The standard options DECK, COMDK, LIST, and ALTNO and the special options SMESSAGE and STATUS are requested.

COMPILER OUTPUT LISTING

The compiler output listing is divided into sections. Each section is a listing that gives information about the source program, the compilation, or the object program. The programmer can select the sections of the listing to be produced by specifying options. The sections are shown in Table 4-4 in the order in which they appear, if requested, in the output listing. Associated with each section in the table is the option whose specification causes that section of the listing to be produced.

Following the table, each section of the listing is described and illustrated. The program EXAMPLE, given in Figure 3-1, was used to produce the sample listings included here.

Table 4-4. Sections of the Compiler Output Listing

Section	Option
Alter listing	ALTNO
Compiler option listing	LSTIN
Expanded source program listing	LSTIN
Symbol table	SYMT
Cross reference table	XREF
Storage space and external symbol listing	LSTIN
Object program map	MAP
Object program listing	LSTOU
Error message listing	SEVERITYn
Compiling statistics listing	STATUS
Storage capacity required at compile time	LSTIN

Alter Listing

The alter listing is a listing of the source program with alter numbers. This listing is used to change the compressed deck of the source program. The numbers associated with the lines of the source program in the alter listing can be different from the line numbers in the expanded source program listing due to the presence of star (*) option cards and the %INCLUDE statement.

The alter listing for the sample program EXAMPLE follows. This listing shows that the special option STATUS was given on a star (*) control card.

```
ALTER NO           SOURCE IMAGE OF THIS PROGRAM

     1 *OPTIONS    STATUS
     2  EXAMPLE:      PROC OPTIONS(MAIN);
     3        DCL (N1, N2, N3, N4, N5, SMALL, LARGE) FIXED BIN;
     4        DCL (MIN, MAX) BUILTIN;
     5        ON ENDFILE(SYSIN) GOTO EXIT;
     6  LOOP: GET LIST(N1, N2, N3, N4, N5);
     7        SMALL = MIN(N1, N2, N3, N4, N5);
     8        LARGE = MAX(N1, N2, N3, N4, N5);
     9        PUT LIST(SMALL, LARGE) SKIP;
    10        GOTO LOOP;
    11  EXIT: END;
```

Compiler Option Listing

The complete set of standard option specifications is given in this listing. Options that are not specified on control cards are assumed to have the default interpretation described earlier in this section.

Any special options given on a star (*) control card are listed. The complete set of special options, however, is only included in the listing when the STATUS option is requested.

The compiler option listing produced as a result of a job that requested the standard options ALTNO, COMDK, CSYM, and DECK and the special option STATUS is given below.

```
OPTIONS USED IN THIS COMPILATION

 *OPTIONS    STATUS


 COMPLETE LIST OF OPTIONS
                LSTIN
             NO LIST
             NO MAP
             NO SYMT
             NO LSTOU
                ALTNO
                CSYM
             NO PARSE
             NO CHECK
             NO OPTZ
             NO SEVERITY
             NO STAB
                DECK
                COMDK
             NO SNUMBER
             NO XREF
                STATUS
             NO SHORT_CALL
             NO LONGFORM
             NO IBMFORM
             NO SEC_SYMDEF
             NO FLOATBIN
```

Expanded Source Program Listing

The expanded source program listing gives a numbered list of the source program. If a %INCLUDE statement is present, it is replaced in this listing by the expanded image. The nesting level of the DO group is given on each line, following the line number.

If the LONGFORM option is requested, the expanded source program listing occupies columns 9 - 136.

The expanded source program listing for the sample program EXAMPLE is given below. Notice that the line numbers differ from the line numbers in the alter listing given earlier in this section due to the presence of the star (*) control card.

```
       COMPILATION LISTING OF PROGRAM: EXAMPLE:      PROC OPTIONS(MAIN);

   1     EXAMPLE:      PROC OPTIONS(MAIN);
   2           DCL (N1, N2, N3, N4, N5, SMALL, LARGE) FIXED BIN;
   3           DCL (MIN, MAX) BUILTIN;
   4           ON ENDFILE(SYSIN) GOTO EXIT;
   5     LOOP: GET LIST(N1, N2, N3, N4, N5);
   6           SMALL = MIN(N1, N2, N3, N4, N5);
   7           LARGE = MAX(N1, N2, N3, N4, N5);
   8           PUT LIST(SMALL, LARGE) SKIP;
   9           GOTO LOOP;
  10     EXIT: END;
```

Symbol Table and Cross Reference Table

The symbol table listing is a list of names declared or used in the source program. The names are given in the following order in the table:

- Names declared explicitly by the DECLARE statement.
- Names declared but not used.
- Names declared explicitly through context outside of the DECLARE statement. (For example, label constants, format constants, and entry constants.)
- Names declared implicitly or by context.

For each name, the following information is listed:

- If the name is that of a structure member, its structure relative address is given in the form of a word offset (in octal) and bit offset (in decimal).
- Address.
- Storage space attributes.
- Data type attributes.

If the XREF option is requested, the cross reference listing is produced for each name in the symbol table. The cross reference listing indicates the line numbers of the declarations and references for each name and an indication as to whether or not the value of the variable is set.

References to a DEFINED variable are included in the cross reference table for the base variable since the storage generation for the two variables are the same. The cross reference table for a pointer variable, which implicitly modifies (i.e., appears in the declaration of) a BASED variable, includes all references to the BASED variable except those using a different pointer for qualification. If the BASED variable has an upper bound, lower bound, or length specified by an expression, a simple reference to the BASED variable implies a reference to any variables used in the expression.

The symbol table and the cross reference table for the sample program EXAMPLE is given here. For reproduction in this manual, the format has been compressed by reducing the length of each field on the listing.

```
                  *** NAMES DECLARED IN THIS COMPILATION ***

IDENTIFIER OFFSET    LOC STORAGE CLASS DATA TYPE      ATTR AND REFERENCES

*NAMES DECLARED BY DECLARE STATEMENT*
LARGE              000014 AUTOMATIC  FIXED BIN(17,0)  DCL 2 SET REF 7 8
MAX                                  BUILTIN FUNCTION INTERNAL DCL 3 REF 7
MIN                                  BUILTIN FUNCTION INTERNAL DCL 3 REF 6
N1                 000006 AUTOMATIC  FIXED BIN(17,0)  DCL 2 SET REF 5 6 7
N2                 000007 AUTOMATIC  FIXED BIN(17,0)  DCL 2 SET REF 5 6 7
N3                 000010 AUTOMATIC  FIXED BIN(17,0)  DCL 2 SET REF 5 6 7
N4                 000011 AUTOMATIC  FIXED BIN(17,0)  DCL 2 SET REF 5 6 7
N5                 000012 AUTOMATIC  FIXED BIN(17,0)  DCL 2 SET REF 5 6 7
SMALL              000013 AUTOMATIC  FIXED BIN(17,0)  DCL 2 SET REF 6 8

*NAMES DECLARED BY EXPLICIT CONTEXT*
EXAMPLE            000035 CONSTANT   ENTRY            EXTERNAL DCL 1 REF 1
EXIT               000167 CONSTANT   LABEL            DCL 10 REF 10 4
LOOP               000074 CONSTANT   LABEL            DCL 5 REF 5 9

*NAMES DECLARED BY CONTEXT OR IMPLICATION*
ENDFILE            000015 STACK REF  CONDITION        REF 4
SYSIN              000003 CONSTANT   FILE             SET REF 4 5
SYSPRINT           000004 CONSTANT   FILE             SET REF 8
```

Storage Space and External Symbol Listing

The storage space listing gives the amount of storage required for the object program and the automatic storage requirements. The object program size is given in words and includes the required storage space for INTERNAL STATIC variables and constants. The number of V count bits is also given. The number of words of automatic storage determined by the constants required for the procedure block and the automatic storage required by BEGIN blocks and ON units are given in this listing.

The external symbol listing gives the external operators, external entries, and external variables used in the program.

The storage space listing and external symbol listing for the sample program EXAMPLE are, as follows:

```
            COMPILATION LISTING OF PROGRAM: EXAMPLE:      PROC OPTIONS(MAIN);


   *STORAGE REQUIREMENTS FOR THIS PROGRAM*

   OBJECT PROGRAM SIZE IS 120 WORDS.  (V COUNT 5)

   EXTERNAL PROCEDURE "EXAMPLE" USES 58 WORDS OF AUTOMATIC STORAGE
   ON UNIT ON LINE 4 USES 6 WORDS OF AUTOMATIC STORAGE

   *THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM*
   GET_LIST_NP_AL      EXT_ENTRY           ON_UNIT_ENTRY       RETURN_MAC
   PUT_LIST_NP_AL      TRA_EXT_1           GET_TERMINATE       PUT_TERMINATE
   ENABLE_FILE         GET_PREP            PUT_PREP

  *NO EXTERNAL ENTRIES ARE CALLED BY THIS PROGRAM*

  *THE FOLLOWING EXTERNAL VARIABLES ARE USED BY THIS PROGRAM*
  SYSIN                SYSIN#              SYSPRINT           SYSPRINT#

  *EXTERNAL NAMES AND CONVERTED NAMES OF THEM*

  EXAMPLE                                                     7EMPLE

  SYSPRINT                                                    8SRINT
```

Object Program Map

The object program map listing is produced when the MAP option or the LIST option is requested. The object program map listing gives for each line of the source program the relative address for the start of the corresponding object program code. The number of words required for the object code translation of the source line is also given in the form of a zero-suppressed decimal number, truncated to 2 digits.

The object program map for EXAMPLE is given here, compressed to four columns per line. The actual computer listing gives seven columns per line.

```
            *OBJECT MAP*

            LINE SIZE  LOC   LINE SIZE  LOC   LINE SIZE  LOC   LINE SIZE  LOC
               1   5 000032          3 000043      4   6 000055      5   5 000074
               6  14 000115     7   14 000133      8   7 000151      9   1 000166
              10   1 000167
```

Object Program Listing

When the LSTOU option is requested, the object program listing is produced. This listing consists of the series of assembly language instructions produced as a result of the translation of the source program.

The object program is produced in the following order:

INTERNAL STATIC region
Label constant array
Literal constants
FORMAT information
Object program procedure

A portion of the object program listing for the sample program EXAMPLE is given below:

```
BEGIN PROCEDURE "EXAMPLE"
ENTRY TO "EXAMPLE"                                   STATEMENT 1 ON LINE 1
    000032  105 130 101 115 000   EXAM
    000033  120 114 105 040 000   PLE
    000034  000000 000007   000   ZERO    0,7
    000035  050000 7000 00  030   TSXBP  .P0090           EXT_ENTRY
    000036  000000 000072   000   ZERO    0,58
    000037  000035 4500 12  000   STZ     29,SP
    000040  000035 7420 12  000   STXSP   29,SP
    000041  000022 6200 12  000   EAXBP   18,SP
    000042  000004 7400 12  000   STXBP   4,SP
    000043  020000 6200 00  030   EAXBP   SYSPRINT#
    000044  000000 6360 10  000   EAQ     0,BP
    000045  040000 7560 00  030   STQ     SYSPRINT
    000046  777751 6360 04  000   EAQ     -23,IC           000017
    000047  020003 7560 00  030   STQ     SYSPRINT#+3
    000050  010000 6200 00  030   EAXBP   SYSIN#
    000051  000000 6360 10  000   EAQ     0,BP
    000052  030000 7560 00  030   STQ     SYSIN
    000053  777731 6360 04  000   EAQ     -39,IC           000004
    000054  010003 7560 00  030   STQ     SYSIN#+3

                                                   STATEMENT 1 ON LINE 4
    000055  000007 7260 07  000   LXL6    7,DL
    000056  030000 6200 00  030   EAXBP   SYSIN
    000057  777723 6350 04  000   EAA     -45,IC           000002
    000060  060000 7010 00  030   TSXLP   .P0376           ENABLE_FILE
    000061  000006 7100 04  000   TRA     6,IC             000067
    000062  000012 7100 04  000   TRA     10,IC            000074
```

Error Message Listing

The error message listing contains the errors that are detected during the translation of the source program. Each error message has an associated level number, between one and four. A description of the error level classification is given in connection with the SEVERITY option earlier in this section.

The error message list gives the error number, the severity level, the line number in the source listing at which the error was detected and explanatory text describing the error.

The error message listing for the sample program EXAMPLE is given below. An error of severity level 1 is printed as a WARNING.

```
WARNING 75
THE UNDECLARED IDENTIFIER "SYSPRINT" HAS BEEN CONTEXTUALLY DECLARED AS A FILE
CONSTANT.  IT WILL ACQUIRE DEFAULT ATTRIBUTES.

WARNING 133
THE UNDECLARED IDENTIFIER "ENDFILE" HAS BEEN CONTEXTUALLY DECLARED AS A
CONDITION NAME.  IT WILL ACQUIRE DEFAULT ATTRIBUTES.

WARNING 75
THE UNDECLARED IDENTIFIER "SYSIN" HAS BEEN CONTEXTUALLY DECLARED AS A FILE
CONSTANT.  IT WILL ACQUIRE DEFAULT ATTRIBUTES.

WARNING 495
IMPLEMENTATION RESTRICTION: LONG EXTERNAL NAME "EXAMPLE" HAS BEEN CONVERTED TO A
6 CHARACTER NAME.  RESTRICTIONS ARE: EXTERNAL FILE NAME SIZE <= 5 AND OTHER
EXTERNAL NAME SIZE <= 6.

WARNING 495
IMPLEMENTATION RESTRICTION: LONG EXTERNAL NAME "SYSPRINT" HAS BEEN CONVERTED TO
A 6 CHARACTER NAME.  RESTRICTIONS ARE: EXTERNAL FILE NAME SIZE <= 5 AND OTHER
EXTERNAL NAME SIZE <= 6.
```

Compiling Statistics Listing

When the STATUS option is requested, the compiling statistics listing is produced. This listing contains statistical information about the performance of the compiler in the translation of the source program.

The compiling statistics listing summarizes the usage of tokens, nodes, symbols, statements, and core. The compiling statistics listing for the sample program EXAMPLE is given below.

```
       COMPILATION LISTING OF PROGRAM: EXAMPLE:      PROC OPTIONS(MAIN);

*STATISTICAL DATA FOR PROGRAM*

<SUMMARY OF TOKEN USAGE>

 THE NUMBER OF TOKENS IS 59
 THE NUMBER OF EMPTY HASH TABLE SLOT IS 160
 THE MAXIMUM NUMBER OF TOKENS IN A SLOT IS 2
 THE TOTAL NUMBER OF WORDS IS 332

<SUMMARY OF SYMBOL USAGE>

 THE TOTAL NUMBER OF COMPILER CREATED SYMBOLS IS 16

<SUMMARY OF NODE USAGE>

BLOCK           4    STATEMENT      27    OPERATOR       52    REFERENCE      80
TOKEN          59    SYMBOL         28    CONTEXT         3    LIST           18
MC_STATE        2    STORAGE        28    LABEL           6    XREF            1

<SUMMARY OF STATEMENT USAGE>

DUMMY_ST        1    ASSIGN ETC.    12    END             2    GET             3
GOTO            2    NULL            2    ON              1    PROCEDURE       2
PUT             2

<SUMMARY OF CORE USAGE>

* MAXIMUM STACK SIZE  = 004651
* SIZE FOR PGM_TREE   = 001262
* EXTENDED CORE SIZE  = 000000
```

Storage Capacity Required at Compile Time

This section of the compiler listing consists of a single line giving the amount of storage required to compile the program, as follows:

```
**       66K WAS USED TO COMPILE THIS PROGRAM.
```

For the sample program EXAMPLE, 66K was required to compile the program.

SECTION V

LOADER

This section provides an introduction to the General Loader activities necessary for the execution of a PL/I program in the GCOS environment. The loader functions, loader control cards, and overlay structures are described. A detailed description of the loader is given in the General Loader manual.

DESCRIPTION OF LOADER FUNCTIONS

The General Loader produces an executable unit from a set of object programs, control cards, and libraries. The General Loader performs the following functions:

Linkage of object programs into a single object unit

Linkage of referenced library routines to the object unit

Assignment of main storage space required by the program, including common reservations

Definition of the overlay structure

Creation of file control blocks for the manipulation of files required by the object program

Upon completion of this processing, the loader passes control to an entry name within the object unit and the execution of the object programs begins.

Loader Processing

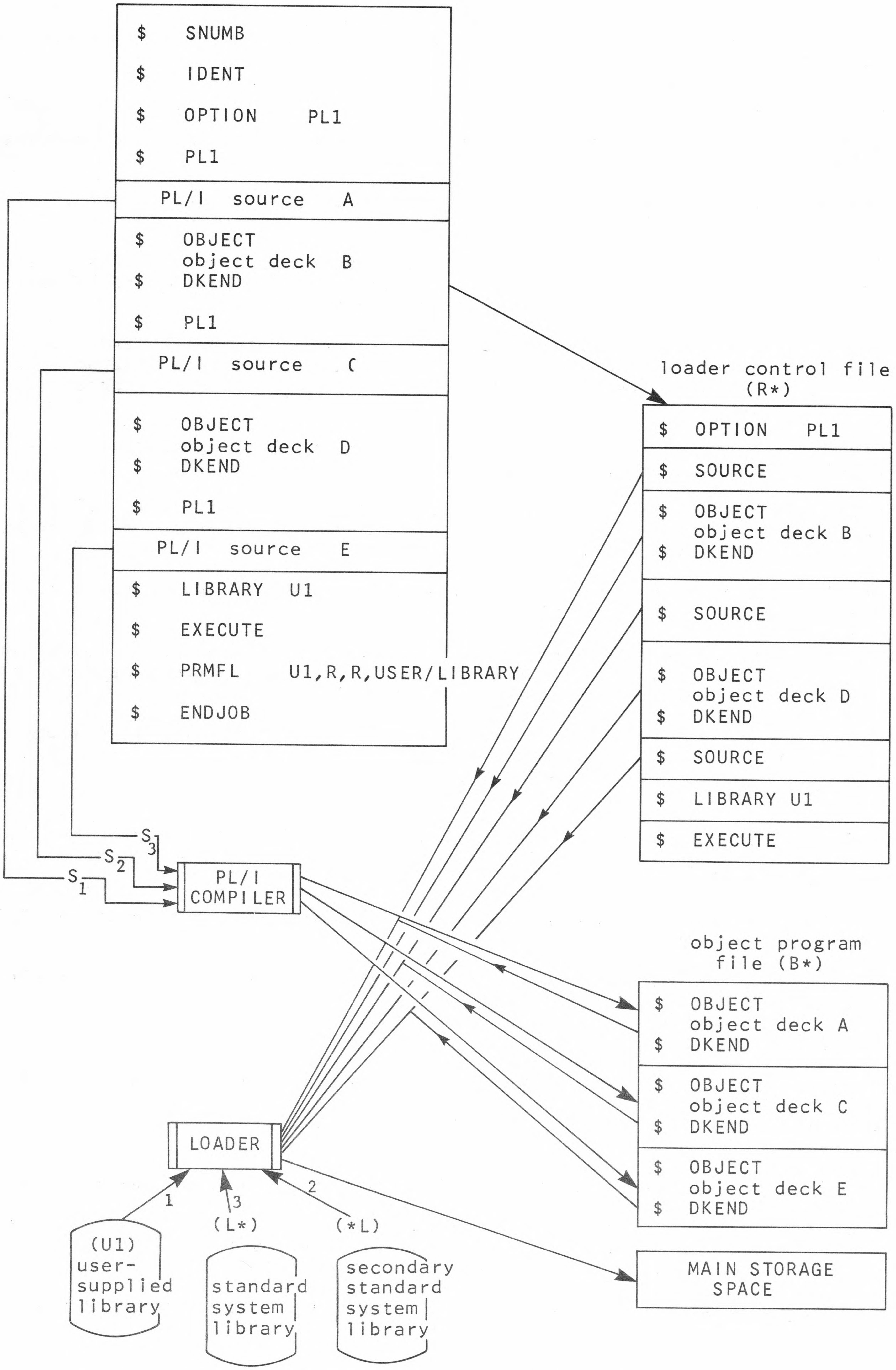
The object decks created by the PL/I compiler and other language processors are composed of two types of cards: preface cards and text cards. Preface cards contain information about the size and external names of the object program. The internal procedure names declared in the program (SYMDEFs), the external procedure names referenced in the program (SYMREFs), and the external variables declared and referenced in the program (Labeled Commons) are given on preface cards. Text cards contain the machine instructions and data for the program.

The loader obtains input from the GCOS standard files identified by the file codes R* and B*. The loader control file (R*) contains control cards and object programs from the input deck. The object program file (B*) contains object programs produced by the PL/I compiler. The loader's primary input is the loader control file. When the loader encounters a $ SOURCE control card on that file, it inputs the corresponding object program from the object program file.

When the object programs are loaded, the external variable procedure names referenced by the programs are resolved. The loader uses information contained on the preface cards to resolve SYMREFs, searching first any user-supplied libraries, then the secondary system standard library (*L), and finally the system standard library (L*). Every library program included in the object program by the PL/I compiler in the translation of the source program is contained in the standard system library. (Some installations may include them in the secondary system standard library.)

Input Deck Processing

Figure 5-1 illustrates the processing of a typical input deck by GCOS, the construction of the files used by the loader, and the processing of these files by the loader. Following the figure, the action taken by GCOS for each control card is described. Then, the action taken by the loader for each control card on the loader control file is given.



```
$    SNUMB
$    IDENT
$    OPTION    PL1
$    PL1
     PL/I  source   A
$    OBJECT
     object deck  B
$    DKEND
$    PL1
     PL/I  source   (
$    OBJECT
     object deck  D
$    DKEND
$    PL1
     PL/I  source   E
$    LIBRARY  U1
$    EXECUTE
$    PRMFL    U1,R,R,USER/LIBRARY
$    ENDJOB
```

loader control file (R*)

```
$  OPTION   PL1
$  SOURCE
$  OBJECT
   object deck B
$  DKEND
$  SOURCE
$  OBJECT
   object deck D
$  DKEND
$  SOURCE
$  LIBRARY U1
$  EXECUTE
```

object program file (B*)

```
$  OBJECT
   object deck A
$  DKEND
$  OBJECT
   object deck C
$  DKEND
$  OBJECT
   object deck E
$  DKEND
```

Figure 5-1. Input Deck Processing

GCOS processes the input deck and performs the following actions for the indicated control cards:

Card	Action
$ SNUMB $ IDENT	Records the information on these cards for accounting purposes.
$ OPTION	Copies the $ OPTION control card to the loader control file (R*).
$ PL1	Writes a $ SOURCE card on the loader control file (R*), sets limits, allocates files and arranges for control to be passed to the PL/I compiler to translate program A. The PL/I compiler reads as its source program all cards up to the next control card, translates the source program A, and produces the object program for A on the object program file (B*).
$ OBJECT $ DKEND	Copies the object deck B with its delimiting control cards to the loader control file (R*).
$ PL1	Writes a $ SOURCE card on the loader control file (R*) and performs, for source program C, the same actions as described above for source program A.
$ OBJECT $ DKEND	Copies the object deck D with its delimiting control cards to the loader control file (R*).
$ PL1	Writes a $ SOURCE card on the loader control file (R*) and performs, for source program E, the same actions as described above for source programs A and C.
$ LIBRARY	Copies the $ LIBRARY card to the loader control file (R*).
$ EXECUTE	Passes control to the General Loader.

The General Loader then reads the loader control file (R*) created by GCOS during the processing of the input deck and performs the following actions for the indicated control cards:

$ OPTION	Sets the loader options necessary for the execution of a PL/I program, namely: LOWLOAD and PSETU.
$ SOURCE	Loads the object program (A) from the object program file (B*).
$ OBJECT $ DKEND	Loads the enclosed object program (B).
$ SOURCE	Loads the object program (C) from the object program file (B*).
$ OBJECT $ DKEND	Loads the enclosed object program (D).
$ SOURCE	Loads the object program (E) from the object program file (B*).
$ LIBRARY	Searches the user-supplied library (U1) to resolve any undefined SYMREFs, then the secondary system standard library (*L) and the system standard library (L*).
$ EXECUTE	Passes control to the object program at the appropriate entry point.

LOADER CONTROL CARDS

Loader control cards give information to the General Loader about the object programs that are to be executed. These control cards indicate the beginning, end, and entry point name for the object programs and define options, libraries, and memory allocation methods. In addition, the program can be divided into overlay segments by the use of the loader control card $ LINK. A more detailed description of these and other loader control cards can be found in the General Loader manual and Control Card Reference Manual.

The loader control cards described in this manual are summarized in Table 5-1. Following the table, the cards are described in more detail.

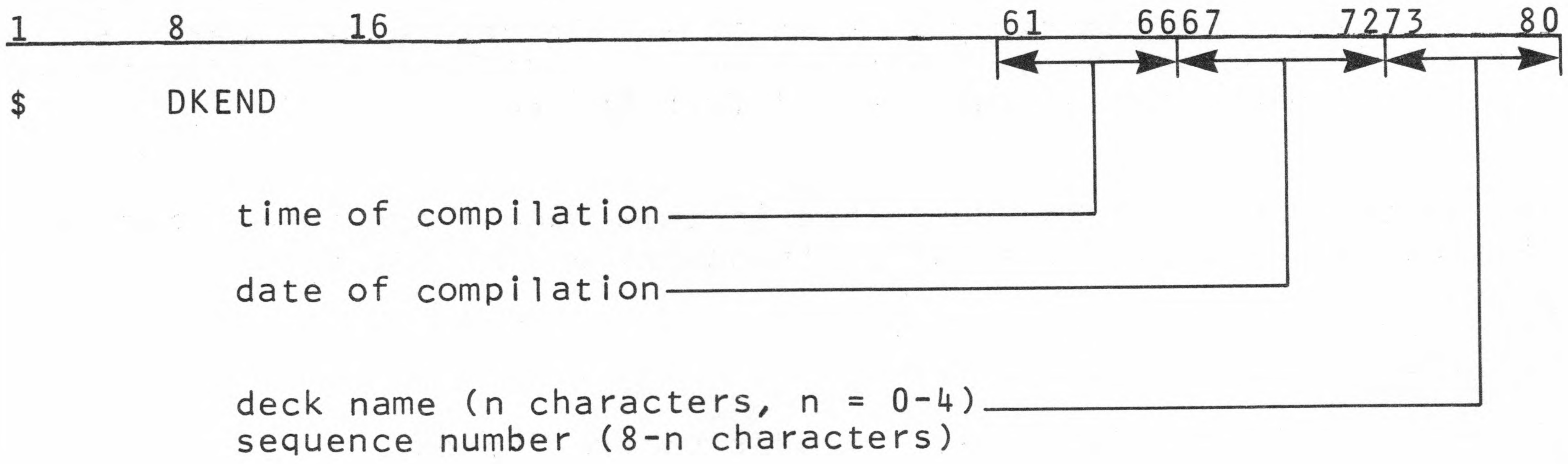
Table 5-1. Loader Control Cards

Card Name	Meaning	Parameters[1]
DKEND	End Object deck.	
ENTRY	Specify SYMDEF name to which Loader passes control.	name
EXECUTE	Conclude loading and pass control to object program.	sense switches, dump option
FFILE	Describe nonstandard file control blocks and options.	options
LIBRARY	Include user libraries to resolve SYMREFs.	file codes of user libraries
LINK	Define overlay structure.	segment names, option
OBJECT	Begin object deck.	
OPTION	Define Loader options	options
SOURCE	Read object program from object program file (B*).	
USE	Define Labeled Common Block or SYMREF.	name and size

[1] Refer to the General Loader manual for a detailed description of the parameters.

DKEND Control Card

The $ DKEND control card indicates the end of the absolute or relocatable deck. The PL/I compiler generates this card at the end of the object deck. The $ DKEND card has the following format:


1
8
16
61
6667
7273
80
$
DKEND
time of compilation
date of compilation
deck name (n characters, n = 0-4)
sequence number (8-n characters)

ENTRY Control Card

The $ ENTRY control card specifies the name to which the loader passes control upon the completion of the loading process. The $ ENTRY control card has the following format:

```
1      8      16
$      ENTRY  name
```

where: name is a SYMDEF corresponding to an external entry point name for the program.

If this card is not included, the loader passes control to either the first external entry name that has the OPTIONS (MAIN) attribute in its PROCEDURE statement or to the special SYMDEF . . . . . ., if present.

EXECUTE Control Card

The $ EXECUTE control card causes GCOS to activate the loader to load all the programs in the activity. The options on the $ EXECUTE control card request the setting of sense switches and the form of the dump. The $ EXECUTE card has the following format:

```
1      8      16
$      EXECUTE options
```

where: the following options can appear:

ON1	Set sense switch 1 on.
ON2	Set sense switch 2 on.
ON3	Set sense switch 3 on.
ON4	Set sense switch 4 on.
ON5	Set sense switch 5 on.
ON6	Set sense switch 6 on.
DUMP	Take full dump if activity terminates abnormally.
NDUMP	Dump only registers if activity terminates abnormally.

If no options are requested on the $ EXECUTE control card, all sense switches are set off and only the registers are dumped on abnormal termination of an activity.

FFILE Control Card

The $ FFILE control card describes nonstandard file control blocks and nonstandard file options. This control card is described later, in the section on "External Files".

LIBRARY Control Card

The $ LIBRARY control card directs the loader to search the user-supplied libraries whose file codes are given as parameters. The libraries are searched in the order in which they appear on the card. The format of the $ LIBRARY card is as follows:

```
1      8      16
$      LIBRARY fc,...
```

where: fc is the 2-character alphanumeric file code of the user library.

Consider the following card:

```
$      LIBRARY A1,C2
```

If this card is included, the loader searches the user library identified by the file code A1 and then the user library identified by C2 to resolve SYMREFs.

LINK Control Card

The $ LINK control card defines the overlay structure. The parameters on this card define the name of the segment, and optionally the name of the previously defined segment to be overlaid by this segment and the NOPAC option. The format of the $ LINK card is as follows:

```
1      8      16
$      LINK   seg-name[,oseg-name[,NOPAC]]
```

where: seg-name is the name of the segment composed of the programs following the control card up to the next $ LINK or $ EXECUTE control card.

oseg-name is the name of the previously defined segment to be overlaid.

NOPAC indicates that SYMDEFs in the overlaid segment can be referenced.

The $ LINK control card is described in more detail later in this section in connection with the definition of overlay structures.

OBJECT Control Card

The $ OBJECT control card indicates the beginning of the absolute or relocatable deck. The PL/I compiler generates this card at the beginning of the object program produced as a result of the translation of the source program. Identification is included on the card in the following format:

```
1      8       16                                        60      67    7273     80
$      OBJECT                                            P

     TTL date (see *TITLE description, Section IV)
     deck name (n characters, n = 0-4)
     sequence number (8-n characters)
```

For example:

```
$      OBJECT                                            P       111574CALC0000
```

indicates the following:

col 60 = P	The card was generated by the PL/I compiler.
col 67 - 72 = 111574	TTL date is November 15, 1974.
col 73 - 76 = CALC	The deck name is 'CALC'.
col 77 - 80 = 0000	The sequence number is 0.

OPTION Control Card

The $ OPTION control card specifies options for the loader. The $ OPTION card has the following format:

```
1      8       16
$      OPTION  options
```

where: option describes the execution of the program and the output of the loader.

The options that can be specified are given in Table 5-2.

Table 5-2. Loader Options

Option Name	Meaning	Default
MAP	Generate a memory map.	MAP
NOMAP	Do not generate a memory map.	MAP
CONGO	Execute even if errors detected.	CONGO
GO	Execute only if no errors detected.	CONGO
NOGO	Do not execute.	CONGO
ERCNT/n/	Abort the program if the total number of errors exceeds n.	ERCNT/150/
SYMREF	Include the SYMREF symbols used by each subprogram in the memory map.	NOSREF
NOSREF	Do not include SYMREF symbols in the memory map.	NOSREF
PL1	Specify all necessary conditions for the execution of PL/I programs (LOWLOAD, PSETU).	
NOMSUB	Suppress the missing routine message.	

Neat!

SOURCE Control Card

The $ SOURCE control card is generated by GCOS on the loader control file (R*) when a system call card (for example $ PL1) is encountered. The format of the $ SOURCE card is as follows:

```
1      8      16
$      SOURCE
```

This card indicates that the object program input for the loader is to be found on the object program file (B*).

USE Control Card

The $ USE control card specifies a name for the Labeled Common Block or SYMREF. A numeric size enclosed in slants immediately following the name defines the name to be a Labeled Common Block; otherwise, the name is assumed to be a SYMREF. The format of the $ USE control card is as follows:

```
1      8       16
$      USE     name[/size/],...
```

where: name is the name of a Labeled Common Block or SYMREF.

size is the amount of storage to be set aside for the Labeled Common Block.

The loader enters the name in its symbol table and, if a size is given, sets aside the necessary storage.

Consider the following example:

```
$      USE     ALPHA/400/,PXY,BETA/200/
```

ALPHA is a Labeled Common Block 400 words long.
PXY is a SYMREF
BETA is a Labeled Common Block 200 words long.

The $ USE control card is necessary for the attachment of INTERACTIVE, INDEXED, and REGIONAL files. This usage is described later, in the section describing file attachment.

OVERLAY STRUCTURE

A program that exceeds main storage capacity can be executed as an overlay structure. By the use of $ LINK control cards, a program is divided into a series of segments. These segments share storage and, therefore, must be loaded during the execution of the program as they are needed.

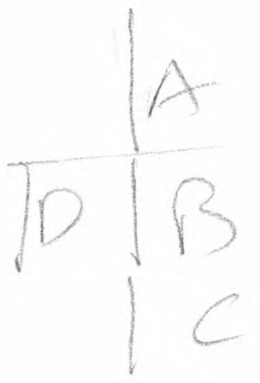
The definition of the overlay structure by $ LINK control cards, the tree representation for an overlay structure, and the routines used to load overlay segments are described in the following paragraphs.

Segment Definition

The $ LINK control cards delimit the segments of the overlay structure. A $ LINK control card with a segment name defines as the named segment the programs between that card and the next $ LINK or $ EXECUTE card. The segment name consists of one to six alphanumeric characters, the first of which must be alphabetic. Segment names must be unique with respect to SYMDEFs and other segment names.

Consider the following fragment of an input deck:

```
1      8       16
       .
       .
       .
$      LINK    ASEG
$      OBJECT
     object program A1
$      DKEND
$      PL1
     source program A2
$      LINK    BSEG
$      PL1
     source program B1

$      LINK    CSEG
       .
       .
       .
```



Two segments are defined in this fragment: ASEG and BSEG. The segment ASEG consists of the object program A1 and the object program produced as a result of the compilation of the source program A2. The segment BSEG consists of the object program produced as a result of the compilation of the source program B1.

Root Segment

The main segment of the overlay structure, the root segment, remains in main storage through the entire activity. The programs in the input deck preceding the first $ LINK control card make up the root segment. The loader generates the name '//////' for the root segment.

Segment Overlays

The $ LINK control card with two segment names as parameters defines a segment overlay. For example:

```
$      LINK    DSEG,BSEG
```

This card indicates that the segment DSEG is defined by the programs following and that the segment DSEG overlays the previously defined segment BSEG. The loader assigns the segment DSEG to the same starting location as the segment BSEG.

Example of an Overlay Setup

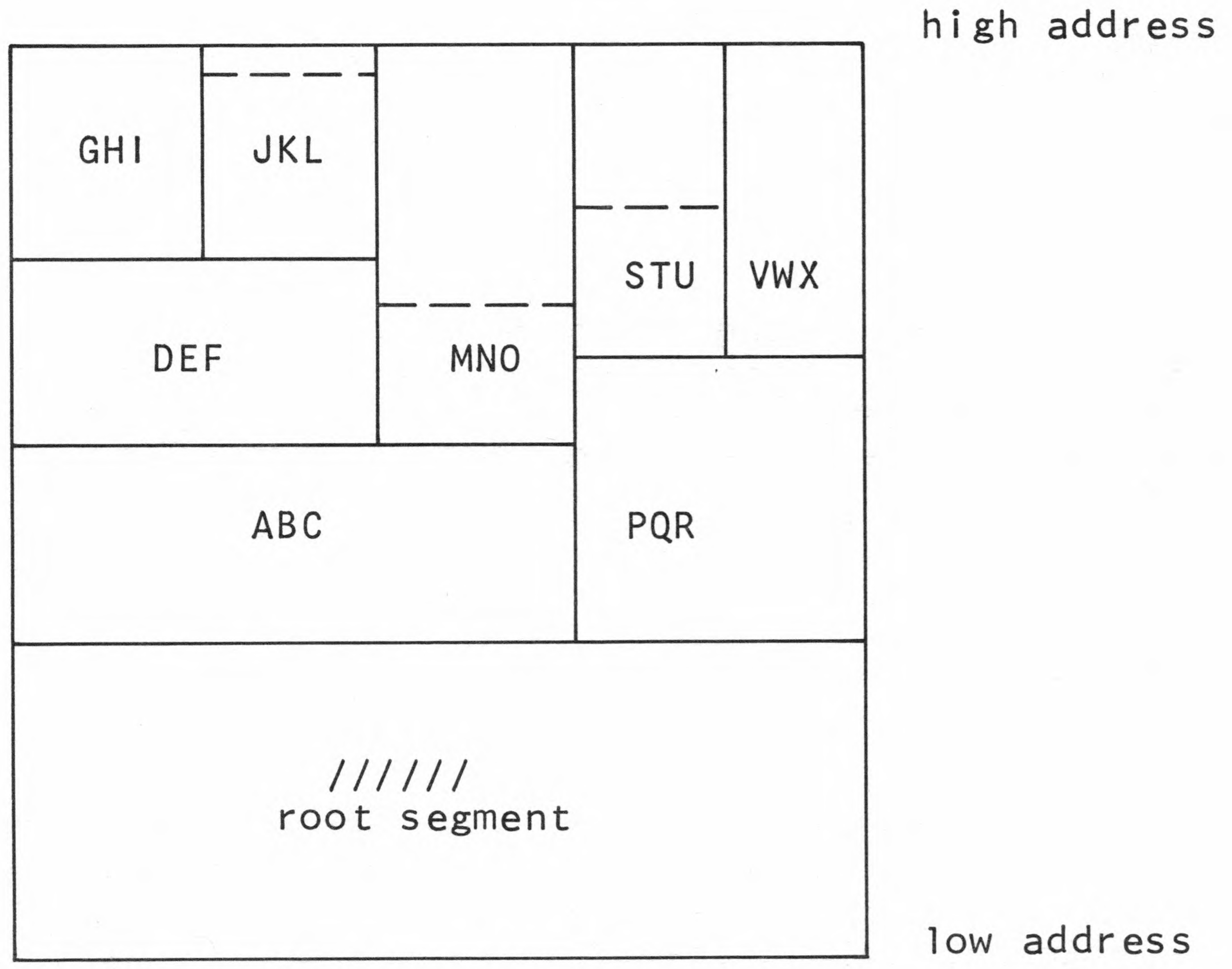
An input deck defining an overlay structure is given here. Following the deck setup a diagram of the memory allocation produced by the loader is given.

Consider the input deck:

```
1      8         16
$      SNUMB
$      IDENT
$      OPTION   PL1
$      PL1                                  root segment
  source program
  for root segment
$      LINK     ABC
       PL1                                  segment ABC
  source program
       A
$      LINK     DEF
$      PL1                                  segment DEF
  source program
       D
$      LINK     GHI
$      PL1                                  segment GHI
  source program
       G
$      LINK     JKL,GHI
$      PL1                                  segment JKL
  source program
       J
$      LINK     MNO,DEF
$      PL1                                  segment MNO
  source program
       M
$      LINK     PQR,ABC
$      PL1                                  segment PQR
  source program
       P
$      LINK     STU
$      PL1                                  segment STU
  source program
       S
$      LINK     VWX,STU
$      PL1                                  segment VWX
  source program
       V
$      EXECUTE
```

This input deck defines a root segment and eight overlay segments, namely: ABC, DEF, GHI, JKL, MNO, PQR, STU, and VWX. The segments JKL and GHI overlay each other, as do segments MNO and DEF, PQR and ABC, and VWX and STU. Each segment in this example is made up of a single source program. However, the segments can be made up of any number of source and object programs as is illustrated in a more general example later in this section.

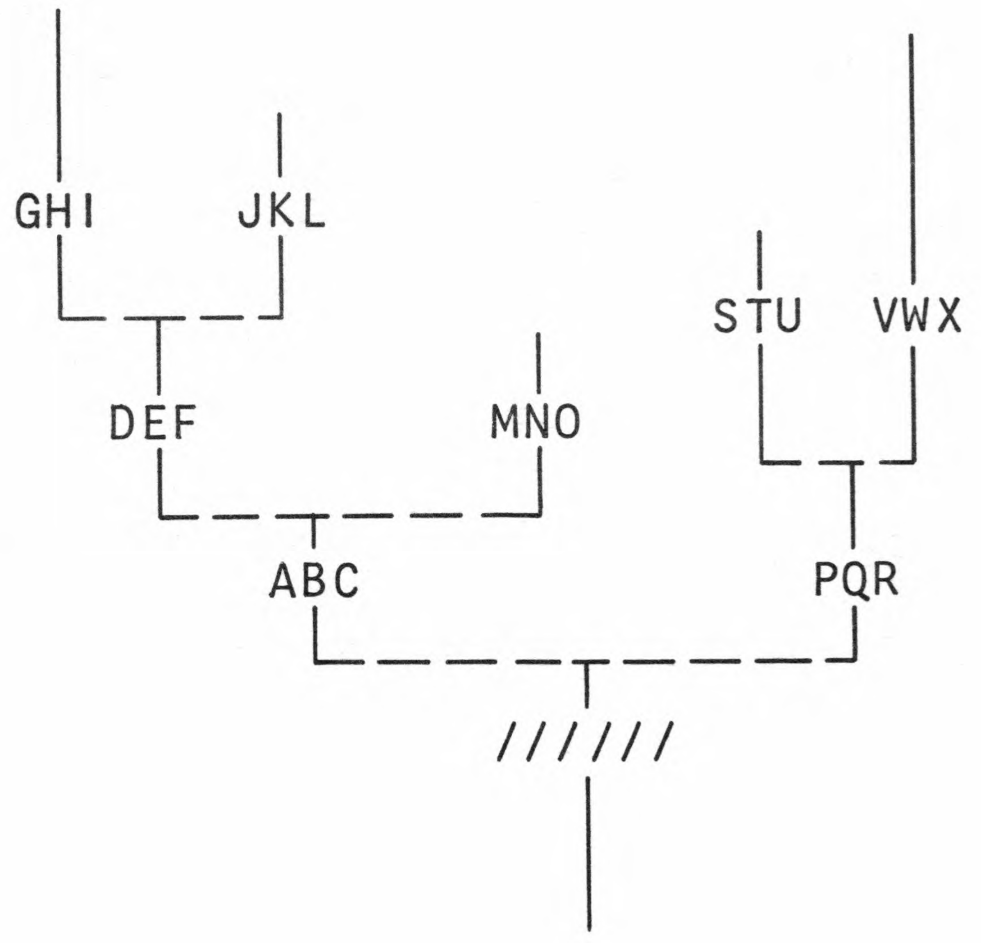
The loader allocates storage based on the $ LINK control card. The allocation of storage can be diagrammed in the following way:



The solid horizontal lines in this diagram indicate the points at which the loader resets its loading origin as a result of encountering a $ lINK control card with two segment names. The broken horizontal lines indicate the end of the shorter overlay segment.

Tree Representation

A tree provides a convenient form of representation for an overlay structure. The tree representation for the structure just described is:



A tree representation makes explicit the notion of the path. A path is a route that can be traced from the root of the tree to one of the tips. The paths of the above tree are:

```
//////-ABC-DEF-GHI
//////-ABC-DEF-JKL
//////-ABC-MNO
//////-PQR-STU
//////-PQR-VWX
```

If two segments are connected by a path, these segments are said to be common to a path. Notice that the root segment, //////, is common to every path of the tree. The segments ABC and JKL are common to a path, namely: the second path on the above list of paths. The segments ABC and VWX are, however, not common to any path of the tree.

References Between Segments

Programs can reference other programs in the same segment or programs contained in segments on a common path with their containing segment. If there is no common path between two segments, the programs of one segment cannot reference the programs of the other segment. Programs of segments not on a common path share storage and are, therefore, not usually in memory at the same time.

Programs of segments on separate paths can communicate with each other through the root segment or through a segment closer to the root segment that is common to both paths. For example, the program S of segment STU cannot communicate directly with the program V of segment VWX, but both S and V can communicate with the program P of segment PQR since there is a path through PQR and STU and through PQR and VWX.

The loader prohibits references between programs belonging to segments on separate paths by removing the names defined in overlaid segments from its symbol table. References to such names therefore become undefined.

It sometimes happens, however, that segments on separate paths co-exist in memory. Consider, in the previous example, the segments GHI and MNO. Clearly, the loading of MNO does not in any way affect the contents of GHI, if GHI is in memory. The NOPAC option of the $ LINK control card allows programs in a segment to reference programs in an overlaid segment. If the program M of segment MNO references the program G of segment GHI, the $ LINK control card defining the segment must be:

```
$       LINK    MNO,DEF,NOPAC
```

The NOPAC option directs the loader to omit the step that removes the names defined in the overlaid segment DEF from the symbol table. These names are then available to the programs of MNO, and the reference from M to G is defined.

Loading Segments

The loader converts each segment into a load module stored on the program link file (H*), which is provided by the system (as a temporary file) if it is not provided by the user. The loader then loads the root segment into main storage and passes control to any entry name within the root segment. The user's program is responsible for loading the overlay segments into main storage as they are needed. Two programs, PLINK and PLLINK, are provided in the PL/I standard library to accomplish this loading. These two programs differ from each other only in the way in which control is returned.

PLINK loads the segment named as its argument from the program link file (H*) and passes control to the entry name defined by the use of the $ ENTRY control card.

PLLINK loads the segment named as its argument from the program link file (H*) and returns control to the statement following the call to PLLINK.

Consider the following portion of an overlay segment:

```
1      8        16
-------------------------------------
                   .
                   .
                   .
$      LINK     XSEG
$      PL1
  A:       PROCEDURE;
            .
            .
            .
           END;
$      PL1
  B:       PROCEDURE;
            .
            .
            .
           END;
$      ENTRY B
$      LINK     YSEG
       .
       .
       .
```

The following two examples illustrate the loading of the segment XSEG, first using PLINK and then using PLLINK.

EXAMPLE USING PLINK

The subprogram PLINK is used to load <u>and</u> transfer control to the entry name B in the segment XSEG in the following example:

```
PROG1:  PROCEDURE;
        .
        .
        .
        DECLARE PLINK ENTRY(CHARACTER(6));
        .
        .
        .
        CALL PLINK('XSEG  ');
        .
        .
        .
        END;
```

EXAMPLE USING PLLINK

The subprogram PLLINK is used to load the segment XSEG in the following example. Control is subsequently transferred to the segment XSEG by the call to the entry name B.

```
PROG2:  PROCEDURE;
        .
        .
        .
        DECLARE PLLINK ENTRY(CHARACTER(6));
        .
        .
        .
        CALL PLLINK('XSEG  ');
        .
        .
        .
        CALL B;
        .
        .
        .
        END;
```

EXAMPLE OF THE USE OF OVERLAYS

The deck setup for an example using overlays is given here. The tree representation defined by the overlay structure and a diagram of the processing of this example by the loader are also included. Note, in this example, that $ ENTRY cards for the overlay segments are not required since only PLLINK is used.

Deck Setup for Example OVLY

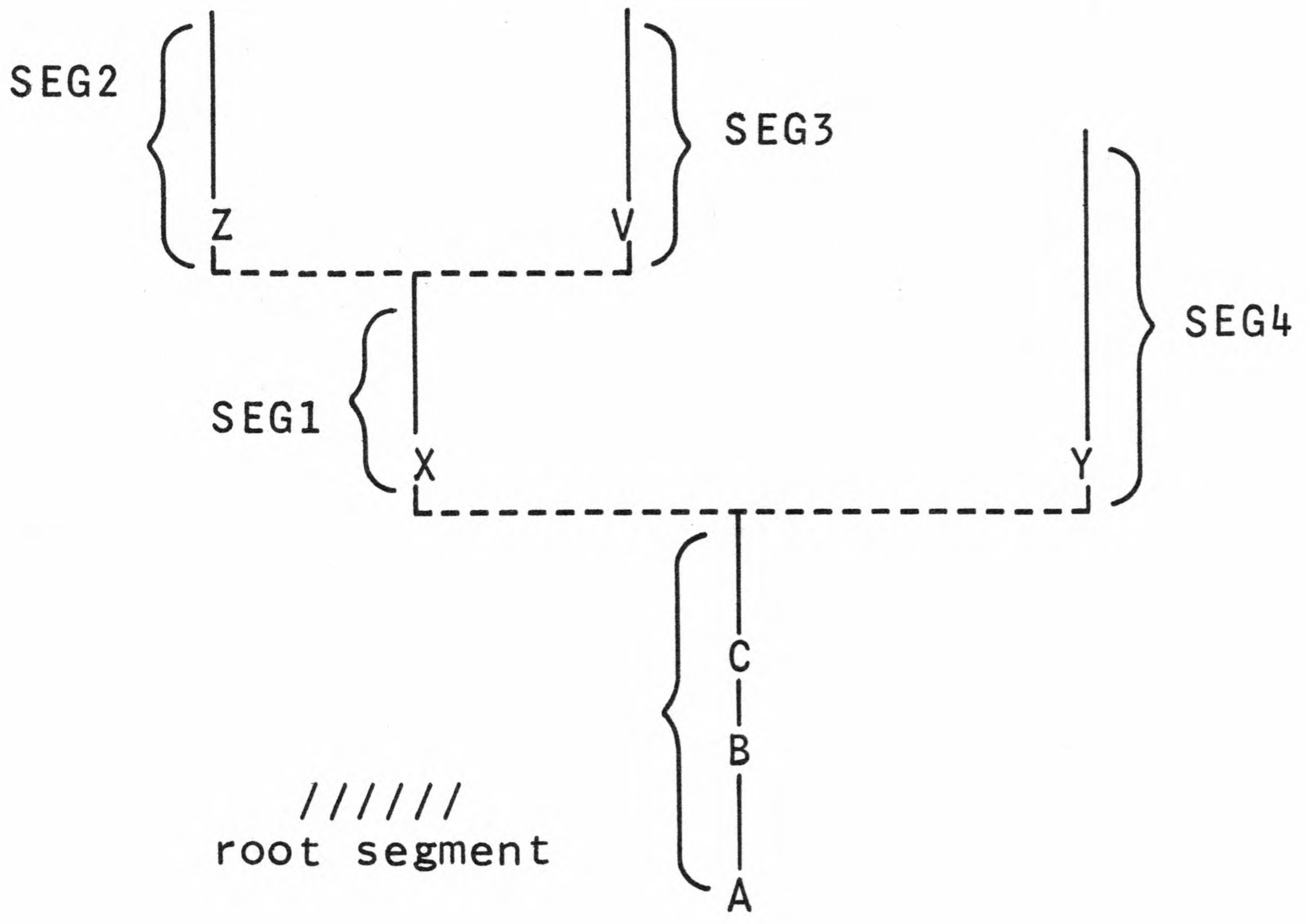
```
1      8       16
$      SNUMB
$      IDENT
$      OPTION  PL1
$      PL1
  A:     PROCEDURE OPTIONS(MAIN);
         DECLARE PLLINK ENTRY(CHARACTER(6));
         .
         .
         .
         CALL PLLINK('SEG1  ');
         CALL X;
         .
         .
         .
         CALL PLLINK('SEG4  ');
         CALL Y;
         .
         .
         .
         END;
$      PL1
  B:     PROCEDURE;
         .
         .
         .
         END;
$      PL1
  C:     PROCEDURE;
         .
         .
         .
         END;
$      ENTRY   A
$      LINK    SEG1
$      PL1
  X:     PROCEDURE;
         DECLARE PLLINK ENTRY(CHARACTER(6));
         .
         .
         .
         CALL PLLINK('SEG2  ');
         CALL Z;
         .
         .
         .
         CALL B;
         .
         .
         .
         CALL PLLINK('SEG3  ');
         CALL V;
         .
         .
         .
         CALL C;
         .
         .
         .
         END;
```

Deck Setup for Example OVLY (cont)

```
1      8       16
$      LINK    SEG2
$      PL1
  Z:     PROCEDURE;
         .
         .
         .
         END;
$      LINK    SEG3,SEG2
$      PL1
  V:     PROCEDURE;
         .
         .
         .
         END;
$      LINK    SEG4,SEG1
$      PL1
  Y:     PROCEDURE;
         .
         .
         .
         CALL B;
         .
         .
         .
         CALL C;
         .
         .
         .
         END;
$      EXECUTE
$      LIMITS
       .
       .
       .
$      ENDJOB
***EOF
```

Tree Representation for OVLY

The overlay structure defined by the input deck of the previous section can be represented by the following tree:



Loader Processing of OVLY

The processing done by the loader in connection with OVLY is diagrammed in Figure 5-2. As in Figure 5-1, the loader inputs control cards and object programs from the loader control file (R*) and object decks, produced as a result of translation, from the object program file (B*). The loader searches the secondary system standard library file (*L) and the system standard library (L*). In addition, this example illustrates the construction of load modules on the program link file (H*).

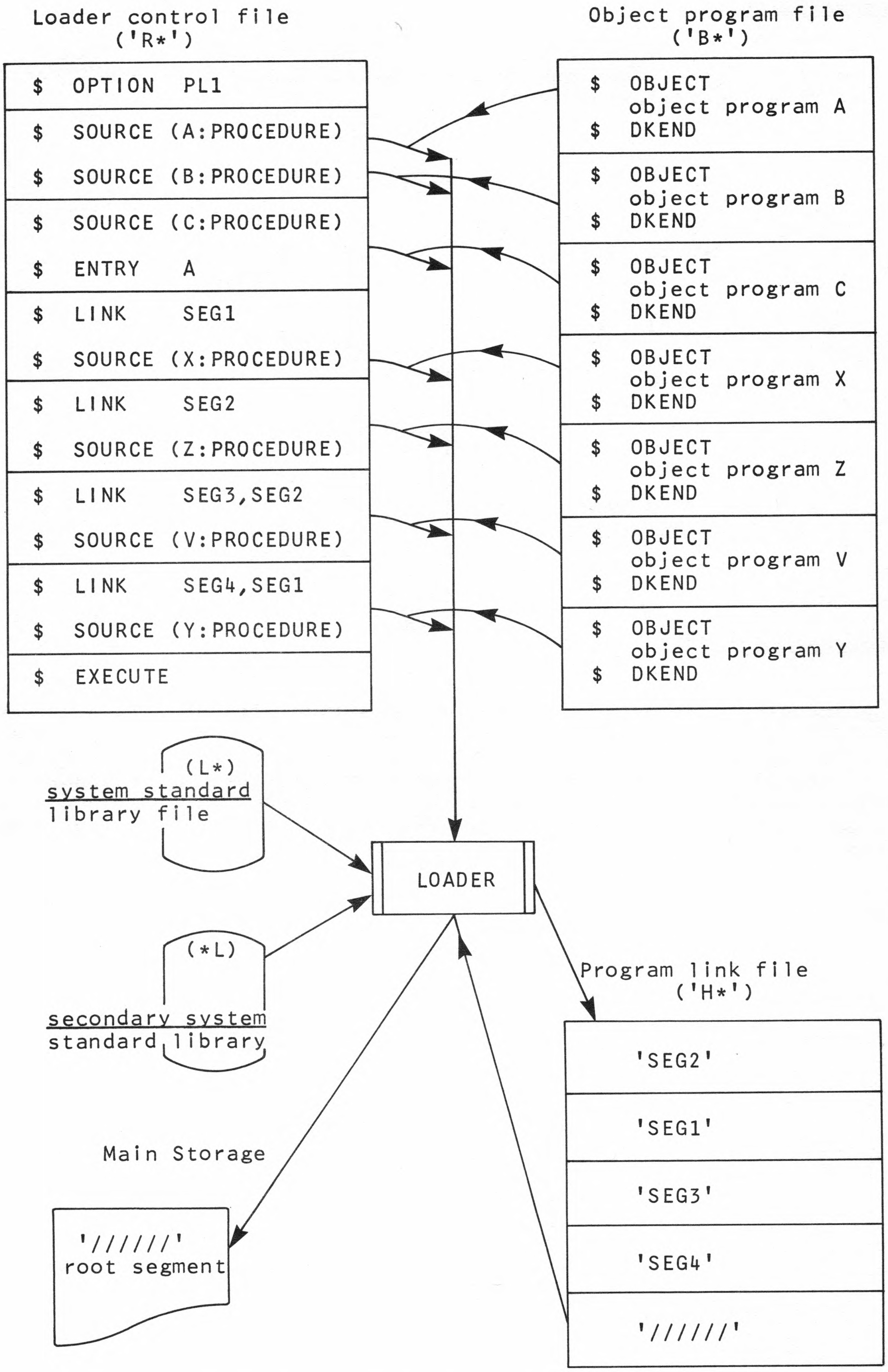


Figure 5-2. Loader Processing of Overlays

SECTION VI

EXTERNAL FILES

This section describes the basic concepts of file processing: organization, access, and transmission. The device assignment control cards and the requirements of the different devices are also included.

FILE ORGANIZATION

GCOS PL/I allows the following four types of file organization:

CONSECUTIVE
INTERACTIVE
INDEXED
REGIONAL keyed

In CONSECUTIVE and INTERACTIVE organization, records are retrieved in the order in which they were written; in INDEXED and REGIONAL organization, records are retrieved by means of a key. The four types of organization are described in detail in the following sections of the manual. For each type of organization, the data transmission statements that can be used are given and the method of attachment to external files is described. Examples of file creation and access are included for each type of organization.

The organization can be specified at compile time by the ENVIRONMENT attribute. If the organization is not given in the program, it can be specified at execution time by a parameter on a control card. In the absence of specification, CONSECUTIVE organization is assumed.

ACCESS MODE

Two types of access, sequential or direct, are available; however, the organization of the file imposes some constraints on the type of access that can be applied to that file. Table 6-1 summarizes the access mode that can be used with each type of organization. The activities permitted for the organization and access are also given.

Table 6-1. Record-Oriented Access Methods

Organization	Access	Activity
CONSECUTIVE	SEQUENTIAL	INPUT OUTPUT UPDATE
INTERACTIVE	SEQUENTIAL	INPUT OUTPUT
INDEXED	SEQUENTIAL	INPUT OUTPUT UPDATE
	DIRECT	INPUT UPDATE
REGIONAL	SEQUENTIAL	INPUT OUTPUT UPDATE
	DIRECT	INPUT OUTPUT UPDATE

As indicated in the table, CONSECUTIVE and INTERACTIVE files cannot be accessed directly and an INDEXED file cannot be opened for DIRECT OUTPUT and thus cannot be created directly. The motivation for these restrictions is given in the detailed description of file organization later in this manual.

TRANSMISSION

PL/I uses two types of transmission: stream-oriented transmission and record-oriented transmission. The PL/I reference manual contains a detailed description of these two transmission methods.

Stream-Oriented Transmission

In stream-oriented transmission, the file is considered to be a continuous stream of characters. However, the conceptual PL/I stream file is attached to an external file that consists of a series of records; consequently, the record size must be provided for stream files. If the LINESIZE option is given in the OPEN statement of an output file, the record size is assumed to be the same as the line size. Otherwise, the record size can be given at execution time on control cards.

There are two PL/I statements for stream-oriented transmission, namely:

GET
PUT

Stream-oriented transmission can be used only with CONSECUTIVE and INTERACTIVE files. The structure of INDEXED and REGIONAL files is predicated upon the relationship between a key and a record.

Stream-oriented transmission can access either BCD or ASCII files. Unless otherwise specified, the file is assumed to be BCD and is converted during transmission to ASCII. Punch stream files are exceptions to this, however (see "Device Requirements" in this section and "Descriptor Files" in the section on "Consecutive and Interactive Organization").

Record-Oriented Transmission

In record-oriented transmission, the minimum unit to be processed is a logical record. No data conversion takes place during transmission.

A file is considered to be a set of logical records. On OUTPUT, a WRITE, REWRITE, or LOCATE statement causes the record to be transmitted to the external file exactly as it is recorded internally. A READ statement causes the record of the external file to be transmitted to memory. The logical records are written on the external file after being blocked by the operating system. Since the records are blocked, the execution of a data transmission statement does not necessarily cause the record to be actually transmitted between memory and a peripheral device. The execution of a CLOSE statement causes any records retained in the blocking buffers to be transmitted to the device.

There are five PL/I statements for record-oriented transmission, as follows:

READ
WRITE
REWRITE
LOCATE
DELETE

The options that can be used in these statements depend upon the type of organization and the method of access. The permissible data transmission statements for each type of organization are given in the sections following.

RECORD STRUCTURE

PL/I handles the following record types:

FIXED
VARIABLE

FIXED records can be used with all types of file organization. VARIABLE record types can be used only with CONSECUTIVE files.

FIXED Records

FIXED records are all of the same defined length. The size of the buffer determines the number of records to be blocked. No record control word appears in fixed length records.

VARIABLE Records

For VARIABLE records, a record control word appears at the beginning of each logical record. VARIABLE records can be used only with files generated in the binary mode. VARIABLE records can contain a record larger than the buffer size. Such a record is called a _partitioned record_. When files with partitioned records are handled, the PRTREC option must be requested on the $ FFILE control card.

ATTACHMENT OF PL/I FILES TO EXTERNAL FILES

The PL/I file is a conceptual unit. When the OPEN statement for the file is executed, the file is attached to an external file by the file code. The _file code_ is determined from the first two characters of the TITLE option. If the TITLE option is not given, the first two characters of the file name are used as the file code. Control cards with the identifying file code are used at execution time to specify a device and to provide additional information about the file.

A CONSECUTIVE file can be attached directly to a peripheral device, if all the default assumptions apply to the file. To change default assumptions, a $ FFILE control card or a descriptor file can be included. INDEXED and REGIONAL files require a descriptor file and a $ USE control card for attachment. Figure 6-1 illustrates the attachment of files with different types of organization. A description of the peripheral device assignment cards is given later in this section. The descriptor file cards depend upon the organization of the file and are, therefore, described separately under each organization type. Similarly, the $ USE is described for INDEXED and REGIONAL files.

In Figure 6-1, the first file F1 is a CONSECUTIVE file with record-oriented transmission. The file F1 is opened with a TITLE option W1. The file code is taken from the first two characters of the TITLE option, so the file code for F1 in this example is W1. All the default assumptions apply to this file; therefore, it can be attached directly by the device assignment card $ TAPE W1. The second file, F2, is a CONSECUTIVE file with stream-oriented transmission. A descriptor file is provided for this file to alter the default assumption about record size. The third file, F3, is a REGIONAL file; therefore, the necessary descriptor file and $ USE control card are provided in addition to the device assignment card for a direct access device. The fourth file, F4, is an INDEXED file; the necessary descriptor file, $ USE control card, and device assignment cards for index and data file are provided. The fifth file, SYSIN, is a standard file and needs no control cards.

```
1      8          16
$      SNUMB
$      IDENT
$      OPTION     PL1
$      PL1

EX1:   PROC;
       DCL F1 RECORD FILE ENVIRONMENT(CONSECUTIVE);
       DCL F2 STREAM FILE;
       DCL F3 RECORD FILE ENVIRONMENT(REGIONAL);
       DCL F4 RECORD FILE;
       DCL SYSIN FILE;    ?
       .
       .
       .
       OPEN FILE(F1)  TITLE('W1') INPUT;
       OPEN FILE(F2)  TITLE('X1') OUTPUT;
       OPEN FILE(F3)  TITLE('Y1') DIRECT UPDATE;
       OPEN FILE(F4)  TITLE('Z1') KEYED SEQUENTIAL INPUT;
       OPEN FILE(SYSIN) INPUT;
       .
       .
       .
       END;

$      USE        .RBUF1/2000/,.RBUF2/2/
$      USE        .XBUF1/3000/,.XBUF2/2/
$      EXECUTE
       .
       .
       .
$      TAPE       W1,...

$      TAPE       A1,...
$      PRMFL      B1,...
$      FILE       C1,...
$      FILE       C2,...
       .
       .
       .
$      DATA       X1
CSP    DATA       FC=A1
CSP    RECORD     CHARSZ=100
       .
       .
       .
$      DATA       Y1
RSP    DATA       FC=B1
RSP    RECORD     RECSZ=40
       .
       .
       .
$      DATA       Z1
ISP    INDEX      FC=C1,PAGESZ=320
ISP    DATA       FC=C2,PAGESZ=320
ISP    RECORD     RECSZ=20.KEYOFF=0,KEYSZ=12
       .
       .
       .
$      ENDJOB
```

Figure 6-1. File Attachment

Device Assignment Control Cards

Device assignment control cards specify the actual device to be used for each file and define additional file information. This section briefly describes seven of these device assignment cards. A more detailed description of these and information about other file control cards is given in the Control Cards Reference Manual.

Device assignment cards must (1) follow the control card that defines the activity, and (2) precede any data cards associated with the activity.

FILE CONTROL CARD

The $ FILE control card allocates a file to a mass storage device. If the device type is not given, the file is allocated to the fastest device type available. The $ FILE control card has the following format:

```
1      8      16
_____________________________________

$      FILE   fc,lud,access,device-list
```

where: fc — is the 2-character alphanumeric file code identifying the external file.

lud — is the logical unit designator, a 2- or 3-character symbol (followed by a disposition code) identifying the file. The first character of the identifier is alphanumeric and the remaining characters numeric. The following disposition codes can be given:

R - Release
S - Save for subsequent activity
P - Purge

access — indicates the number of links and the file type: sequential (L) or random (R).

device-list — specifies a device type preference list for the allocation of mass storage. The device types that can be given are:

```
DSS167  )
DSS170  )
DSS180  )
DSS181  ) (DSPK)
DSS190  )
DSS191  )
DSS270  )
MSU0310 ) (MASS)
MSU0400 )
```

Consider, for example, the following $ FILE control card:

```
1      8       16
$      FILE    AA,X1S,2L,DSS167,DSS180
```

This card requests that the external file with file code AA be assigned to DSS167 and, if that device is not available, to DSS180. The file is accessed sequentially and occupies two links (2L). The file is to be saved (S) for a subsequent activity.

PRMFL CONTROL CARD

The $ PRMFL control card is used to access an existing permanent file. The $ PRMFL card has the following format:

```
1      8       16
$      PRMFL   fc,permit,type,file-string
```

where: fc is the 2-character file code identifying the file.

permit is the allowable access, as follows:

R	- Read
W	- Write
A	- Append
E or X	- Execute
REC	- Recovery

type indicates sequential (L) or random (R).

file-string is the file descriptor. It contains the catalog name, password (if needed), and file name.

Consider, for example, the following $ PRMFL control card:

```
1      8       16
$      PRMFL   H*,W,R,ALPHA/CW
```

This card requests the permanent file created by FILSYS on H*. The requested access is append (A). The file is random (R), and the file string is ALPHA/CW.

TAPE CONTROL CARDS

These cards assign tape units. Three tape control cards are available:

$ TAPE7 assigns a seven-track tape unit.
$ TAPE9 assigns a nine-track tape unit.
$ TAPE meaning may be installation dependent. See the Control Cards Reference Manual.

The format of the tape control cards is as follows:

```
1      8         16
_____________________________________________________

$      TAPE      fc,lud,mri,serial,seq,file-name
       TAPE7
       TAPE9
```

where: fc is the 2-character alphanumeric file code identifying the file.

lud is the logical unit designator, a 2- or 3-character symbol identifying the file and a disposition code. The following disposition codes can be given:

S - Save
C - Continue
D - Dismount
R - Release
P - Purge

mri is the multireel indicator. Any nonblank character in this field indicates a second tape is assigned to the file.

serial is the tape serial number of the first reel of the file.

seq is the sequence number of the reel at which processing begins.

file-name is a 1- to 12-character name given for external identification of the file; this name is used to issue mounting instructions to the operator.

SYSOUT CONTROL CARD

The $ SYSOUT control card assigns the file identified by the file code to SYSOUT for online conversion. The $ SYSOUT control card has the following format:

```
1      8         16
_____________________________________________________

$      SYSOUT    fc
```

READ CONTROL CARD

The $ READ control card allocates the file identified by the file code to the card reader. The $ READ control card has the following format:

```
1      8         16
_____________________________________________________

$      READ      fc
```

PRINT CONTROL CARD

The $ PRINT control card allocates the file identified by the file code to the line printer. The $ PRINT control card has the following format:

```
1      8      16
$      PRINT  fc
```

PUNCH CONTROL CARD

The $ PUNCH control card allocates the file identified by the file code to the card punch. The $ PUNCH control card has the following format:

```
1      8      16
$      PUNCH  fc
```

Device Requirements

Table 6-2 summarizes the device requirements of different devices and indicates the type of organization that can be applied to that device. In addition, the transmission method and mode for the device are given.

Table 6-2. Device Requirements

Device Type	Allowable Organization	Transmission Method	Mode
card reader	CONSECUTIVE	stream-oriented or record-oriented with fixed length records	BCD (stream) binary (record)
card punch	CONSECUTIVE	stream-oriented or record-oriented with fixed length records	BCD or IBMEL (stream) binary (record)
line printer	CONSECUTIVE	stream-oriented or record-oriented with fixed length records	BCD
magnetic tape	CONSECUTIVE	record-oriented stream-oriented	ASCII ASCII or BCD
mass storage	CONSECUTIVE	record-oriented stream-oriented	ASCII ASCII or BCD
	INDEXED	record-oriented	binary
	REGIONAL	record-oriented	binary

SECTION VII

CONSECUTIVE AND INTERACTIVE ORGANIZATION

This section describes the attachment of files with CONSECUTIVE and INTERACTIVE organization. The general requirements for the attachment of a file with CONSECUTIVE organization are followed by examples of the creation and access of CONSECUTIVE stream files and record files. The special requirements of INTERACTIVE files and an example of the creation and access of an INTERACTIVE file concludes the section.

CONSECUTIVE ORGANIZATION

A CONSECUTIVE file can be accessed only in the order in which it was written. For devices like the card reader, punch, and line printer, this is the only acceptable form of organization.

A CONSECUTIVE file can be attached to an external file directly if all the default assumptions apply. If a CONSECUTIVE file is attached to a direct access device, the SEQUENTIAL file option must be specified. To alter the default assumptions for a CONSECUTIVE file, the $ FFILE control card and/or a descriptor file can be specified.

Attachment of a CONSECUTIVE File

To specify and attach a CONSECUTIVE file, the following requirements must be met:

- The file must be designated as CONSECUTIVE. The CONSECUTIVE keyword can be specified in the ENVIRONMENT attribute at compile time or a descriptor file containing CSP cards can be supplied at execution time. In the absence of the ENVIRONMENT attribute and a descriptor file, CONSECUTIVE organization is assumed.

- The file must be assigned to a peripheral device by a $ TAPE, $ FILE, $ PRMFL, $ SYSOUT, $ READ, $ PUNCH or $ PRINT control card.

- To override the default assumptions about buffer size, number of buffers, mode, or record type, the $ FFILE control card can be used. The default assumptions are:

buffer size	320 words
number of buffers	1
record length	variable
mode	binary

- To override the default assumptions about record size, tabs, mode, and rewinding, a descriptor file of CSP cards can be provided. The default assumptions are given with the explanation of the descriptor file, later in this section.

$ FFILE CONTROL CARD

For files with CONSECUTIVE organization, the file control block can be created using the $ FFILE control card. The format of the $ FFILE control card is as follows:

```
1      8      16
$      FFILE  fc,option,...
```

where: fc is the 2-character alphanumeric code identifying the file.

options describe the nonstandard properties of the file.

The options of interest to the PL/I programmer are given in the following list:

Option	Meaning
STDLBL	A standard label is generated and checked.
NSTDLB	No label is generated.
NBUFFS/n	The number of buffers to be used is n, (n = 1 or 2).
BUFSIZ/n	The size of the buffer is n, where n is a decimal number $\leq$ 4095.
MODBCD or MBCD	The recording mode is BCD.
MODMIX	The recording mode is mixed (BCD and binary).
FIXLNG/n	The file contains fixed length records of length n, where n $\leq$ 4095.
PRTREC	The file contains partitioned records.

DESCRIPTOR FILE FOR A CONSECUTIVE FILE

CONSECUTIVE files can be more fully specified by the use of a descriptor file. The descriptor file contains information about the rewinding of the file, the character set, format, and record size.

Two types of control cards are used to provide information about files with CONSECUTIVE organization: the CSP DATA card and the CSP RECORD card. Columns 1 - 3 of these cards contain the code CSP to indicate that the cards apply to a CONSECUTIVE file. The format of the CSP DATA card is as follows:

```
1      8      16
CSP    DATA   FC=fc,option,...
```

where: fc is the 2-character alphanumeric code identifying the file.

option provides additional information.

The options that can be used on a CSP DATA card are given in the following list:

Option	Meaning	Default
OLEAVE	Open file without rewinding.	Rewind on opening.
LEAVE	Close file without rewinding.	Rewind on closing.
LOCK	Lock file.	Do not lock file.
ASCII	File consists of ASCII characters.	Stream file is BCD.
BCD	Punch stream file consists of BCD.	Punch stream file as IBMEL (see appendix on "Character Conversion Tables").
TAB	Set tabs at specified columns, i.e., TAB(1,15,19)	TAB(1,11,21,...131)
NTAB	Print data continuously	TAB(1,11,21,...131)
INTERACTIVE	File is INTERACTIVE.	File is CONSECUTIVE.

The CSP ETC control card can be used to continue the CSP DATA card.

The CSP RECORD control card has the following format:

```
1      8      16
CSP    RECORD option
```

where: option indicates the size of the record, as follows:

RECSZ=n The logical record contains a maximum of n words.

CHARSZ=n The logical record contains a maximum of n characters.

EXAMPLE OF CONSECUTIVE FILE ATTACHMENT

The following fragment illustrates the attachment of a file with CONSECUTIVE organization.

```
1      8       16
$      SNUMB
$      IDENT
$      OPTION  PL1
$      PL1

EX1:     PROC;
         .
         .
         .
         OPEN FILE(F1) OUTPUT TITLE('X1') STREAM;
         .
         .
         .
$      FFILE   A1,BUFSIZ/400,NBUFFS/2
$      TAPE    A1,A1D
$      DATA    X1
CSP    DATA    FC=A1
CSP    RECORD  CHARSZ=100

       .
       .
       .
$      ENDJOB
```

The TITLE option in the OPEN statement specifies the file code X1. A descriptor file is included following the $ DATA control card with the file code X1. The descriptor file specifies that the file code is A1 and that the size of the records of the file is 100 characters. A $ FFILE control card is included to alter the default assumptions about the number of buffers and the buffer size.

Stream-Oriented Transmission

Stream-oriented transmission can be applied to files with CONSECUTIVE organization. Although a stream file is a continuous sequence of characters, it is attached to an external file that consists of a series of records. The record size of the external file is specified either by the LINESIZE option in the OPEN statement (for an output file) or directly on a CSP card.

EXAMPLES OF STREAM FILE ACCESS

Figure 7-1 illustrates the creation of a stream file. Data is taken from the system input file and placed in the stream file MASTER. The file MASTER is attached to an external tape file with records 80 characters long.

Figure 7-2 illustrates stream file access. The file MASTER created in the previous figure is opened, and those entries belonging to the engineering department are printed on the system output file.

```
1      8       16
$      SNUMB
$      IDENT
$      OPTION  PL1
$      PL1

SFC:   PROC OPTIONS(MAIN);
       DCL MASTER STREAM FILE ENVIRONMENT(CONSECUTIVE);
       DCL SYSIN FILE;
       DCL 01 DIRECTORY,
              02 PLANT         CHAR(16),
              02 DEPARTMENT    CHAR(16),
              02 SECTION       CHAR(16),
              02 NAME,
                 03 LAST       CHAR(16),
                 03 FIRST      CHAR(16);
       ON ENDFILE(SYSIN)   GOTO EXIT;
       OPEN FILE(MASTER) OUTPUT LINESIZE(80) TITLE("MF");
INSF:  GET LIST(PLANT,DEPARTMENT,SECTION,LAST,FIRST);
       PUT FILE(MASTER) LIST(DIRECTORY);
       GOTO INSF;
EXIT:  CLOSE FILE(MASTER);
       END;

$      EXECUTE
$      LIMITS  10,40K,-2K
$      TAPE    SC,X1D
$      DATA    I*
CLEVELAND ENGINEERING  33B JONES WALTER
CLEVELAND PURCHASING   24C SMITH HENRY
CLEVELAND PURCHASING   24D MARTIN JOSEPH
PHILADELPHIA PLANNING 224  FRANKLIN ROBERT
PHILADELPHIA ENGINEERING 335 GEORGE WALTER
WASHINGTON MARKETING  AA45 JENSON THOMAS
PHILADELPHIA ENGINEERING 336 SMITH CLYDE
ALBANY PURCHASING XX22156 BURR ARTHUR
ALBANY ENGINEERING XX223457 HAMILTON NATHAN
$      DATA    MF
CSP    DATA    FC=SC
CSP    RECORD  CHARSZ=80
$      ENDJOB
***EOF
```

Figure 7-1. CONSECUTIVE Stream File Creation

```
1      8       16
$      SNUMB
$      IDENT
$      OPTION  PL1
$      PL1

SFA:   PROC OPTIONS(MAIN);
       DCL MASTER STREAM FILE ENVIRONMENT(CONSECUTIVE);
       DCL  SYSPRINT FILE;
       DCL 01 DIRECTORY,
              02 PLANT        CHAR(16),
              02 DEPARTMENT   CHAR(16),
              02 SECTION      CHAR(16),
              02 NAME,
                 03 LAST      CHAR(16),
                 03 FIRST     CHAR(16),
       ON ENDFILE(MASTER) GOTO EXIT;
       OPEN FILE(MASTER)  INPUT;
LOOP:  GET FILE(MASTER) LIST(DIRECTORY);
       IF DEPARTMENT = "ENGINEERING"
            THEN PUT SKIP LIST(DIRECTORY);
       GOTO LOOP;
EXIT:  CLOSE FILE(MASTER);
       END;

$      EXECUTE
$      LIMITS  10,40K,-2K,20000
$      TAPE    SA,X1D
$      DATA    MA
CSP    DATA    FC=SA
CSP    RECORD  CHARSZ=80
$      ENDJOB
***EOF
```

Figure 7-2. CONSECUTIVE Stream File Access

Record-Oriented Transmission

Records of a CONSECUTIVE file have no key and are retrieved in the order in which they are written. A CONSECUTIVE file is created by the execution of a sequence of WRITE statements with the SEQUENTIAL OUTPUT attribute. Once the file is created, it can be accessed by READ statements with the SEQUENTIAL INPUT or SEQUENTIAL UPDATE attributes. The REWRITE statement cannot be used for a file with CONSECUTIVE organization.

DATA TRANSMISSION STATEMENTS

The data transmission statements that can be used to create and access a CONSECUTIVE RECORD file are given in Table 7-1. Braces are used to group alternative forms, each written on a separate line. Brackets are used to indicate a construct that is optional.

Table 7-1. Data Transmission Statements for CONSECUTIVE RECORD Files

SEQUENTIAL OUTPUT
WRITE FILE(file-name) FROM(variable-name); LOCATE based-var FILE(file-name   SET(pointer-var) ;
SEQUENTIAL INPUT or SEQUENTIAL UPDATE
READ FILE(file-name) $\left\{\begin{array}{l}\text{INTO(variable-name)}\\ \text{SET(pointer-var)}\\ \text{IGNORE(expression)}\end{array}\right\}$;

EXAMPLES OF RECORD FILE ACCESS

Figure 7-3 illustrates the creation of a CONSECUTIVE RECORD file and Figure 7-4 illustrates the access of the file just created.

```
1      8         16
$      SNUMB
$      IDENT
$      USERID    SMCNAME$PASSWORD
$      OPTION    PL1
$      PL1       LIST

CFC:   PROC OPTIONS(MAIN);
       DCL DISC RECORD SEQUENTIAL FILE ENVIRONMENT(CONSECUTIVE);
       DCL SYSIN FILE;
       DCL 01 IMAGE,
              02 NAME,
                 03 LAST              CHAR(30),
                 03 FIRST             CHAR(30),
              02 CITY                 CHAR(30),
              02 STATE                CHAR(26),
              02 CODE                 CHAR(4);
       ON ENDFILE(SYSIN) GOTO EXIT;
       OPEN FILE(DISC) OUTPUT;
LOOP:  GET LIST(LAST,FIRST,CITY,STATE,CODE);
       WRITE FILE(DISC) FROM(IMAGE);
       GOTO LOOP;
EXIT:  CLOSE FILE(DISC);
       END;

$      EXECUTE
$      LIMITS    10,20K,-2K
$      PRMFL     DF,W,S,DATA/BANK
$      DATA      DI
CSP    DATA      FC=DF
CSP    RECORD    RECSZ=30
$      DATA      I*
JONES ROBERT PHILADELPHIA PENNSYLVANIA AA
SMITH HENRY WAKEFIELD OHIO AB
SMITH ROBERT STONEHAM CALIFORNIA  AA
SMITH MARY STONEHAM CALIFORNIA   BA
SMITH CHARLES RANDOLPH MARYLAND AA
SMITH MARTIN SHARON WASHINGTON BA
SMITH CHARLES NORWOOD FLORIDA BA
$      ENDJOB
***EOF
```

Figure 7-3. CONSECUTIVE RECORD File Creation

```
1      8       16
$      SNUMB
$      IDENT
$      USERID  SMCNAME$PASSWORD
$      OPTION  PL1
$      PL1     LIST

CFA:   PROC OPTIONS(MAIN);
       DCL (DISC,ATAPE) RECORD SEQUENTIAL FILE ENVIRONMENT(CONSECUTIVE);
       DCL 01 IMAGE,
              02 NAME,
                 03 LAST     CHAR(30),
                 03 FIRST    CHAR(30),
              02 CITY        CHAR(30),
              02 STATE       CHAR(26),
              02 CODE        CHAR(4);
       ON ENDFILE(DISC) GOTO EXIT;
       OPEN FILE(DISC) INPUT;
       OPEN FILE(ATAPE) OUTPUT;
LOOP:  READ FILE(DISC) INTO(IMAGE);
       IF CODE = "AA" THEN WRITE FILE(ATAPE) FROM(IMAGE);
       GOTO LOOP;
EXIT:  CLOSE FILE(DISC);
       CLOSE FILE(ATAPE);
       END;

$      EXECUTE
$      LIMITS  10,20K,-2K
$      TAPE    TF,X1S
$      FFILE   TF,NBUFFS/2,BUFSIZ/640
$      PRMFL   DF,R,S,DATA/BANK
$      DATA    DI
CSP    DATA    FC=DF
CSP    RECORD  RECSZ=30
$      DATA    AT
CSP    DATA    FC=TF
CSP    RECORD  RECSZ=30
$      ENDJOB
***EOF
```

Figure 7-4. CONSECUTIVE RECORD File Access

INTERACTIVE ORGANIZATION

To communicate with a remote terminal in the DIRECT PROGRAM ACCESS mode, a stream file with INTERACTIVE organization is used.

Attachment of an INTERACTIVE File

To specify and attach a file with INTERACTIVE organization, the following requirements must be met:

- The file must be designated as INTERACTIVE. The INTERACTIVE keyword can be specified in the ENVIRONMENT attribute at compile time or the INTERACTIVE attribute can be specified on the CSP DATA card at execution time.

- A $ USE .RTYP control card must be included before the $ EXECUTE control card to cause the loading of the proper File and Record Control routine for accessing the terminal.

- A $ DAC control card must be included to provide direct access capability between a remote terminal and a program in execution.

The $ DAC control card contains the file code and a single character that is to be appended to SNUMB identifier to provide an inquiry name.

Example of INTERACTIVE Files

Figure 7-5 gives a program fragment illustrating the attachment of two INTERACTIVE files. The inquiry name for this example is '12345I'.

```
1      8          16
______________________________________________________________

$      SNUMB      12345
$      IDENT
$      OPTION     PL1
$      PL1

IFAC:  PROC;
       DCL D1 STREAM FILE ENVIRONMENT(INTERACTIVE);
       DCL D2 STREAM FILE ENVIRONMENT(INTERACTIVE);
       .
       .
       .
       OPEN FILE(D1) INPUT TITLE("AB");
       OPEN FILE(D2) OUTPUT LINESIZE(120);
       .
       .
       .
       GET FILE(D1) LIST(X);
       .
       .
       .
       PUT FILE(D2) LIST(Y);
       .
       .
       .
       END;

$      USE        .RTYP
$      EXECUTE
$      DAC        X1,I
$      DAC        X2,I
$      DATA       AB
CSP    DATA       FC=X1,INTERACTIVE
CSP    RECORD     RECSZ=10
$      DATA       D2
CSP    DATA       FC=X2,INTERACTIVE
CSP    RECORD     CHARSZ=120
$      ENDJOB
***EOF
```

Figure 7-5. Attachment of INTERACTIVE Files

SECTION VIII

INDEXED ORGANIZATION

This section describes the access and structure of files with INDEXED organization. The method of attachment for an INDEXED file and the utilization report produced as a result of using an INDEXED file are given. Examples of the creation and access of an INDEXED file are included.

INDEXED files are processed by the Index Sequential Processor (ISP) in GCOS. For additional information on INDEXED files, refer to the Indexed Sequential Processor manual.

INDEXED FILE ACCESS

A file with INDEXED organization consists of a series of records, each containing an imbedded key. The imbedded key is a character string within the record. The length of the imbedded key and the position of the key within the record are specified on a control card at execution time. The maximum length for a key is 32 characters.

File Creation

Records in an INDEXED file are arranged in the order defined by the imbedded keys. To create an INDEXED file, the file is opened for SEQUENTIAL OUTPUT and records are written so that the imbedded keys are in order with respect to the ASCII collation sequence. The execution of a WRITE statement during file creation for a record whose imbedded key is lower in the ASCII collation sequence than the key of a previously written record causes the KEY condition to be raised.

Once a file is created, additional records can be inserted. The file is opened for UPDATE and records are logically inserted in the file according to the position of its imbedded key in the ASCII collation sequence with respect to the keys of the other records of the file. The structure of the file is described later in this section.

File Access

An INDEXED file can be accessed either sequentially or directly. Sequential processing accesses records in the order defined by the imbedded keys. Direct processing accesses a record by matching the source key from the data transmission statement to an imbedded key within the file.

SEQUENTIAL ACCESS OF AN INDEXED FILE

When an INDEXED file is accessed sequentially, the source key is not required. Records are accessed in the order of the imbedded keys. If no records have been added to the file since its creation, the order of the imbedded keys is the same as the order in which the records were written. However, if a record has been added with an imbedded key lower in the ASCII collation sequence than the key of the last record of the file, the order of the imbedded keys is different from the order in which the records were written.

For example, if a file is created with records having keys:

A, B, D, F, I, P, T, X

and then an additional record is added with the key G, the order of the imbedded keys is:

A, B, D, F, G, I, P, T, X

Sequential processing of the file retrieves the records in the above order.

In SEQUENTIAL UPDATE, the execution of a DELETE statement without the KEY option causes the most recently retrieved record to be eliminated. If the FROM option does not appear in a REWRITE statement, the execution of that statement causes the record just retrieved to be replaced.

DIRECT ACCESS OF AN INDEXED FILE

All direct access data transmission statements must include either the KEY or the KEYFROM option. Files can be opened either for INPUT or UPDATE.

In direct mode, records can be read, replaced, eliminated, or added. The source key must be specified on the data transmission statement. The source key is compared against the imbedded keys of the file, using the rules that govern character string comparison. If no match is found for the source key specified with a READ, REWRITE or DELETE statement, the KEY condition is raised. If a match is found for the source key specified with a WRITE statement, the KEY condition is also raised, indicating that the key of the record to be added is already contained in the file.

Data Transmission Statements for INDEXED Files

Table 8-1 lists the data transmission statements that can be used with INDEXED files.

Braces are used to group alternative forms, each written on a separate line. Brackets are used to indicate a construct that is optional.

Table 8-1. Data Transmission Statements for INDEXED Files

```
SEQUENTIAL OUTPUT

  WRITE FILE(file-name) FROM(variable-name) [KEYFROM(expression)] ;
  LOCATE based-var FILE(file-name)[KEYFROM(expression)] [SET(pointer-var)];

SEQUENTIAL INPUT

                         { INTO(variable-name) }  [ { KEY(expression)       } ]
  READ FILE(file-name)   {                     }  [ {                       } ] ;
                         { SET(pointer-var)    }  [ { KEYTO(char-strng-var) } ]

  READ FILE(file-name)     IGNORE(expression);

SEQUENTIAL UPDATE

                         { INTO(variable-name) }  [ { KEY(expression)       } ]
  READ FILE(file-name)   {                     }  [ {                       } ] ;
                         { SET(pointer-var)    }  [ { KEYTO(char-strng-var) } ]

  READ FILE(file-name)     IGNORE(expression)  ;
  REWRITE FILE(file-name)[FROM(variable-name)] ;
  DELETE FILE(file-name)  ;

DIRECT INPUT

  READ FILE(file-name)     INTO(variable-name)  KEY(expression)  ;

DIRECT UPDATE

  READ FILE(file-name)     INTO(variable-name)  KEY(expression)  ;
  REWRITE FILE(file-name)  FROM(variable-name)  KEY(expression)  ;
  WRITE FILE(file-name)    FROM(variable-name)  KEYFROM(expression)  ;
  DELETE FILE(file-name)   KEY(expression)  ;
```

STRUCTURE OF AN INDEXED FILE

An INDEXED file consists of two separate files, namely: a data file, containing the written records of the INDEXED file, and an index file, containing information about the position of the keys within the data file, for efficient access. These two files are separate and can be stored on separate direct access devices.

Pages

The data file and the index file, like all direct access files, are divided into pages. A page is the unit of information passed between random access storage and main memory during processing. The page size and the percentage of the page to be filled can be specified on control cards at execution time.

Relationship Between the Data File and the Index File

For every page in the data file, an entry exists in the index file, called a fine index. When the fine index exceeds one page, a coarse index is built. The coarse index portion of the index file contains an entry for every page in the fine index portion.

When an INDEXED file is accessed directly, the index file is used to efficiently locate the desired record in the data file, as follows:

1. The source key from the data transmission statement is compared to the entries in the coarse index to obtain the page number in the fine index.

2. The source key is compared to the entries on the designated fine index page to obtain the page number in the data file.

3. The source key is compared to the keys on the designated data file page to obtain the desired record.

Structure of the Data File

The data file begins with a record containing 62 words of control information and concludes with an end-of-file. The data file contains the records written; each record contains, in addition to the key and data, a record control word and a pointer. The record control word specifies the record length, record type, and deletion status. The pointer specifies the next logical record according to the order of the imbedded keys.

Records can be variable in length; but the key must be located at the same position in every record. The pages of the data file are filled with records. If there is not sufficient space on a page to accommodate an entire record, the space is left unused and a new page is started.

The pages of the data file that are not filled with records during file creation are called overflow pages. These overflow pages can be used for records added after file creation.

Space for the addition of records can be reserved uniformly throughout the file by specifying a percentage fill figure at execution time. The percentage fill parameter is described later in this section in connection with the descriptor file. The uniform distribution of space throughout the file is useful if records with keys distributed throughout the file are to be added after file creation.

Figure 8-1 illustrates the structure of the data file. Record #3 was added to the file after file creation and is, therefore, stored physically on an overflow page and linked into its logical position within the file. If space had been reserved uniformly throughout the file, this record could possibly have been located physically on the page to which it logically belongs.

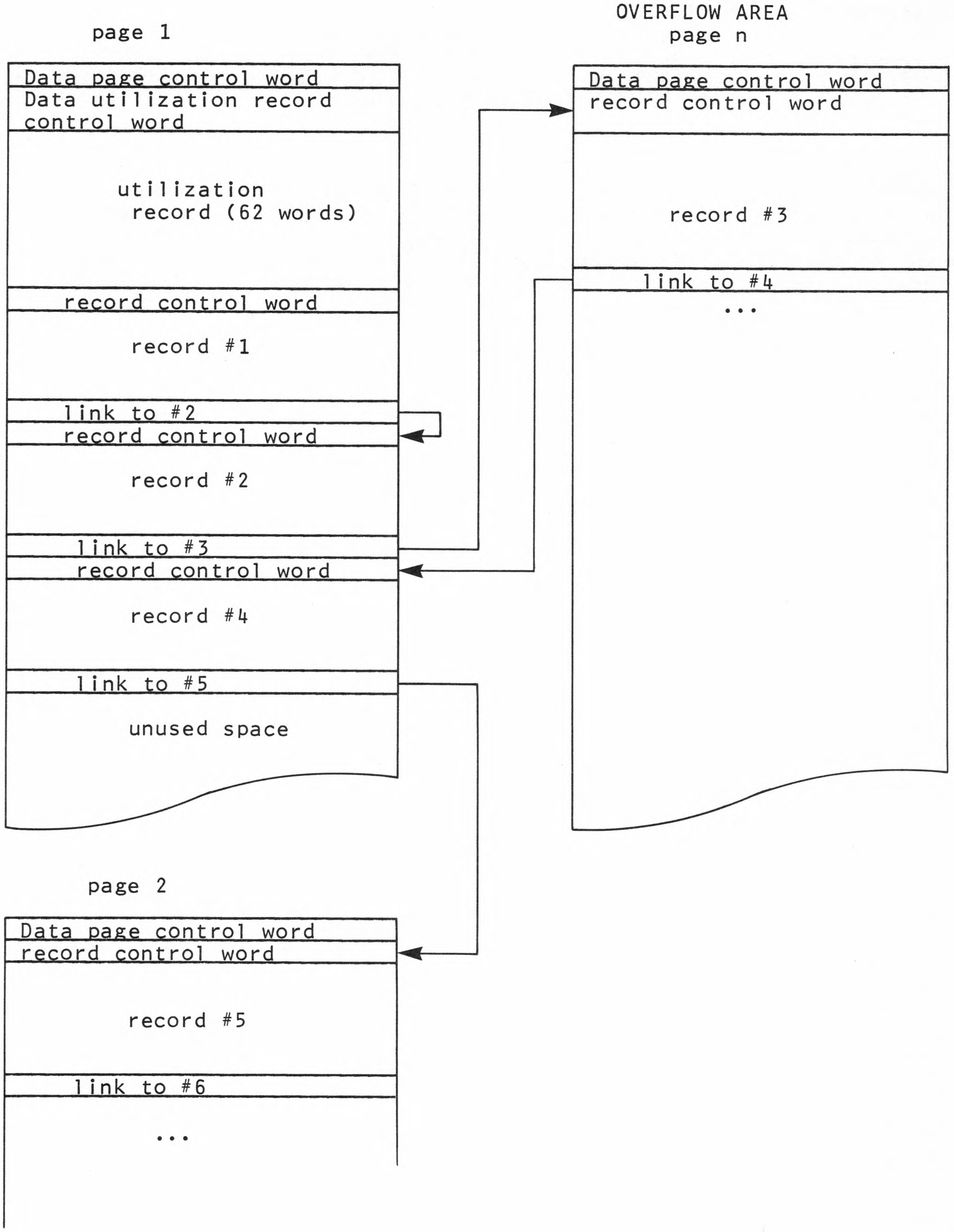
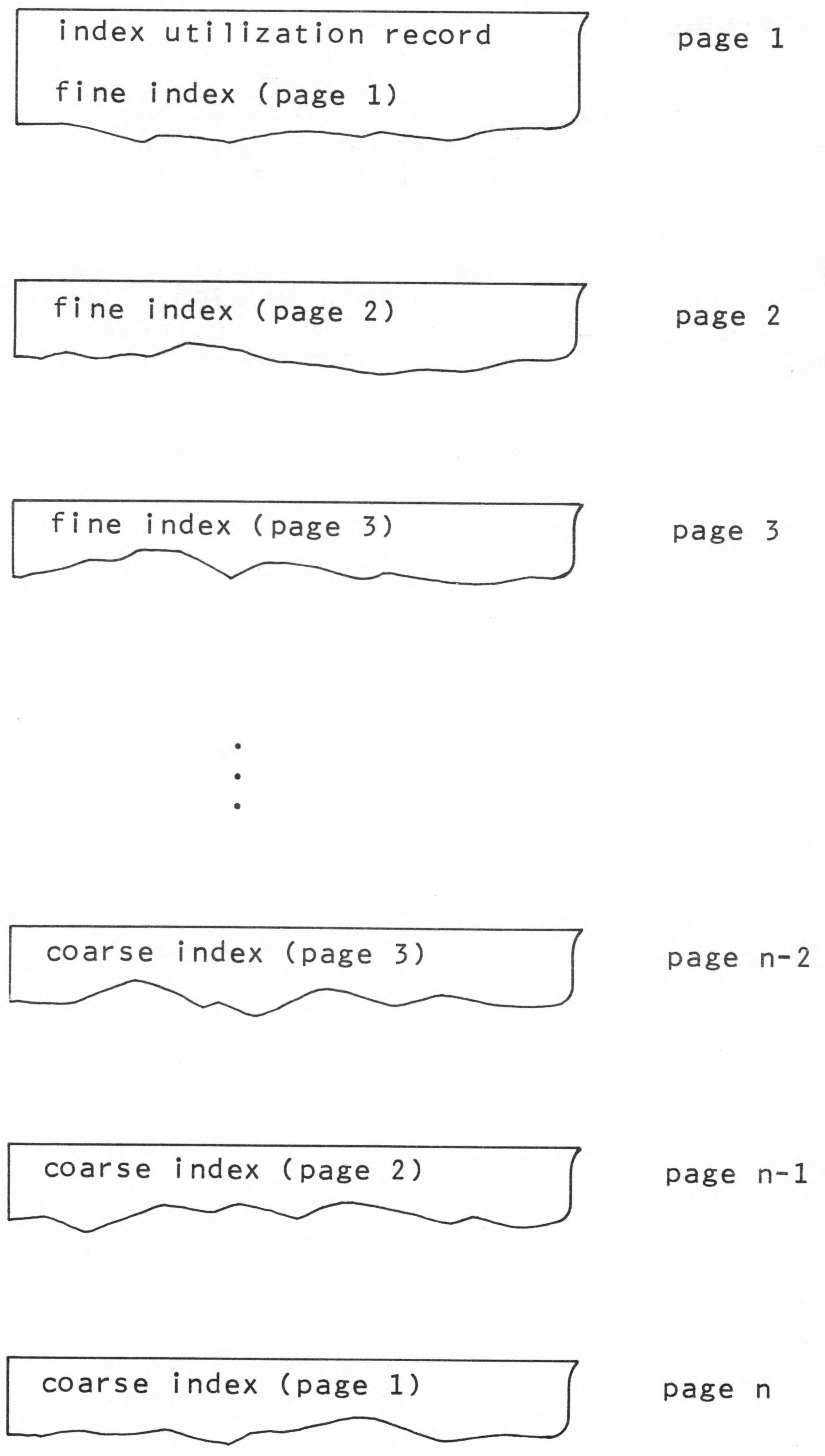


Figure 8-1. Structure of the Data File

Structure of the Index File

The index file begins with a record containing 64 words of control information and concludes with an end-of-file. The fine index is created in ascending order from the beginning of the file. When the fine index portion exceeds one page, the coarse index is created in descending order from the end of the file. If the fine index and the coarse index overlap, an error message is produced and the program is aborted. Guidelines for determining the size of the index file are given later in this section. The size of the index file is related to the page size of the data file; the larger the page size in the data file, the fewer fine index entries in the index file.

Figure 8-2 illustrates the structure of the index file.



n = number of pages allocated to the index file.

Figure 8-2. Structure of the Index File

ATTACHMENT OF AN INDEXED FILE

To specify and attach an INDEXED file the following requirements must be met:

- The file must be designated as an INDEXED file. The INDEXED keyword can be specified in the ENVIRONMENT attribute at compile time or, if no ENVIRONMENT attribute is given, a descriptor file containing ISP cards can be supplied at execution time.
- A descriptor file must be provided to specify the file codes of the data file and index file and the record and key sizes.
- A device assignment control card for a direct access device must be provided for both the data and the index files.
- The size of the work region to be reserved for INDEXED files must be specified on an appropriate $ USE control card.

Descriptor File for an INDEXED File

Each INDEXED file must have an associated descriptor file that specifies the file code of the data file and the file code of the index file. In addition, the maximum record size and key size must be given.

Optionally, the page size and percentage fill for both the data file and the index file can be specified. In the absence of specification, the page size is assumed to be 320 words with 100% fill.

If the key is not located at the beginning of the record, the key offset must be specified. Records can be variable in length, but the offset of the key within the record must be fixed.

The format of the cards in the descriptor file is now given. A discussion of the parameters and some guidelines for their selection follow the card format description.

CONTROL CARDS FOR INDEXED FILES

Three types of control cards are used to provide additional information for files with INDEXED organization: the ISP INDEX card, the ISP DATA card, and the ISP RECORD card.

The ISP INDEX card has the following format:

```
1      8      16
______________________________________
ISP    INDEX  FC=fc[,PAGESZ=IPS]
```

where: fc is the file code of the index file.
IPS is the page size in words.

The ISP DATA card

```
1      8      16
ISP    DATA   FC=fc[,PAGESZ=DPS][,PAGEFIL=PF]
```

where: fc is the file code of the data file.
DPS is the page size in words.
PF is the percent of the page to be used.

The ISP RECORD card has the following format:

```
1      8      16
ISP    RECORD RECSZ=RS,KEYSZ=KS[,KEYOFF=KF]
```

where: RS is the size of the fixed length records in words.
KS is the size of the key in BCD character units.
KF is the offset of the key in BCD character units.

An ISP ETC descriptor card can be used to continue any of these cards.

Page Size

Many factors enter into the determination of page size. Since a link[1] is divided into pages, the page size should be chosen to divide evenly into the link size (3840 words). If the file is composed of fixed length records, the page size should provide for minimum unused space by being the closest number to a multiple of the record size (including the two record control words) plus the page control word.

However, since the page buffers are all the same size for INDEXED files, all INDEXED files should have the same page size for most efficient utilization of these buffers. Moreover, the page size can be altered from run to run to obtain better efficiency. Studies have shown that for random access small page sizes are most efficient and for sequential access larger pages are most efficient.

Percent Fill

The percent of the page to be used can also be altered from run to run. A percent less than 100 causes space to be reserved throughout the file. Subsequent adjustment of the percent fill allows records to be added in the previously unused space. Thus, if sufficient unused space is available on a page, records added to the file after its creation can be physically placed on the page to which they logically belong.

[1] For further background information, see Section IV of the File Management Supervisor manual.

Record and Key Parameters

The record size (RS) of the largest record in the file must be given. Since a maximum of four ASCII characters can be contained in a word, the number of words per record is calculated by dividing the total number of ASCII characters in the record by four.

$$RS = CEIL\left(\frac{\text{number-of-ASCII-characters}}{4}\right)$$

where: CEIL is the PL/I truncating function that returns the smallest integer greater than or equal to its argument.

The key size (KS) is determined by multiplying the number of ASCII characters in the key by a factor that expresses its length in BCD character units.

$$KS = \text{number-of-ASCII-characters-in-key} * \frac{9}{6}$$

The offset is determined by multiplying the number of ASCII characters before the key by the same factor. If the offset is not given, an offset of zero is assumed.

Memory Reservation

Space must be provided for file tables and page buffers by a $ USE control card, as follows:

```
1      8      16
$      USE    .XBUF1/n/,.XBUF2/2/
```

where: n is the number of words required for the INDEXED files of a program.

Each INDEXED file used requires a 160-word file table allocation. The page buffers are shared among all the INDEXED files of the program. An estimate of the number of words required can be obtained as follows:

n = 8 + 160 * NF + MAX(1016,(MPS+4)*NPB)

where: NF is the number of INDEXED files which are open.
MPS is the maximum page size for all the INDEXED files.
NPB is the number of page buffers needed. ISP requires that it be at least 3.
MAX is the PL/I built-in function.

PAGE BUFFERS

A page buffer is an area of memory used to hold a page during processing. The number of page buffers allocated for a file affects the efficiency of operation. For example, the efficiency of direct access of an INDEXED file can be improved if there are sufficient page buffers to allow the fine index and the coarse index to be retained in memory.

The Utilization Report produced as a result of the execution of a program using INDEXED files can be used to determine the change in efficiency accomplished by the change in the number of page buffers allocated. The ratio of logical to physical reads and writes provides a good indication of the improvement. The Utilization Report for INDEXED files is described later in this section.

Calculation of File Size

The size of the data file and the index file can be calculated by the following methods. The number of links required for the data file and index file can be specified on the device assignment card for each file. A sample calculation of file size is given in the example following this section.

The PL/I truncating functions FLOOR and CEIL are used in these formulas. These functions discard the fractional part of their arguments to produce an integer, as follows:

FLOOR(R) is the largest integer $\leq$ R

CEIL(R) is the smallest integer $\geq$ R

CALCULATION OF DATA FILE SIZE

The formulas given here for calculation of the data file size use the following variables whose values are furnished by the user.

NR = total number of records in the file
RS = record size (in words)
DPS = data page size (in words)
PF = percent fill

Values for the following variables are calculated using the given formulas.

N1 = number of records that can be stored on the first page
N2 = number of records that can be stored on each page after the first
NDP = total number of data pages required
NDL = number of 3840-word links required for the data file

$$\mathrm{N1} = \mathrm{FLOOR}\left[\frac{\mathrm{FLOOR}\ [\mathrm{DPS} * \mathrm{PF}/100] - 65}{\mathrm{RS} + 2}\right]$$

If NR + 1 $\leq$ N1, only one data page is required. Otherwise,

$$\mathrm{N2} = \mathrm{FLOOR}\left[\frac{\mathrm{FLOOR}\ [\mathrm{DPS} * \mathrm{PF}/100] - 1}{\mathrm{RS} + 2}\right]$$

$$\mathrm{NDP} = \mathrm{CEIL}\left[\frac{\mathrm{NR} - \mathrm{N1}}{\mathrm{N2}}\right] + 1$$

$$\mathrm{NDL} = \mathrm{CEIL}\left[\frac{\mathrm{NDP} * \mathrm{DPS}}{3840}\right]$$

CALCULATION OF INDEX FILE SIZE

The size of the index file is calculated using the following variables whose values are furnished by the user.

NDP = number of data pages (from previous calculation)
IPS = index page size (in words)
KS = key size (in characters)
KF = key offset (in characters)

The values to be calculated are as follows:

IEW = size of an index entry (in words)
I1 = number of index entries stored on the first index page
I2 = number of index entries stored on each page after the first
NFP = number of fine index pages required
NCP = number of coarse index pages required
NIP = total number of index pages required
NIL = number of 3840-word links required for the index file

$$\mathrm{IEW} = \mathrm{FLOOR}\left[\frac{\mathrm{KF} + \mathrm{KS} - 1}{4}\right] - \mathrm{FLOOR}\left[\frac{\mathrm{KF}}{4}\right] + 2$$

$$\mathrm{I1} = \mathrm{FLOOR}\left[\frac{\mathrm{IPS} - 67}{\mathrm{IEW}}\right]$$

If NDP $\leq$ I1, then only one index page is required. Otherwise,

$$\mathrm{I2} = \mathrm{FLOOR}\left[\frac{\mathrm{IPS} - 2}{\mathrm{IEW}}\right]$$

$$\mathrm{NFP} = \mathrm{CEIL}\left[\frac{\mathrm{NDP} - \mathrm{I1} - 1}{\mathrm{I2}}\right] + 1$$

Since NFP > 1, one or more coarse index pages are required.

$$\mathrm{NCP} = \mathrm{FLOOR}\left[\frac{\mathrm{NFP} + \mathrm{I2} - 1}{\mathrm{I2}}\right]$$

$$\mathrm{NIP} = \mathrm{NFP} + \mathrm{NCP}$$

$$\mathrm{NIL} = \mathrm{CEIL}\left[\frac{\mathrm{IPS} * \mathrm{NIP}}{3840}\right]$$

Example of INDEXED File Attachment

The following fragment illustrates the attachment of a file with INDEXED organization:

```
1      8      16
____________________________________________

$      SNUMB
$      IDENT
$      OPTION  PL1
$      PL1

IFA:   PROC;
       DCL 01 TABLE,
              02 CODE PIC "9999",
              02 NAME CHAR(20),
              02 CONT CHAR(50),
              02 LAST CHAR(6);
       ...
       OPEN FILE(Z1) SEQUENTIAL OUTPUT;
       ...
       WRITE FILE(Z1) FROM(TABLE) KEYFROM(NAME);
       ...
       END;

$      USE     .XBUF1/1132/,.XBUF2/2/
$      EXECUTE
       ...
$      FILE    C1,A1S,1R,MSU0400
$      FILE    C2,A2S,1R,MSU0310
       ...
$      DATA    Z1
ISP    INDEX   FC=C1,PAGESZ=320
ISP    DATA    FC=C2,PAGESZ=320,PAGEFIL=80
ISP    RECORD  RECSZ=20,KEYSZ=30,KEYOFF=6
       ...
$      ENDJOB
***EOF
```

DESCRIPTOR FILE CALCULATIONS

In this example, a page size of 320 words with 80% fill is specified. Therefore, only 256 words are used for record storage. The remaining 64 words on each page are reserved and can be used later for the addition of records to the file.

The parameters of the ISP RECORD card are determined by examining the record TABLE. The record size in words (RS) is determined by adding the number of ASCII characters in the record TABLE and dividing by four to get the number of words required, as follows:

$$RS = \text{CEIL}\left[\frac{\text{number-of-ASCII-characters}}{4}\right]$$

$$= \text{CEIL}\left[\frac{4 + 20 + 50 + 6}{4}\right] = \text{CEIL}\left[\frac{80}{4}\right] = 20$$

The key size in BCD character units (KS) is determined by taking the number of ASCII characters in the key and multiplying by a factor that expresses the length in BCD character units. Since a BCD character requires 6 bits and an ASCII character requires 9 bits, the calculation is:

$$\text{KS} = \text{no. of ASCII chars in key} * \left[\frac{\text{bits-per-ASCII-char}}{\text{bits-per-BCD-char}}\right]$$

$$\text{KS} = 20 * \left[\frac{9}{6}\right] = 30$$

The offset in BCD character units (KF) is determined by multiplying by the same factor, the number of ASCII characters by which the key is offset from the start of the record, as follows:

$$\text{KF} = \text{no.-of-ASCII-chars-before-key} * \left[\frac{\text{bits-per-ASCII-char}}{\text{bits-per-BCD-char}}\right]$$

$$\text{KF} = 4 * \left[\frac{9}{6}\right] = 6$$

MEMORY RESERVATION CALCULATION

The work region allocation is calculated, as follows:

$$n = 160 * 1 + (320 + 4) * 3 = 1132$$

The $ USE control card, in this example, requests 1132 words for the use of INDEXED files.

FILE SIZE CALCULATION

Assuming the data file consists of 25 records, the size of the data file is calculated from the page size and percent fill as follows:

$$\begin{aligned} \text{N1} &= \text{FLOOR} \left[\frac{\text{FLOOR}\ [320 * .80] - 65}{20 + 2}\right] \\ &= \text{FLOOR} \left[\frac{256 - 65}{22}\right] \\ &= 8 \end{aligned}$$

$$\begin{aligned} \text{N2} &= \text{FLOOR} \left[\frac{256 - 1}{22}\right] \\ &= 11 \end{aligned}$$

$$\begin{aligned} \text{NDP} &= \text{CEIL} \left[\frac{25 - 8}{11}\right] + 1 \\ &= 2 + 1 \\ &= 3 \end{aligned}$$

$$\begin{aligned} \text{NDL} &= \text{CEIL} \left[\frac{3 * 320}{3840}\right] \\ &= 1 \end{aligned}$$

One link is, therefore, specified for the data file on the device assignment card, as follows:

```
$       FILE    C2,A2S,1R,MSU0310
```

The size of the index file is calculated from the following set of formulas. The size of the index file page is taken to be the same as that of the data file page - 320 words.

The size of the index file is calculated as follows:

$$\begin{aligned} \text{IEW} &= \text{FLOOR}\left[\frac{4 + 20 - 1}{4}\right] - \text{FLOOR}\left[\frac{4}{4}\right] + 2 \\ &= 5 - 1 + 2 \\ &= 6 \end{aligned}$$

$$\begin{aligned} \text{I1} &= \text{FLOOR}\left[\frac{320 - 67}{6}\right] \\ &= 42 \end{aligned}$$

Since NDP < I1, only one index page is required. Therefore,

$$\begin{aligned} \text{NIL} &= \text{CEIL}\left[\frac{320 * 1}{3840}\right] \\ &= 1 \end{aligned}$$

One link is, therefore, specified for the index file on the device assignment card, as follows:

```
    $      FILE    C1,A1S,1R,MSU0310
```

UTILIZATION REPORT

When a program using INDEXED files is executed, a utilization report is prepared and upon completion of the job, the report is printed. This report provides a record of the program's file access and contains information that can be used to improve the efficiency with which the INDEXED files are accessed.

The utilization report for an INDEXED file has four columns containing information about the data file, information about the index file, file attributes, and job attributes. The following items are included in the utilization report.

LOGICAL READS
This counter is incremented by one each time a READ statement is executed.

LOGICAL WRITES
This counter is incremented by one each time a WRITE, REWRITE, or DELETE statement is executed.

PHYSICAL READS
This counter is incremented by one when a page is transferred from the external device to a page buffer.

PHYSICAL WRITES
This counter is incremented by one when a page is transferred from a page buffer to the external device.

PAGE SIZE
The number of words in a page.

PAGES ALLOCATED
The number of pages contained in the file.

PAGES USED INITIALLY
The number of pages actually used at initialization of the file.

OVERFLOW PAGES USED
The number of pages, used for the storage of records, beyond the last page used at file initialization.

TOTAL PAGES USED
The sum of PAGES USED INITIALLY and OVERFLOW PAGES USED.

COARSE PAGES
The number of index file pages required for the coarse index.

FINE PAGES
The number of index file pages required for the fine index.

MAXIMUM RECORD SIZE
The number of words in the largest record.

KEY SIZE
The number of characters in the key, expressed in BCD character units.

KEY OFFSET
The offset of the key from the beginning of the record in BCD character units.

COLLATING SEQUENCE
The collating sequence used for ordering keys.

FILE INITIALIZED
Date and time of initialization of the file.

FILE LAST UPDATED
Date and time of the last update of the file.

DELETED RECORDS
The number of deleted logical records currently in the file.

OVERFLOW RECORDS
The number of records written on overflow pages. This count includes active and deleted records.

TOTAL RECORDS
The number of records currently in the file. This count includes both active and deleted records.

BUFFER SIZE
The number of words in the buffer page. The size of the buffer page is determined by the largest page of all INDEXED files used.

NUMBER OF BUFFERS
The number of page buffers used by the program.

FILE ACCESS
The type of file activity.

EXAMPLES OF INDEXED FILE ACCESS

Figure 8-3 illustrates the creation of an INDEXED file. Data is taken from the system input file. The INDEXED file TABLE is opened for sequential output and records are written in the order of the key NAME. The utilization report produced from the execution of the job is given in Figure 8-4.

The format of this utilization report is compressed for inclusion in the manual, but the information is not changed. The utilization report shows that eight logical writes were performed, corresponding to the eight input items. Since the page size was specified to be 320 words and one link was requested for the file, 12 pages are allocated for the file. The specification of the buffer allocation of 1780 words on the $ USE control card results in the allocation of five buffers.

```
1      8       16
$      SNUMB
$      IDENT
$      OPTION  PL1
$      PL1     LIST

IFC:   PROC OPTIONS(MAIN);
       DCL TABLE RECORD FILE KEYED ENVIRONMENT(INDEXED);
       DCL SYSIN FILE;

       DCL 01 PAYROLL,
              02 PLANT      CHAR(12),
              02 NUMBER     CHAR(8),
              02 NAME       CHAR(24),
              02 GROSSPAY   PIC"ZZ,ZZZ,ZZZ",
              02 DEDUCTIONS PIC"ZZ,ZZZ,ZZZ";
       ON ENDFILE(SYSIN) GOTO EXIT;
       OPEN FILE(TABLE) OUTPUT SEQUENTIAL TITLE("AA");
LOOP:  GET LIST(PLANT,NUMBER,NAME,GROSSPAY,DEDUCTIONS);
       WRITE FILE(TABLE) FROM(PAYROLL) KEYFROM(NAME);
       GOTO LOOP;
EXIT:  CLOSE FILE(TABLE);
       END;

$      USE     .XBUF1/1780/,.XBUF2/2/
$      EXECUTE
$      LIMITS  10,50K,-2K
$      FILE    DX,A1S,1R,MSU0400,MSU0310
$      FILE    IX,A2S,1R,MSU0400
$      DATA    AA
ISP    INDEX   FC=IX,PAGESZ=320
ISP    DATA    FC=DX,PAGESZ=320
ISP    RECORD  RECSZ=16,KEYSZ=36,KEYOFF=30
$      DATA    I*
CLEVELAND 25067 JONAS 36367 7500
CLEVELAND 25068 JONSON 25163 5635
CLEVELAND 25069 JUDD 14453 2336
WASHINGTON 34567 KLAUS 1 0
ALBANY 122269 MONTVALE 12263 2215
CLEVELAND 25070 MOST 24567 5432
CLEVELAND 3524 SMTH 44778 12343
PHILADELPHIA 222233 TAYLOR 55569 23454
$      ENDJOB
***EOF
```

Figure 8-3. INDEXED File Creation

```
              INDEXED Sequential Processor Utilization Report


        Data File DX                                   Index File IX


Logical Reads                    0          Physical Reads                  0
Logical Writes                   8          Physical Writes                 1
Physical Reads                   0          Page Size (Words)             320
Physical Writes                  2          Pages Allocated                12
Page Size (Words)              320          Coarse Pages                    0
Pages Allocated                 12          Fine Pages                      1
Pages Used Initially             1          Total Pages Used                1
Overflow Pages Used              0
Total Pages Used                 1



        File Attributes                              Job Attributes


Maximum Record Size (Words)     16          Buffer Size (Words)           320
Key Size (Characters)           36          Number of Buffers               5
Key Offset (Characters)         30          File Access                 Build
Collating Sequence            6000
File Initialized  01/16/75 14.54
File Last Updated 01/16/75 14.54
Deleted Records                  0
Overflow Records                 0
Total Records                    8
```

Figure 8-4. Utilization Report for INDEXED File Creation

Figure 8-5 illustrates the access of the INDEXED file just created. Corrections to the spelling of two keys are taken from the system input file. The records with the correctly spelled key are written in the file and the records with the incorrectly spelled key are deleted from the file. The utilization report produced from the execution of this job is given in Figure 8-6.

The utilization report shows two logical reads corresponding to the two records entered under the incorrectly spelled key, and four logical writes, corresponding to the deletion of these two records and the addition of the two records under the correctly spelled key. Since all the records are on the same page, only one physical write and one physical read are necessary.

```
1      8       16
_____________________________________________________________

$      SNUMB
$      IDENT
$      OPTION  PL1
$      PL1     LIST

IFA:   PROC OPTIONS(MAIN);
       DCL TABLE RECORD FILE KEYED ENVIRONMENT(INDEXED);
       DCL SYSIN FILE;
       DCL (WRONGNAME,RIGHTNAME) CHAR(24) ALIGNED;
       DCL 01 PAYROLL,
              02 PLANT       CHAR(12),
              02 NUMBER      CHAR(8),
              02 NAME        CHAR(24),
              02 GROSSPAY    PIC"ZZ,ZZZ,ZZZ",
              02 DEDUCTIONS  PIC"ZZ,ZZZ,ZZZ";

       ON ENDFILE(SYSIN) GOTO EXIT;
       OPEN FILE(TABLE) UPDATE DIRECT TITLE("AA");
LOOP:  GET LIST(WRONGNAME,RIGHTNAME);
       READ FILE(TABLE) INTO(PAYROLL) KEY(WRONGNAME);
       NAME = RIGHTNAME;
       WRITE FILE(TABLE) FROM(PAYROLL) KEYFROM(RIGHTNAME);
       DELETE FILE(TABLE) KEY(WRONGNAME);
       GOTO LOOP;
EXIT:  CLOSE FILE(TABLE);
       END;
$      USE     .XBUF1/2000/,.XBUF2/2/
$      EXECUTE
$      LIMITS  10,50K,-2K
$      FILE    DX,A1S,1R,MSU0310,MSU0400
$      FILE    IX,A2S,1R,DSS270
$      DATA    AA
ISP    INDEX   FC=IX,PAGESZ=320
ISP    DATA    FC=DX,PAGESZ=320
ISP    RECORD  RECSZ=16,KEYSZ=36,KEYOFF=30
$      DATA    I*
JONAS JONES
SMTH SMITH
$      ENDJOB
***EOF
```

Figure 8-5. INDEXED File Access

```
                  INDEXED Sequential Processor Utilization Report


          Data File DX                               Index File IX


Logical Reads                     2        Physical Reads                  1
Logical Writes                    4        Physical Writes                 0
Physical Reads                    1        Page Size (Words)             320
Physical Writes                   1        Pages Allocated                12
Page Size (Words)               320        Coarse Pages                    0
Pages Allocated                  12        Fine Pages                      1
Pages Used Initially              1        Total Pages Used                1
Overflow Pages Used               0
Total Pages Used                  1


          File Attributes                            Job Attributes


Maximum Record Size (Words)      16        Buffer Size (Words)           320
Key Size (Characters)            36        Number of Buffers               5
Key Offset (Characters)          30        File Access                Update
Collating Sequence             6000
File Initialized  01/16/75 14.54
File Last Updated 01/16/75 14.55
Deleted Records                   2
Overflow Records                  0
Total Records                    10
```

Figure 8-6. Utilization Report for INDEXED File Access

SECTION IX

REGIONAL ORGANIZATION

This section describes the structure of files with REGIONAL organization. The method of attachment for REGIONAL files and the utilization report produced as a result of accessing a REGIONAL file are given. Finally, examples of the creation and access of a REGIONAL file are included.

REGIONAL files are processed by the Regional Sequential Processor (RSP) in the PL/I system.

REGIONAL FILE ACCESS

A file with REGIONAL organization consists of a number of regions, corresponding to the fixed length logical records of the file. REGIONAL files can be assigned only to direct access devices.

A record in a REGIONAL file does not have an imbedded key; instead, the source key on the data transmission statement indicates the region. Since the regions of the file correspond one-to-one to the logical records of the file, the source key specifies the position of the record within the file. The source key is a character string consisting of a maximum of 32 characters representing a positive integer value.

File Creation

A REGIONAL file can be generated either in sequential or direct mode. When the REGIONAL file is generated sequentially, the source keys must be given in ascending order if the KEYFROM option is specified on the data transmission statement. When the values of the source keys skip some integers, the omitted regions are filled with dummy records. For example, if source keys 1, 2, 4, 6, ... are specified, regions 3, 5, ... are filled with dummy records. When the file is closed, any remaining records are filled with dummy records.

When the file is created directly, all regions are filled with dummy records upon opening the file. Then the records are inserted in the regions specified by the value of the source key on the data transmission statement.

A dummy record contains an identifying code in the first word. The dummy record code can be specified on a control card at execution time. If the dummy record code is not specified, the octal number '177000000000' is used.

File Access

Once a file is created, it can be accessed either in sequential or direct mode. Each record retrieved should be checked to determine whether it is a data record or a dummy record.

SEQUENTIAL ACCESS

A SEQUENTIAL file can be opened with either the INPUT or UPDATE attribute. The data transmission statement cannot contain the KEY option, but the KEYTO option can be used; thus the file can have the KEYED attribute.

Records are retrieved in the order of ascending region number. Actual and dummy records are retrieved sequentially.

In SEQUENTIAL UPDATE, the execution of a REWRITE statement results in the replacement of the record retrieved by the READ statement (with the SET option, if the file has the UNBUFFERED attribute). The execution of a DELETE statement causes the most recently retrieved record to be replaced by a dummy record.

DIRECT ACCESS

The data transmission statement for direct access must include the KEY option to specify the region to be accessed. A new record is written in the region corresponding to the key value by the execution of a WRITE statement. The execution of a DELETE statement causes a dummy record to be written in the specified region.

No record checks are made by the system to determine whether the record being written over is an actual or dummy record. It is the user's responsibility to maintain the integrity of the file by checking the record code systematically.

Data Transmission Statements for REGIONAL Files

Table 9-1 lists the data transmission statements that can be used with REGIONAL files.

Table 9-1. Data Transmission Statements for REGIONAL Files

SEQUENTIAL OUTPUT		
WRITE FILE(file-name) FROM(variable-name) [KEYFROM(expression)];		
LOCATE based-var FILE(file-name)[KEYFROM(expression] [SET(pointer-var)] ;		
SEQUENTIAL INPUT		
READ FILE(file-name)	{INTO(variable-name) SET(pointer-var)}	[{KEY(expression) KEYTO(char-strng-var)}] ;
READ FILE(file-name)	IGNORE(expression);	
SEQUENTIAL UPDATE		
READ FILE(file-name)	{INTO(variable-name) SET(pointer-var)}	[{KEY(expression) KEYTO(char-strng-var)}] ;
READ FILE(file-name)	IGNORE(expression);	
REWRITE FILE(file-name);		
DELETE FILE(file-name) ;		
DIRECT OUTPUT		
WRITE FILE(file-name)	FROM(variable-name)	KEYFROM(expression) ;
DIRECT INPUT		
READ FILE(file-name)	INTO(variable-name)	KEY(expression) ;
DIRECT UPDATE		
READ FILE(file-name)	INTO(variable-name)	KEY(expression) ;
REWRITE FILE(file-name)	FROM(variable-name)	KEY(expression) ;
WRITE FILE(file-name)	FROM(variable-name)	KEYFROM(expression) ;
DELETE FILE(file-name)	KEY(expression) ;	

STRUCTURE OF A REGIONAL FILE

A REGIONAL file consists of a number of 320-word buffers, each of which contains at least one whole record; i.e., the maximum allowable record size is 320 words. As many records as can be fully contained will be placed in a single buffer, and when the remaining buffer space is less than the record size, the next record will be placed in the next buffer.

Figure 9-1 illustrates the structure of a REGIONAL file. This file was created using the default octal pattern for a dummy record and specifying keys of 2, 3, and 4. The first region of the file and the regions after region 4 are all filled with dummy records. The record size for this example is 100 words. Therefore, three records can be contained in each buffer and twenty words at the end of each buffer are unused.

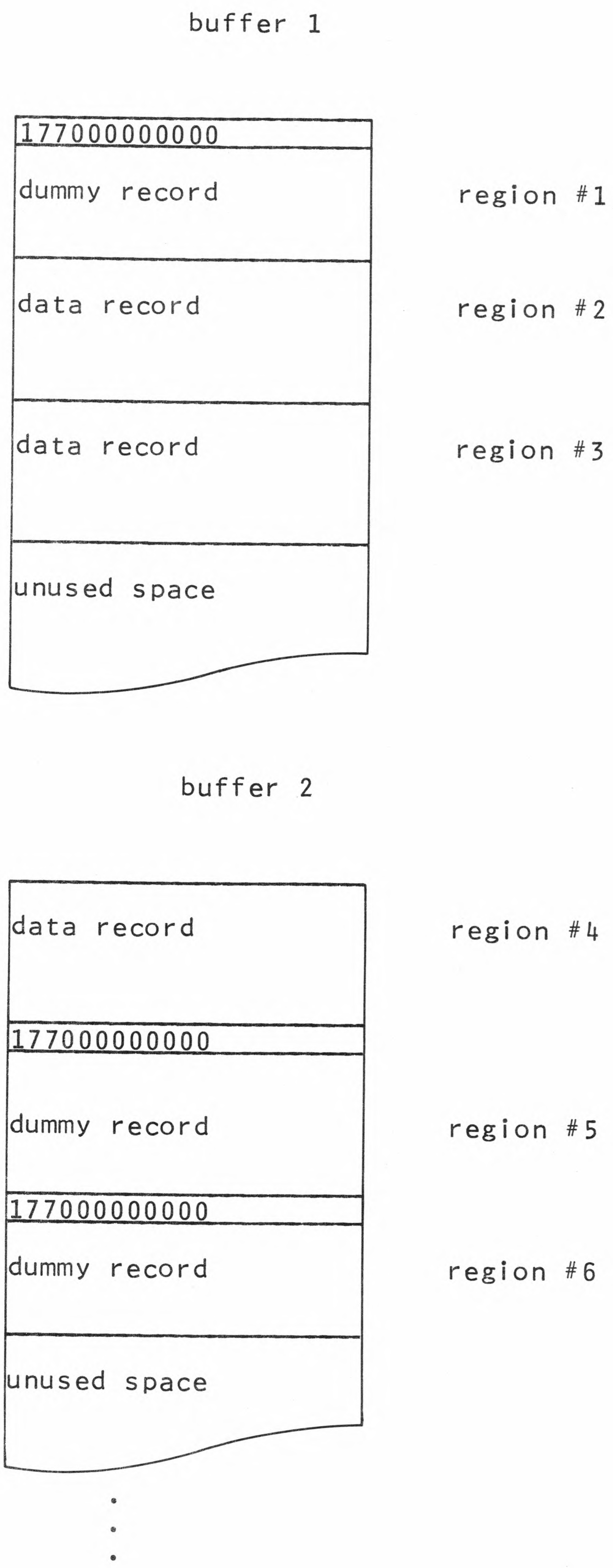


Figure 9-1. Structure of a REGIONAL File

ATTACHMENT OF A REGIONAL FILE

To specify and attach a REGIONAL file, the following requirements must be met:

- The file must be designated as REGIONAL. The REGIONAL keyword can be specified in the ENVIRONMENT attribute at compile time, or, if no ENVIRONMENT attribute is given, a descriptor file containing RSP cards can be supplied at execution time.
- A descriptor file must be provided to specify the file code and the size of the fixed length record.
- A device assignment control card for a direct access device must be provided for the file.
- The size of the work region to be reserved for REGIONAL files must be specified by $ USE control card.

Descriptor File for a REGIONAL File

A file with REGIONAL organization has two types of cards associated with it, namely: the RSP DATA card and the RSP RECORD card. The format of these two cards is as follows:

```
1      8       16
RSP    DATA    FC=fc

RSP    RECORD  RECSZ=n[,DBIT=d]
```

where: fc is the two character alphanumeric code identifying the file.

n is the number of words in the fixed length record.

d is the octal dummy record pattern. If d is given, it must be a 12 digit octal value.

If the dummy record pattern is not given, the pattern '177000000000' is assumed.

Memory Reservation

Space must be provided for file control blocks and buffers by an appropriate $ USE control card, as follows:

```
1      8       16
$      USE     .RBUF1/n/,.RBUF2/2/
```

where: n is the number of words required for the REGIONAL files of a program.

Each REGIONAL file requires a 400 word allocation. An estimate of the number of words needed can be obtained as follows:

n = 400 * F

where: F is the maximum number of REGIONAL files open at one time.

The work region allocation is shared among the REGIONAL files. When only one file is open, the remaining words in the allocation can be used as buffer space. The opening of a second file subtracts 400 words from the area that can be used for buffers, and so on. When a file is closed, its 400 word allocation is released.

Calculation of File Size

To determine the number of links required for a REGIONAL file, the number of records per 320-word buffer is calculated; then the number of buffers required for the file is calculated on the basis of the total number of records in the file; finally, the number of links required is determined by dividing the number of buffers required by the number of buffers per link, as follows:

$$\text{records-per-buffer} = \text{FLOOR}\left[\frac{320}{\text{record-size}}\right]$$

$$\text{buffers-per-file} = \text{CEIL}\left[\frac{\text{records-in-file}}{\text{records-per-buffer}}\right]$$

$$\text{links-required} = \text{CEIL}\left[\frac{\text{buffers-per-file}}{12}\right]$$

Example of REGIONAL File Attachment

The following fragment illustrates the attachment of a file with REGIONAL organization:

```
1      8         16
_______________________________________________

$      SNUMB
$      IDENT
$      USERID    XXXXXX$XXXXXX
$      OPTION    PL1
$      PL1

  EX3:   PROC
         ...
         OPEN FILE(F2) UPDATE TITLE('Y1') RECORD
         ...
$      USE       .RBUF1/2000/,.RBUF2/2/
$      EXECUTE
       ...
$      PRMFL     B1,...
       ...
$      DATA      Y1
RSP    FC=B1
RSP    RECSZ=40
       ...
$      ENDJOB
```

The $ USERID control card contains the system master catalog name and the log-on password. A $ USERID control card must be included in the deck if a $ PRMFL card is used. The USERID control card prevents unauthorized use of the system resources.

UTILIZATION REPORT

When a program using REGIONAL files is executed, a utilization report is prepared and upon completion of the job, the report is printed. The following items are included in the utilization report:

LOGICAL READS — This counter is incremented by one each time a READ statement is executed.

LOGICAL WRITES — This counter is incremented by one each time a WRITE, REWRITE, or DELETE statement is executed. This count includes the dummy records automatically written.

PHYSICAL READS — This counter is incremented by one when a page is transferred from the external device to a page buffer.

PHYSICAL WRITES — This counter is incremented by one when a page is transferred from a page buffer to an external device.

DUMMY WRITES — This counter is incremented by one whenever a dummy record is written. The number of dummy records automatically written is included in this count.

ACTUAL RECORDS MAX — The maximum region number actually used in a data transmission statement is given.

FILE LIMIT — The number of records in the file is given.

BUFFERS USED — This counter is incremented by one each time any page is transferred into a buffer for access to its records.

BUFFER SIZE MAX — The number of words in a buffer is given.

DUMMY RECORD OCT. — The pattern used as the first word of a record to indicate a dummy record is given in octal.

EXAMPLES OF REGIONAL FILE ACCESS

Figure 9-2 illustrates the creation of a REGIONAL file. The file is created in direct mode, so upon opening the file, the file is filled with dummy records. The utilization report produced from the execution of this job is given in Figure 9-3.

The utilization report shows that the file consists of 80 records. This number results from the record size (20 words) and the fact that the permanent data file SAMPLE/REGION has a maximum size of 5 blocks (1600 words) of mass storage space reserved for it as a result of a previous FILSYS activity (refer to the <u>File Management Supervisor</u>).

Opening the file causes 80 dummy records to be written, then 11 actual data records are written in the program. The number of <u>logical writes</u> is, therefore, 91.

The number of buffers used (12) includes 5 buffers used to write the 80 dummy records plus 7 buffers used as a result of the WRITE statement executions. The number of buffers used in the latter case is determined by the amount of buffer space in memory, the input sequence of keyed records, and the algorithm used by the REGIONAL processor for determining which buffer in memory (if there are more than one) will be overwritten when none contains the referenced record. In this example the amount of buffer space is 800 words, allowing for 2 buffers in memory. When neither contains the referenced record for a given WRITE statement execution, the algorithm turns out the least recently accessed buffer to bring in the needed page.

```
1      8       16
$      SNUMB
$      IDENT
$      USERID  XXXXXX$XXXXXX
$      OPTION  PL1
$      PL1     LIST

RFC:   PROC OPTIONS(MAIN);
       DCL POOL FILE RECORD KEYED ENVIRONMENT(REGIONAL);
       DCL SYSIN FILE;
       DCL 01 REC,
              02 ORDER  CHAR(32),
              02 IMAGE  CHAR(48);
       ON ENDFILE(SYSIN) GOTO EXIT;
       OPEN FILE(POOL) OUTPUT DIRECT TITLE("YY");
IN:    GET LIST(ORDER,IMAGE);
       WRITE FILE(POOL) FROM(REC) KEYFROM(ORDER);
       GOTO IN;
EXIT:  CLOSE FILE(POOL);
       END;

$      USE     .RBUF1/800/,.RBUF2/2/
$      EXECUTE
$      LIMITS  5,55K,-2K
$      PRMFL   RX,W,R,SAMPLE/REGION
$      DATA    YY
RSP    DATA    FC=RX
RSP    RECORD  RECSZ=20
$      DATA    I*
39     UPHAM
22     ESSEX
12     ROWE
45     BELLEVUE
6      STRATFORD
10     ORIENT
21     LEBANON
42     PORTER
34     ARDSMOOR
5      WASHINGTON
8      HILLSIDE
$      ENDJOB
***EOF
```

Figure 9-2. REGIONAL File Creation

```
                   RSP Utilization Report

                                          Data File RX
Logical Reads                                        0
Logical Writes                                      91
Physical Reads                                      12
Physical Writes                                     12
Dummy Writes                                        80
Actual Records Max                                  45
File Limit                                          80
Buffers Used                                        12
Buffer Size Max                                    320
Dummy Record Oct.                         177000000000
```

Figure 9-3. Utilization Report for REGIONAL File Creation

Figure 9-4 illustrates the access of the REGIONAL file just created. The utilization report produced as a result of the execution of this job is given in Figure 9-5.

The input data causes the records with keys 22 and 6 to be changed, the records with keys 34 and 10 to be deleted, and the record with key 7 to be added. The utilization report indicated that five logical writes were performed, one for each input item. Of these five logical writes, two writes were dummy writes since the deletion of a record in a REGIONAL file involves writing a dummy record. Three physical reads and physical writes were necessary since records 6, 7, and 10 are located in buffer #1, record #22 is located in buffer #2, and record #34 is located in buffer #3.

```
1      8       16
______________________________________________________________

$      SNUMB
$      IDENT
$      USERID  XXXXXX$XXXXXX
$      OPTION  PL1
$      PL1     LIST

RFA:   PROC OPTIONS(MAIN);
       DCL POOL FILE RECORD KEYED ENVIRONMENT(REGIONAL);
       DCL 01 REC,
              02 ORDER  CHAR(32),
              02 IMAGE  CHAR(48);
       ON ENDFILE(SYSIN) GOTO EXIT;

       OPEN FILE(POOL) UPDATE DIRECT TITLE ("YY");
LOOP:  GET LIST(ORDER,IMAGE);
       IF IMAGE = '*' THEN DELETE FILE(POOL) KEY(ORDER);
          ELSE REWRITE FILE(POOL) FROM(REC) KEY(ORDER);
       GOTO LOOP;
EXIT:  CLOSE FILE(POOL);
       END;

$      USE     .RBUF1/400/,.RBUF2/2/
$      EXECUTE
$      LIMITS  5,55K,-2K
$      PRMFL   RX,W,R,SAMPLE/REGION
$      DATA    YY
RSP    DATA    FC=RX
RSP    RECORD  RECSZ=20
$      DATA    I*
22     GLOUCESTER
34     *
6      AVON
7      HOLLAND
10     *
$      ENDJOB
***EOF
```

Figure 9-4. REGIONAL File Access

```
                RSP Utilization Report

                                 Data File RX
Logical Reads                               0
Logical Writes                              5
Physical Reads                              3
Physical Writes                             3
Dummy Writes                                2
Actual Records Max                         34
File Limit                                 80
Buffers Used                                3
Buffer Size Max                           320
Dummy Record Oct.                177000000000
```

Figure 9-5. Utilization Report for REGIONAL File Access

SECTION X

LINKING PL/I AND OTHER LANGUAGES

This section describes the mechanism for linking PL/I programs and programs written in other languages. The format and contents of the argument list are described.

DATA

Data can be shared between programs written in PL/I and programs written in other languages, provided the format and mapping of a PL/I data type is equivalent to the format and mapping of a data type in the other language. PL/I has a large number of data types; usually, a subset of these data types is available in another language. The internal representation for each PL/I data type is given in the next section of this manual. The information there and below applies both to data content of RECORD I/O files and arguments passed via CALL to subprograms.

Equivalent Data Representations

The following pairs of data declarations describe equivalent storage representations in Series 60 PL/I and COBOL-68.

```
           COBOL-68                         PL/I


     01 A PIC 9(8)  COMP-1.           DCL A FIXED BIN(35);
     01 B PIC 9(18) COMP-1.           DCL B FIXED BIN(71);
     01 C PIC 9(8)  COMP-2.           DCL C FLOAT BIN(27);
     01 D PIC 9(18) COMP-2.           DCL D FLOAT BIN(63);
     01 E PIC 9(10) COMP-3.           DCL E FIXED BIN(35);


     01 A OCCURS 2 TIMES.             DCL 01 A(2) ALIGNED,
     02 B OCCURS 3 TIMES.                 02 B(3),
     03 C OCCURS 4 TIMES.                 03 C(4),
     04 X PIC 9(8) COMP-1.                04 X FIXED BIN(35);
```

The following pairs of data declarations describe equivalent storage representation in Series 60 PL/I and FORTRAN.

```
          FORTRAN                         PL/I

     INTEGER A                       DCL A FIXED BIN(35);
     REAL B                          DCL B FLOAT BIN(27);
     DOUBLE PRECISION C              DCL C FLOAT BIN(63);
     COMPLEX D                       DCL D COMPLEX FLOAT BIN(27);
     CHAR*n E                        DCL E CHAR(n) ALIGNED;
```

Note that n must be less than or equal to 511 in FORTRAN and must be less than or equal to 256 in PL/I if the variable is involved in an Input/Output statement or requires conversion. Also, since PL/I character data is ASCII strings in 9-bit bytes, the FORTRAN program called by PL/I must be compiled in the ASCII mode.

The following pairs of data declarations describe equivalent storage representations in Series 60 PL/I and COBOL-74.

```
          COBOL-74                        PL/I

     01 A COMP-6.                    DCL A FIXED BIN(35);
     01 B PIC S999V99                DCL B CHAR(6);
          SIGN LEADING SEPARATE.
     01 C PIC X(10).                 DCL C CHAR(10);
     01 D OCCURS 2 TIMES.            DCL 01 D(2) ALIGNED,
     02 E OCCURS 3 TIMES.                02 E(3),
     03 F OCCURS 4 TIMES.                03 F(4),
     04 G COMP-6.                        04 G FIXED BIN(35);
```

Note that a future version of PL/I is expected to have a DECIMAL arithmetic format compatible with COBOL-74 COMPUTATIONAL data but incompatible with the present PL/I DECIMAL representation. The future form will use the packed decimal hardware format.

INTERFACE

When a PL/I program calls a program written in another language, the called program is responsible for saving any index registers used by PL/I programs and restoring these index registers when control is returned to the PL/I calling program. The argument list and return address are transmitted to the called program by index registers. The following index registers are involved:

Index register 6 contains the starting address of the argument list.

Index register 1 contains the return address to be used for normal return to the PL/I program.

Index register 2 contains the current stack frame header address and must be saved by the called program and restored before return to the PL/I calling program.

Argument List

The argument list contains a control word, followed by an entry for each argument. If any of the arguments has a variable length dimension (or, if the OPTIONS (VARIABLE) attribute is used in the procedure declaration), an additional entry is made for every argument in the list. The argument list has the following format:

0	18	24 35
m	n	
arg-1	off-1	0
arg-2	off-2	0

.
.
.

arg-n	off-n	0
desc-1	0	
desc-2	0	

.
.
.

desc-n	0

where: n indicates the number of arguments in binary fixed point.

m indicates the number of arguments in binary fixed point if descriptors are required.

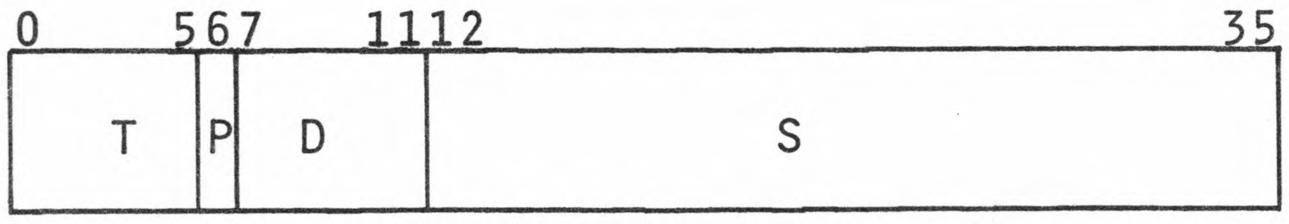
arg-i is a pointer value indicating the address of the i-th argument. If the argument has a bit offset, the offset value occupies the least significant bits of bits 18 through 23.

desc-i is a pointer value indicating the address of the i-th descriptor. The bit offset of this pointer value is always zero since argument descriptors begin on a word boundary.

If none of the arguments requires a descriptor, m is zero. If the called procedure does not have any arguments, index register 6 contains the address of a word containing zero.

Argument Descriptor

An argument descriptor contains information about the transferred arguments in the following format:



where: T indicates the data type of the arguments.

P indicates the packing status of the argument, as follows:

P=1 indicates the argument is packed.
P=0 indicates the argument is unpacked.

D gives the number of dimensions in an array. The array bounds and multipliers follow the base descriptors.

S gives the size.

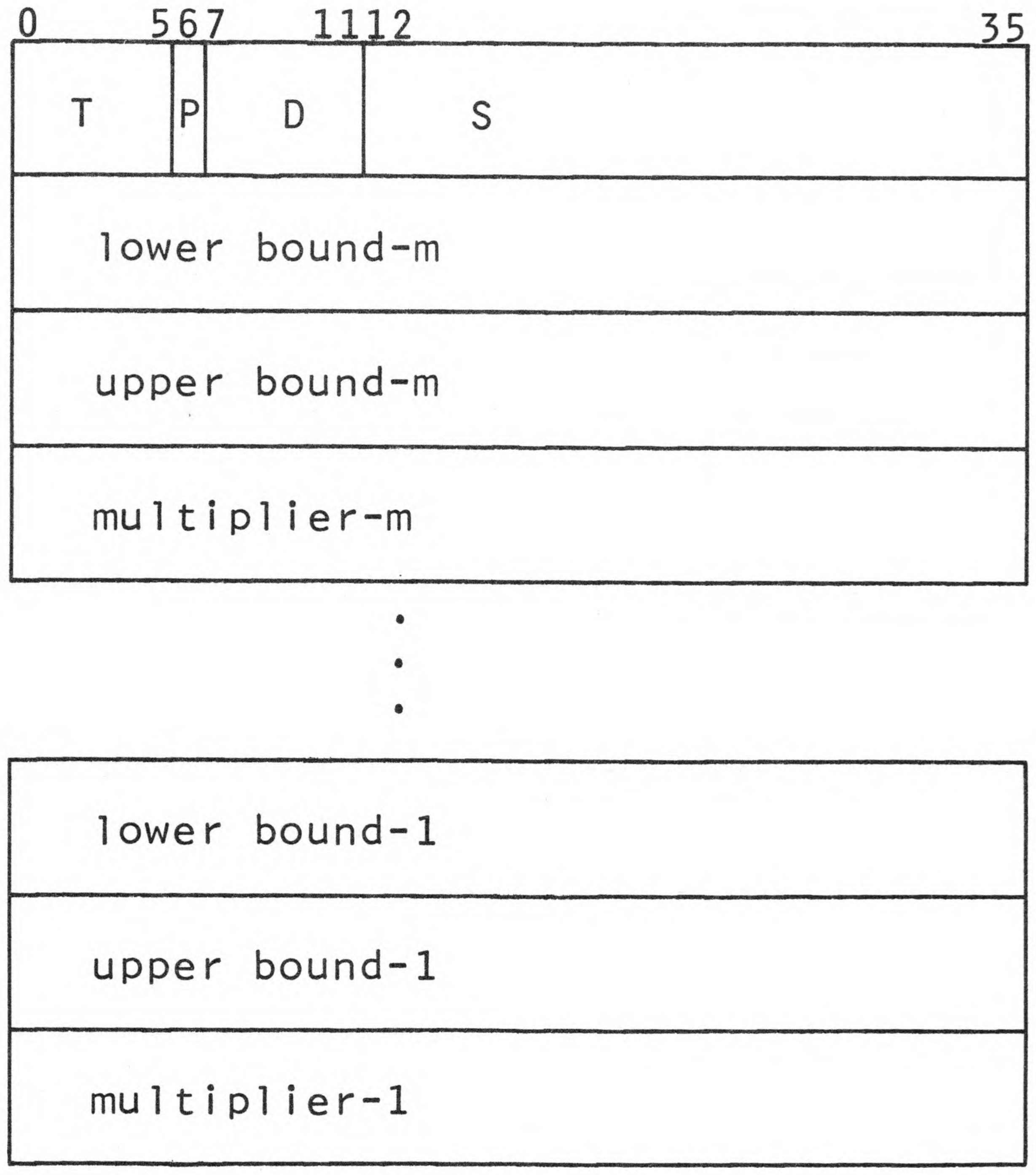
TYPE

The data type of the argument is indicated by a code. The code values and their interpretations are as follows:

Data Type Code	Data Type
1	Real binary fixed single precision
2	Real binary fixed double precision
3	Real binary float single precision
4	Real binary float double precision
5	Complex binary fixed single precision
6	Complex binary fixed double precision
7	Complex binary float single precision
8	Complex binary float double precision
9	Real decimal fixed
10	Real decimal float
11	Complex decimal fixed
12	Complex decimal float
13	Pointer
14	Offset
15	Label
16	Entry
17	Structure
18	Area
19	Bit string
20	Varying bit string
21	Character string
22	Varying character string
23	File

DIMENSIONS IN AN ARRAY

The array bounds and multipliers follow the descriptor for the array argument, as follows:



information for the m-th (rightmost) dimension

information for the first (leftmost) dimension

When the array elements are packed, the multiplier is in bits; otherwise, it is in words.

SIZE

The size field in the descriptor gives the following information depending upon the argument type:

string - the number of bits or characters.

area - the number of words.

structure - the number of elements in the structure.

arithmetic - the scale in the leftmost 12 bits and the precision in the rightmost 12 bits. The scale is a two's complement signed value.

EXAMPLE

The following PL/I program calls two external procedures, as follows:

```
P1:  PROC;
       DCL X  FIXED;
       DCL D  FIXED DECIMAL;
       DCL B1 BIT(1),
           B2 BIT(2);
       DCL 01 S,
              02 B(5) UNALIGNED,
                 03 C  FIXED,
                 03 D2 FIXED,
              02 A DECIMAL;
       DCL SUB1 ENTRY(FIXED,FIXED DECIMAL,BIT(1),FIXED UNALIGNED);
       DCL SUB2 ENTRY(FIXED,1,2(*) UNALIGNED,3 FIXED,3 FIXED,2 DECIMAL);
       CALL SUB1(X,D,B1,D2);
       CALL SUB2(X,S);
     END;
```

The object program produced for P1 includes procedure calls to SUB1 and SUB2. Before the transfer, index register 6 is set to point to the argument list. The procedures SUB1 and SUB2 must save index register 2 and restore it upon return. The argument lists for the two procedures are as follows:

Argument List for SUB1:

0	18	24 35
0		4
L(X)	0	
L(D)	0	
L(B1)	0	
L(D2)	18	

Argument List for SUB2:

0	18	24 35
2		2
L(X)		
L(S)		
L(DX)		
L(DS)		

and the argument descriptors are as follows:

	0 5	6	7 11	12 35	
DX 0	1				descriptor for X
DS 1	17			2	descriptor for S
2	17	1	1	2	descriptor for B
3				1	lower bound of B
4				5	upper bound of B
5				36	multiplier for B
6	1	1	1	17	descriptor for C
.				1	lower bound of C
.				5	upper bound of C
.				36	multiplier for C
.	1	1	1	17	descriptor for D2
.				1	lower bound of D2
.				5	upper bound of D2
.				36	multiplier for D2
.	9			7	descriptor for A

The descriptor for a structure is immediately followed by the descriptors for each of its members. Also notice that the members of dimensioned structures contain copies of the bounds of the containing structure.

OPTIONS ATTRIBUTE

The OPTIONS attribute can be used to generate standard calling sequences for programs written in GMAP, COBOL, or FORTRAN. For example, consider the following program, which calls external procedures written in PL/I, GMAP, COBOL, and FORTRAN:

```
P1:  PROC OPTIONS(MAIN);
       DCL X1 FIXED STATIC;
       DCL F1 FLOAT EXTERNAL;
       DCL D1 FIXED DECIMAL EXTERNAL;
       DCL CF1 COMPLEX FLOAT;
       DCL B1 BIT(1);
       DCL B2 BIT(2);
       DCL B3 BIT(3) ALIGNED STATIC;
       DCL C1 CHAR(1) EXTERNAL;
       DCL PSUB ENTRY(FIXED,FIXED DECIMAL,BIT(1),BIT(2));
       DCL GSUB ENTRY(FIXED,BIT(3) ALIGNED)
                OPTIONS(GMAP);
       DCL CSUB ENTRY(FIXED,FLOAT,FIXED DECIMAL,CHAR(1))
                OPTIONS(COBOL);
       DCL FSUB ENTRY(FIXED,FLOAT,COMPLEX FLOAT,CHAR(1))
                OPTIONS(FORTRAN);
       .
       .
       .
       CALL PSUB(X1,D1,B1,B2);
       .
       .
       .
       CALL GSUB(X1,B3);
       .
       .
       .
       CALL CSUB(X1,F1,D1,C1);
       .
       .
       .
       CALL FSUB(X1,F1,CF1,C1);
       .
       .
       .
     END;
```

Two different calling sequences can be generated for a procedure call to an entry declared with the OPTIONS attribute specifying GMAP, COBOL or FORTRAN. The simplest is possible only if the referenced arguments have the attribute STATIC and have no execution time location variability such as nonconstant subscripts. In such cases the code is:

```
TSX1      entryname
TRA       n+2,IC
ARG       0
ARG       arg-1
ARG       arg-2
 .
 .
 .
ARG       arg-n
```

When any argument fails to meet the above criteria, a normal PL/I calling sequence is generated. In this second case, an argument list is built as usual and a PL/I procedure call is made to a run-time support routine. The support routine then builds a calling sequence similar to that above, executes it, and, upon regaining control, returns through the normal PL/I mechanism.

The four code generation cases illustrated in the sample program above are:

```
CALL PSUB                   Normal PL/I procedure

     EAQ      .STATIC0      X1 in internal static storage
     STQ      11,SP
     EAQ      D1            External reference
     STQ      12,SP
     EAQ      8,SP          B1 in automatic storage
     STQ      13,SP
     EAQ      9,SP          B2 in automatic storage
     STQ      14,SP
     LDQ      4,DL          There are four arguments
     STQ      10,SP
     EAX6     10,SP
     TSXLP    PSUB          External reference

CALL GSUB                   Inline subroutine call

     TSXLP    GSUB          External reference
     TRA      4,IC
     ARG      0
     ARG      .STATIC0      X1
     ARG      .STATIC1      B3

CALL CSUB                   Inline subroutine call

     TSXLP    CSUB          External reference
     TRA      6,IC
     ARG      0
     ARG      .STATIC0      X1
     ARG      F1            External reference
     ARG      D1            External reference
     ARG      C1            External reference

CALL FSUB                   Dynamically built subroutine call

     EAQ      .STATIC0      X1
     STQ      11,SP
     EAQ      F1            External reference
     STQ      12,SP
     EAQ      6,SP          CF1 in automatic storage
     STQ      13,SP
     EAQ      C1            External reference
     STQ      14,SP
     LDQ      4,DL          Number of arguments
     STQ      10,SP
     EAXBP    FSUB          External reference
     EAX6     10,SP         Set arg-list pointer
     TSXLP    .P0369        GMAP-CALL
```

SECTION XI

INTERNAL REPRESENTATION OF PL/I DATA

To discuss the positioning of variables in storage, it is necessary to make a distinction between major variables and member variables. A major variable is either a level 01 structure or a variable not contained within a structure. A member variable is a variable contained within a structure. A major variable is positioned at a word or even-word boundary depending on its data type. A member variable is positioned at a bit, byte, word, or even-word boundary depending on its data type and alignment.

VARIABLES

Each PL/I variable has a data type, an aggregate type, and an alignment type. The data type and the aggregate type determine the values that can be accommodated by a storage unit. The alignment type affects the way in which the variable is laid out in storage.

Alignment

Every variable has an alignment attribute. An ALIGNED variable is stored for convenient access and an UNALIGNED variable is stored for conservation of storage.

If the alignment attribute is not declared for a variable, the variable acquires this attribute in the following way:

- If the variable is contained in a structure with an explicitly declared alignment attribute, the variable acquires the alignment attribute of the smallest containing structure with an explicit alignment declaration. For example, in the following structure:

```
DCL 01 S1,
       02  S2 ALIGNED,
           03 B1 BIT(2),
           03 S3 UNAL,
              04 B2 BIT(3),
           03 B3 BIT(4),
       02  B4 BIT(5);
```

  B1 acquires the alignment attribute of S2, namely: ALIGNED.
  B2 acquires the alignment attribute of S3, namely: UNALIGNED.
  B3 acquires the alignment attribute of S2, namely: ALIGNED.
  B4 remains unresolved.

- If the alignment of a variable cannot be resolved by the explicit declaration of containing structures, the alignment is determined by the variable's data type. A nonvarying string scalar or a structure acquires the UNALIGNED attribute. All other variables acquires the ALIGNED attribute. The following list indicates the default assumption made for an unresolved variable:

UNALIGNED	ALIGNED
nonvarying string variables	varying string variables
structures	arithmetic variables
	address variables
	area variables
	arrays

Representation

There are four units available for the representation of data, namely: bits, bytes, words, and double-words. The characteristics of the data type determine the minimum unit that can be used to represent it. The following list indicates the minimum units for some data types:

Data Type	Minimum Unit
binary arithmetic bit strings	bit
decimal arithmetic character strings	byte
varying strings file, entry, and label	word
complex arithmetic	double-word

The unit of representation determines the boundary requirement of the variable in memory. For example, a decimal number starts and ends on a byte boundary.

Positioning in Memory

A variable can be positioned in memory either to facilitate its access or to conserve storage. A frequently accessed variable should be positioned by the user at a word or even-word boundary and occupy an integral number of words. An infrequently-accessed variable should be positioned at its minimum unit boundary and occupy only the storage necessary for its representation.

The compiler assumes that a major variable is frequently accessed and therefore, positions it at a word boundary independent of its alignment attribute. Furthermore, a major variable that is an external or a double precision binary arithmetic variable is positioned at an even-word boundary.

The positioning of a member scalar variable depends upon its data type and its alignment. All ALIGNED scalar variables are positioned at a word or even-word boundary. UNALIGNED scalar variables are positioned at the boundary determined by their data type.

The positioning of a member aggregate variable depends upon the maximum unit of representation of its components and its alignment. ALIGNED aggregates are positioned at a word or even word boundary. UNALIGNED member aggregates are positioned at the maximum unit of representation of their components. For example, an UNALIGNED member aggregate consisting of UNALIGNED bit strings starts at a bit boundary and occupies only as many bits as necessary to represent its contents. However, an UNALIGNED member aggregate consisting of UNALIGNED bit and character strings starts at a byte boundary and occupies enough bytes to represent its contents.

A more detailed discussion of the positioning of member variables is given in the second half of this section.

Supplementary Storage

When a variable is positioned for efficient access, it sometimes occupies more storage than is necessary for its representation. This additional storage is called supplementary storage. The supplementary storage is used in conjunction with the minimum storage required for the variable to permit a larger and more convenient representation of the stored value. For example, the value is stored for whole word referencing and no shifting or masking is required.

Filler Storage

The positioning of variables can create unused space. When the unit of representation of two adjacent variables is different, filler storage is often required. For example, a variable represented in bits can be followed in memory by a variable represented in bytes. If the last bit occupied by the first variable is not the last bit of a byte, the bits between the last bit of the variable and the first bit of the next byte are filler storage. Filler storage is never allocated at the beginning of a variable.

Filler storage is also created by alignment requirements. For example, an ALIGNED complex number followed by an UNALIGNED bit string containing 1 bit followed by another ALIGNED complex number results in 71 bits of filler storage.

Grouping variables with the same unit of representation and alignment minimizes the amount of filler storage allocated. Filler storage within an array is especially costly since the unused space occurs within each element of the array.

Packed Property

The terms packed and unpacked are applied to variables to describe their internal representation. A scalar variable is said to be packed if it is positioned at a boundary determined by its minimum unit of representation and occupies only enough of those units to represent its value. A scalar variable is said to be unpacked if it is positioned at a boundary greater than its minimum unit of representation.

An arithmetic, nonvarying string, or pointer variable that is declared UNALIGNED is packed. An aggregate variable that is declared UNALIGNED and contains only packed variables is packed.

The symbol table listing gives the alignment attribute and packed property for every member variable.

STORAGE LAYOUT RULES FOR PL/I MEMBER VARIABLES

Exact rules for the layout of a member variable in the 36-bit, 4-byte words of memory are given here. The rules assume that the starting layout address of the variable is the terminating word and bit address of the immediately preceding structure member (the bit offset for the first member of a level structure is 0). The rules are given for scalar variables, then for structure variables, and finally for array variables.

Storage Layout for Member Scalars

To determine the storage layout for a given scalar variable at a given word and bit address, proceed as follows:

1. Begin the layout at the given address.

2. Use Table 11-1 to determine the required boundary for the variable. If the starting address is not at a boundary of the required type, then lay out filler storage up to the next boundary of the required type.

3. Use Table 11-1 to determine the minimum storage for the variable. Add the specified amount of storage to the layout.

4. If the layout does not end at a boundary of the required type (as determined in Step 2), then lay out supplementary storage up to the next boundary of the required type.

Table 11-1. Boundary and Length for Scalar Variables

Data Type	Required Boundary		Minimum Storage
	ALIGNED[1]	UNALIGNED[2]	
REAL FIXED BINARY(p,q)			
$1 \le p \le 35$	word	bit	(p+1) bits
$36 \le p \le 71$	even word	bit	(p+1) bits
REAL FIXED DECIMAL(p,q)	word	byte	(p+1) bytes
REAL FLOAT BINARY(p)			
$1 \le p \le 27$	word	bit	(p+9) bits
$28 \le p \le 63$	even word	bit	(p+9) bits
REAL FLOAT DECIMAL(p)	word	byte	(p+2) bytes
COMPLEX FIXED BINARY(p,q)	even word	bit	2*(p+1) bits
COMPLEX FIXED DECIMAL(p,q)	word	byte	2*(p+1) bytes
COMPLEX FLOAT BINARY(p)	even word	bit	2*(p+9) bits
COMPLEX FLOAT DECIMAL(p)	word	byte	2*(p+2) bytes
CHARACTER			
NONVARYING	word	byte	(m1) bytes
VARYING	word	word	(m1+4) bytes
BIT			
NONVARYING	word	bit	(m1) bits
VARYING	word	word	(m1+36) bits
PICTURE "P" (with related data type CHAR(n))	word	byte	(n) bytes
LABEL	word	word	1 word
ENTRY	word	word	1 word
FORMAT	word	word	1 word
POINTER	word	bit	36 bits
OFFSET	word	bit	36 bits
FILE	word	word	1 word
AREA(as)	word	word	(as) words

[1] Applies to both major and member scalar variables.

[2] Applies to member scalars only.

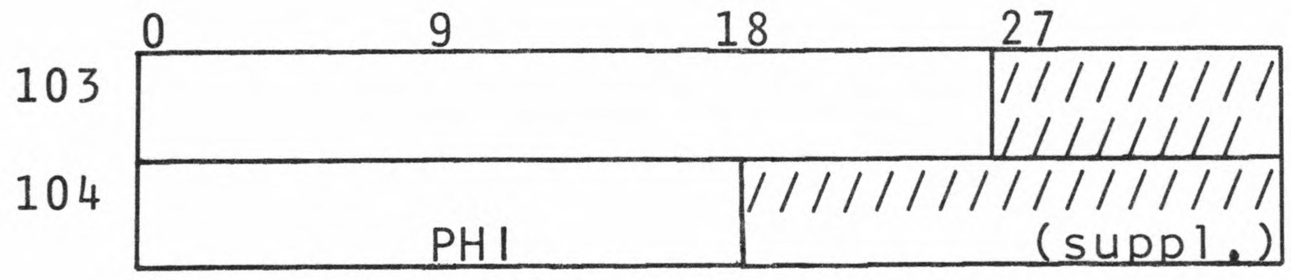
As the basis for an example of the layout of a scalar variable, consider the following declaration:

```
          DCL  01  A1,
                   .
                   .
                   .
                   02 PHI  FIXED,
                .
                .
                .
```

According to the default rules, this declaration is equivalent to:

```
          DCL  01  A1 UNALIGNED,
               .
               .
               .
                   02 PHI FIXED BIN(17) ALIGNED,
```

Suppose the starting address for PHI is bit 27 of word 103. Then the rules just given prescribe the following layout for PHI:



This layout is determined as follows:

1. The layout begins at bit 27 of word 103.

2. According to Table 11-1, the required boundary for the variable is <u>word</u>. Since the starting address is not at a word boundary, the layout begins with one byte of filler storage (fully shaded).

3. The layout continues with the minimum storage for the variable, 18 bits.

4. Since the required boundary is <u>word</u>, the layout concludes with 18 bits of supplementary storage (half shaded).

The storage available for PHI is a full word, the minimum plus the supplement. Therefore, the value of PHI can be stored in a way that is suitable for the full-word operations of the hardware. PHI is right-justified in the word to eliminate the need for masking and shifting operations.

Storage Layout for Member Structures

To determine the storage layout for a given member structure variable at a given word and bit address, proceed as follows:

1. Begin the layout at the given address.

2. Determine the required boundary type for the structure as follows:

   a. Make a list of the required boundaries for the members of the structure.

   b. If the structure itself is ALIGNED, then add the boundary word to the list.

   c. Find the boundary on the list that refers to the largest unit of storage and take that to be the required boundary for the structure.

   If the starting address is not a boundary of the required type, then lay out filler storage up to the next boundary of storage.

3. Continue the layout of the structure by laying out storage for each of its members.

4. If the required boundary is even word or word and the layout does not end at a word boundary, then lay out supplementary storage to the next word boundary.

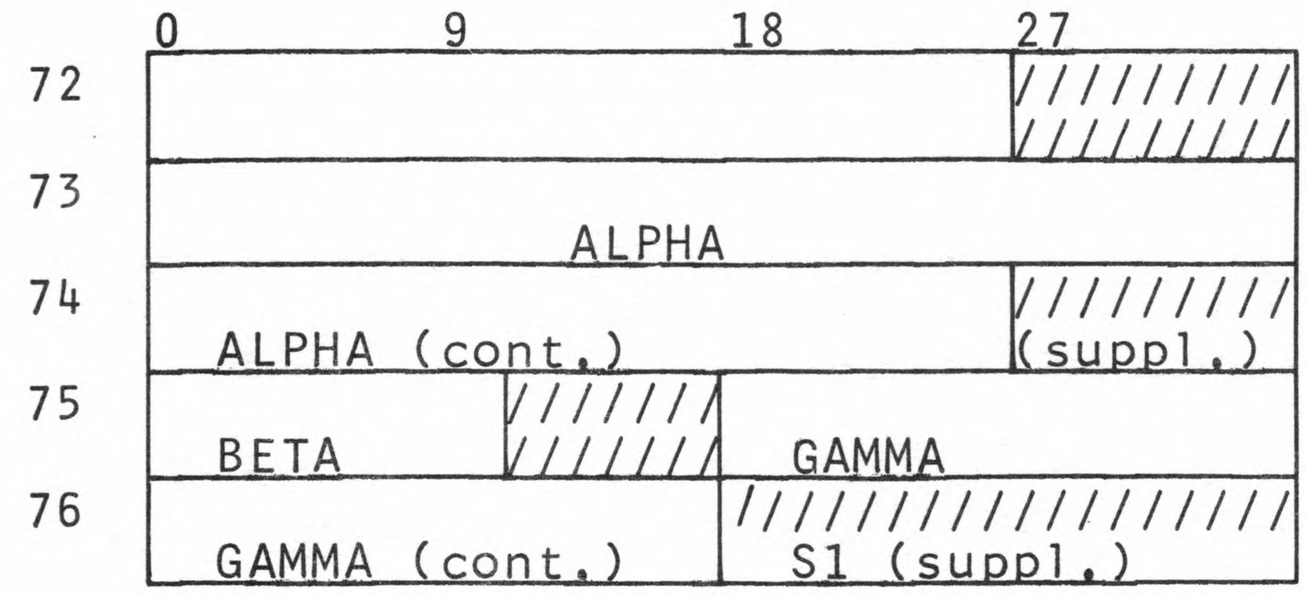
As the basis for an example of the layout of a structure variable, consider the following declaration:

```
DCL  01  S,
         .
         .
         .
         02  S1,
             03 ALPHA DEC(6,2),
             03 BETA  BIT(12),
             03 GAMMA CHAR(4),
         .
         .
         .
```

According to the default rules, this declaration is equivalent to:

```
DCL  01  S  UNAL,
         .
         .
         .
         02  S1  UNAL,
             03  ALPHA REAL FIXED DECIMAL(6,2) ALIGNED,
             03  BETA BIT (12) NONVARYING UNAL,
             03  GAMMA CHARACTER(4) NONVARYING UNAL,
         .
         .
         .
```

Suppose the starting address for the structure S1 is word 72, bit 27. Then the rules prescribe the following layout for S1:



This layout is determined as follows:

1. The layout of S1 begins at bit 27 of word 72.

2. The list of required boundaries for the members of S1 is:

   word
   bit
   byte

   The maximal boundary from this list is <u>word</u>. Since the layout begins on a bit boundary, 9 bits of filler storage are required. Hence, S1 begins at word 73, bit 0.

3. The layout continues with the 3 members of the structure. Each is laid out according to the rules for a scalar, as follows:

   ALPHA The required boundary is <u>word</u> and the minimum storage is 7 bytes. The layout ends with 1 byte of supplementary storage.

   BETA The required boundary is <u>bit</u> and the minimum storage is 12 bits. No filler or supplementary storage is used.

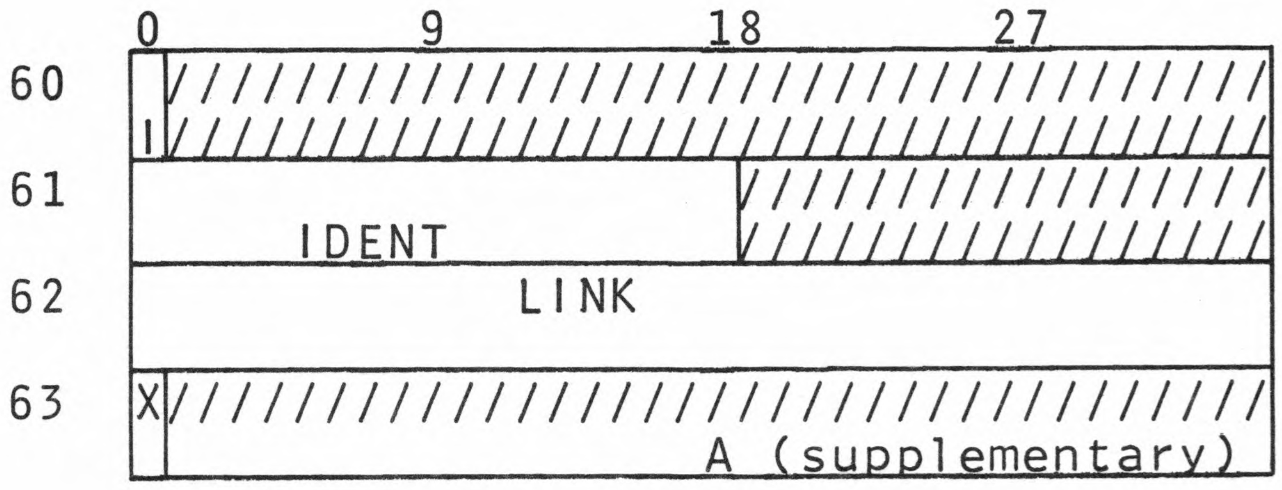
   GAMMA The required boundary is <u>byte</u> and the minimum storage is 4 bytes. The layout begins with 6 bits of filler storage. No supplementary storage is required at the end.

4. Since the layout of the last member ends in the middle of a word and the required boundary for the structure is <u>word</u>, the layout of the structure ends with 2 bytes of supplementary storage.

The order in which the members of a structure are arranged can have a significant effect on the amount of storage required for the layout of a structure. As an example, consider:

```
DCL  01 A,
        02 I BIT,
        02 CELL,
           03 IDENT CHAR(2),
           03 LINK PTR,
        02 X BIT;
```

The layout for A requires 4 full words, as follows:

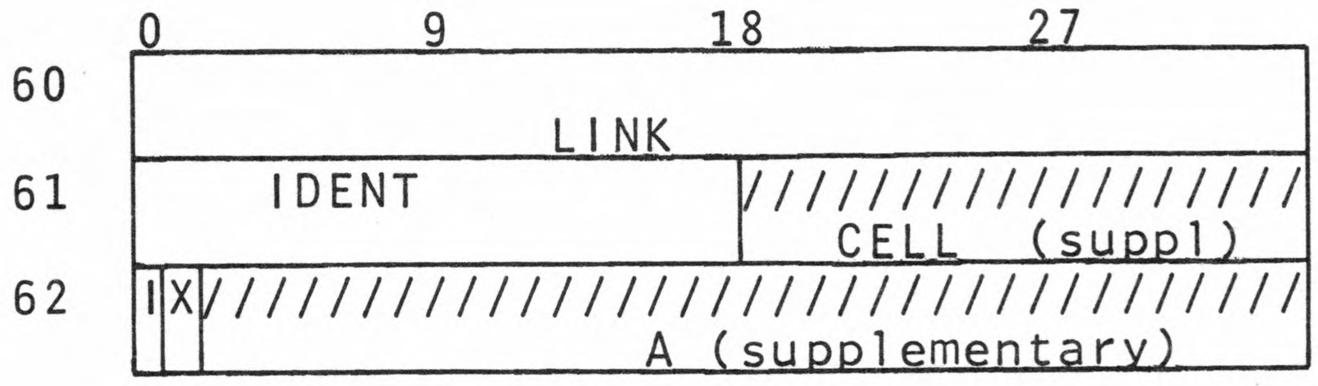


This layout arises from the fact that LINK is an ALIGNED POINTER and specifies a word boundary not only for its own storage, but also for the structure A.CELL of which it is a member.

Consider the following revision of the declaration of the structure A:

```
DCL 01 A,
       02 CELL,
          03 LINK PTR,
          03 IDENT CHAR(2),
       02 I BIT,
       02 X BIT;
```

In most cases, this change in the ordering of the members of A has no effect on the usage of the structure, but the resulting layout occupies 3 words instead of 4:

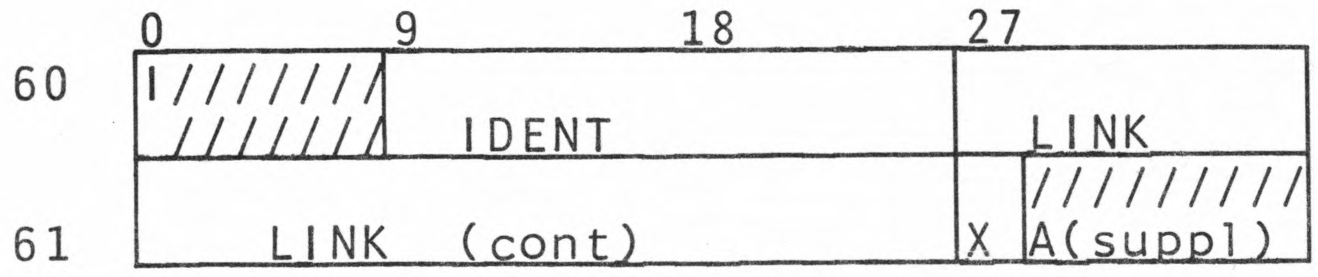


Some storage is still wasted in this layout. IDENT, I, and X could all fit in 1 word. However, to further improve the allocation in storage, a change in the level structure is required; that type of change could well affect the usage of the structure.

Consider a different revision of the declaration of the structure A:

```
DCL 01 A UNAL,
       02 I BIT,
       02 CELL,
          03 IDENT CHAR(2),
          03 LINK PTR,
       02 X BIT;
```

Because of the addition of the attribute UNAL for A, the layout uses 2 words instead of 4:



For this version, however, the interpretation of the value of the POINTER value takes more time than for the ALIGNED value.

Storage Layout for Member Arrays

To determine the storage layout for a given array variable at a given address, proceed as follows:

1. Begin the layout at the given address.
2. The required boundary for the array is the same as for the elements of the array. If the starting address is not a boundary of the required type, then lay out filler storage up to the next boundary of the required type.
3. Continue the layout of the array by laying out storage for each of its elements.
4. If an element does not end at the required boundary, then lay out supplementary storage to the next boundary of the required type.

The alignment attribute of an array is especially important, since it is in the layout of large arrays that the alignment can have a significant effect on storage requirements. As a simple illustration, consider the following declarations:

```
DCL PM1(50,50) BIT;

DCL PM2(50,50) BIT ALIGNED;
```

The array PM1 requires 70 words, whereas PM2 requires 2500 words.

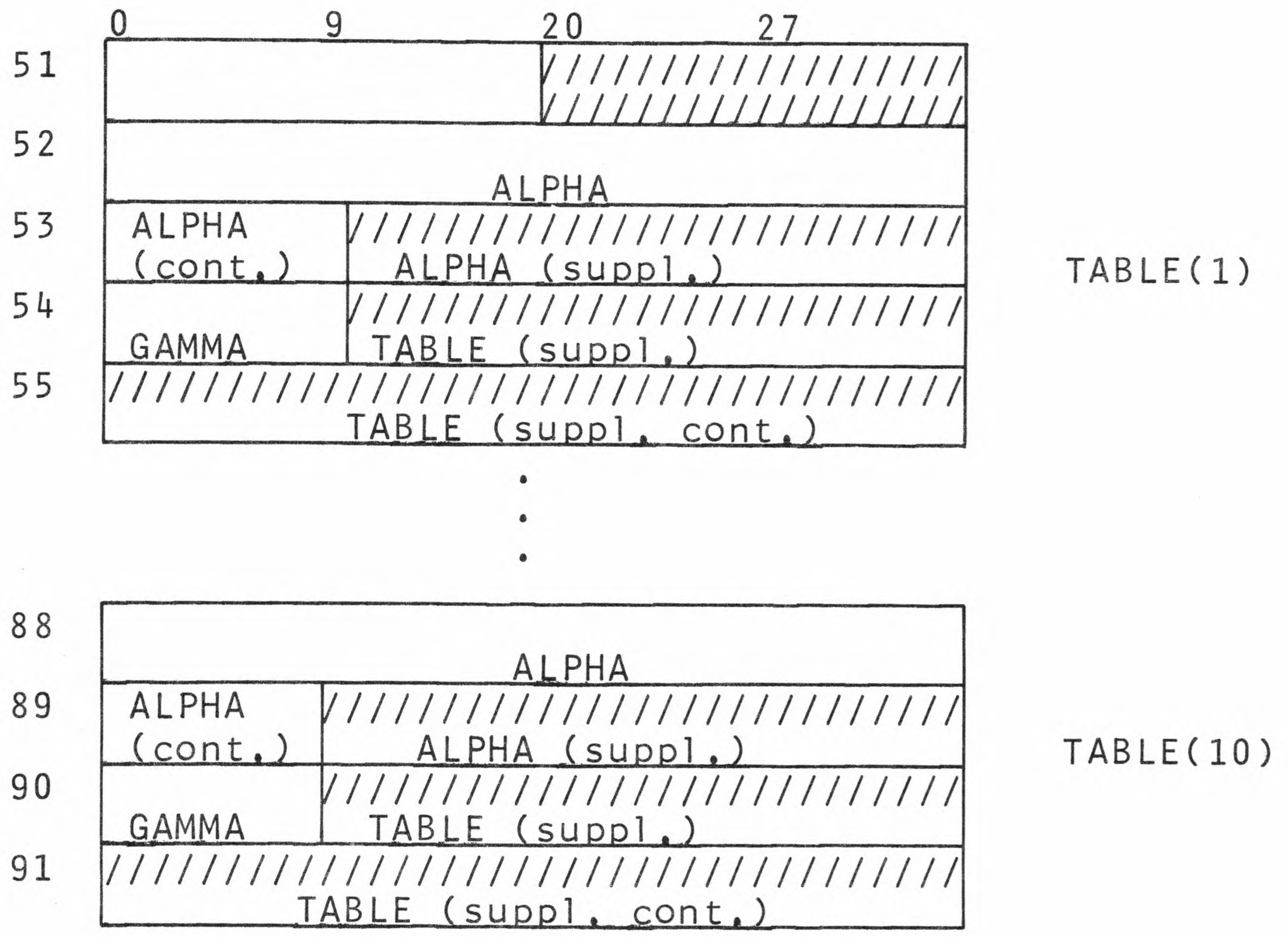
For a second example of the layout of an array, consider the following declaration:

```
DCL  01  S,
         .
         .
         .
         02 TABLE(10),
            03 ALPHA FIXED BIN(44),
            03 GAMMA CHAR(1),
         .
         .
         .
```

According to the default rules, this declaration is equivalent to:

```
DCL 01 S,
         .
         .
         .
         02 TABLE(10) UNAL,
            03 ALPHA  FIXED BIN(44) ALIGNED,
            03 GAMMA  CHAR(1) UNAL,
         .
         .
         .
```

Suppose the starting address for the layout of TABLE is word 51, bit 20. Then the rules prescribe the following layout for TABLE:



The layout is determined, as follows:

1. The layout of TABLE begins at bit 20 of word 51.

2. The list of required boundaries from Table 11-1 for members of an element of TABLE is:

even word	(ALPHA)
byte	(GAMMA)

   The maximal boundary is <u>even word</u>, which becomes the required boundary for a TABLE element (and for TABLE itself). Since the layout starts on a bit boundary, 16 bits of filler storage are required. Thus, the first element of TABLE actually begins at word 52, bit 0.

3. The layout for each element of TABLE is determined as follows (from the rules for laying out member scalars):

   ALPHA The required boundary is even word and the minimum storage is 45 bits. Twenty-seven bits of supplementary storage are required to bring the layout up to the next even word boundary. Note that in the resulting double-word the value of ALPHA will be right-justified because it is binary.

   GAMMA The required boundary is byte and the minimum storage is one byte. No filler or supplementary storage is required.

4. Since the last member (GAMMA) of the element does not end on the required even-word boundary for the element, lay out supplementary storage of seven bytes.

Note that the storage requirements for TABLE can be considerably reduced by altering its declaration:

```
DCL 01 S,
       .
       .
       .
       02 TABLE(10) UNAL,
          03 ALPHA FIXED BIN(44),
          03 GAMMA CHAR(1),
       .
       .
       .
```

In this case, TABLE will actually begin at bit 27 of word 51, each element will occupy only 6 bytes of storage, and TABLE will end on bit 26 of word 66.

SECTION XII

INCLUDE FILES

This section describes the creation and maintenance of the INCLUDE file. The INCLUDE file contains the macro bodies that can be referenced in PL/I programs by the use of the %INCLUDE statement. The use of the utility program SRCLIB is described and illustrated.

SRCLIB PROGRAM

The system utility program, SRCLIB, is used to create and maintain an INCLUDE file. The actions to be performed are specified by a series of control cards. The SRCLIB control cards provide for the initialization and creation of the INCLUDE file, subsequent modification by the inclusion and deletion of macro text, and copying and listing activities.

USE OF THE SRCLIB PROGRAM

The SRCLIB program is called, using a $ PROGRAM control card, from the same library file as the compiler.

```
$       PROGRAM SRCLIB
```

Files Used by the SRCLIB Program

Several files are used by the SRCLIB program. For each file code, a description is given in the following list:

File Code — Description

.L — INCLUDE file, which contains the macro text to be created or maintained.

IN — input file, which contains the control cards and text used by the SRCLIB program.

any — input or output file, which can be used either to contribute text to the INCLUDE file or save text from the INCLUDE file.

WK — work file, which is required for the activity when the MODIFY control card is present.

The SRCLIB program gets its directions from the input file, IN. The control cards in that file determine the actions to be taken on the INCLUDE file, .L. Some of these control cards specify the file code of another file for input to or output from the INCLUDE file. The presence of the MODIFY control card makes it necessary to include a work file, WK.

SRCLIB CONTROL CARDS

SRCLIB control cards give information to the SRCLIB program about the creation and maintenance of the INCLUDE file. SRCLIB control cards are summarized in Table 12-1. Following the table, each control card is described in detail.

Table 12-1. SRCLIB Control Cards

Card Name	Meaning	Parameters
ALTER	Add text to or delete text from the INCLUDE file.	line-n[,line-m]
COPY	Copy specified macro to the specified file.	text-name,file-code[,BCD]
CREATE	Place new text in the INCLUDE file.	text-name [,file-code]
DELETE	Delete the specified macro from the INCLUDE file.	text-name
INITIAL	Initialize the INCLUDE file.	
LIST	List the specified macro or the entire INCLUDE file.	[text-name]
MODIFY	Modify the specified macro in the INCLUDE file.	text-name
SAVE	Copy the entire INCLUDE file to the specified file.	file-code[,BCD]

The text-name is the name of a macro in the INCLUDE file. In all cases, the text-name is limited to a maximum of 32 characters.

The file-code is the two character name used to identify a file.

ALTER Control Card

The * ALTER control card is used to modify the INCLUDE file. A * ALTER control card gives the line numbers that are to be modified within the macro text. The * MODIFY control card gives the name of the macro to be modified and therefore, must always precede a series of ALTER cards. The format of the * ALTER control card is as follows:

```
1      8       16
__________________________________________________
*      ALTER   line-n[,line-m]

where:  line-n  is a decimal integer
        line-m  is a decimal integer
```

If only line-n is specified, the text following the * ALTER card is inserted in the macro named by the MODIFY control card before the line specified.

If both line-n and line-m are specified, line-n through line-m of the macro specified by the * MODIFY card are deleted. If text follows the * ALTER card, it is inserted at the point of deletion.

For example, consider the following sequence of control cards:

```
*       MODIFY  MAC1
*       ALTER   10
        DCL X1 FIXED;
        DCL X2 FLOAT;
*       ALTER   20,24
*       ALTER   31,36
        CALL P1;
```

The declarations of X1 and X2 are inserted in the macro MAC1 before line 10; lines 20 through 24 of MAC1 are deleted; and lines 31 through 36 of MAC1 are replaced by the procedure call to P1.

COPY Control Card

The * COPY control card is used to copy a macro from the INCLUDE file to the file specified by the file code. The format of the * COPY control card is as follows:

```
1      8       16
__________________________________________________________
*      COPY    text-name,file-code[,BCD]

where:  text-name  identifies the macro to be copied.
        file-code  identifies the file to which to copy the text.
        BCD        indicates that the text is to be represented in BCD
                   rather than ASCII on the designated file.
```

The * COPY control card allows the user to select and copy a single macro from the INCLUDE file. To copy the entire INCLUDE file, the * SAVE control card is used.

CREATE Control Card

The * CREATE control card is used to create or extend the INCLUDE file. The format of the * CREATE control card is as follows:

```
1      8      16
*      CREATE text-name[,file-code]
```

where: text-name identifies the name to be associated with the macro text.

file-code identifies the file containing the macro text.

If the file-code is not specified, the card images following the control card in the IN file are used as the macro text. If the file-code is specified, the macro text is taken from the file identified by that file code.

If the text-name specified for the new macro already exists in the INCLUDE file or if there is not sufficient space in the INCLUDE file to enter the new text, the * CREATE control card is ignored and a warning message printed.

DELETE Control Card

The * DELETE control card is used to delete a macro from the INCLUDE file. The format of the * DELETE card is as follows:

```
1      8      16
*      DELETE text-name
```

where: text-name indicates the name of the macro to be deleted from the INCLUDE file.

If the specified macro is not found in the INCLUDE file, the * DELETE card is ignored and a warning message printed.

INITIAL Control Card

The * INITIAL control card is used to initialize the INCLUDE file. The format of the * INITIAL card is as follows:

```
1      8      16
*      INITIAL
```

An initialized INCLUDE file contains no macro names or text.

LIST Control Card

The * LIST control card is used to output the text associated with a macro name. The format of the * LIST card is as follows:

```
1      8       16
*      LIST    [text-name]
```

where: text-name indicates the macro whose text is to be output.

If the text-name is omitted from the * LIST card, the text for all names registered in the INCLUDE file is listed.

MODIFY Control Card

The * MODIFY control card is used to indicate the macro that is to be modified by the * ALTER cards that follow. The format of the * MODIFY control card is as follows:

```
1      8       16
*      MODIFY  text-name
```

where: text-name indicates the macro to be modified by the * ALTER cards.

If the text-name given on the * MODIFY card cannot be found in the INCLUDE file, the * MODIFY card is ignored and a warning message printed.

The work file, WK, must be furnished when modifying the INCLUDE file.

SAVE Control Card

The * SAVE control card is used to copy the entire INCLUDE file to the file specified by the file-code. The format of the * SAVE control card is as follows:

```
1      8       16
*      SAVE    file-code[,BCD]
```

The file is organized in the system standard format, and unless BCD is specified as a parameter, the file is represented in the ASCII character set.

An INCLUDE file is saved as a series of control cards and macro text, as follows:

```
1      8       16
_________________________________

*      INITIAL
*      CREATE  text-name-1

       text-1

*      CREATE  text-name-2

       text-2

*      CREATE  text-name-3

       text-3

         .
         .
         .
```

An INCLUDE file that has been saved, therefore, can be used as the file identified by the file code IN to produce an INCLUDE file.

EXAMPLES

Examples that illustrate the creation and maintenance of an INCLUDE file are included in this section. The first example illustrates the creation of the INCLUDE file. In the next example, the text of several macros is modified. Next, the INCLUDE file is saved, several more changes are made, and the file is saved again. The first INCLUDE file that was saved first is then used in a PL/I program.

Example 1 - Creation of an INCLUDE File

In this example, the INCLUDE file is initialized, and then three macros are added.

```
1      8       16
________________________________________

$      PROGRAM SRCLIB
$      PRMFL   .L,W,R,MY/INCL
$      DATA    IN
*      INITIAL
*      CREATE  TEXT1
       DCL X1 FIXED;
       DCL D1 FIXED DECIMAL;
       DCL B1 BIT(1);
*      CREATE  TEXT2
P1:    PROC1;
       DCL A FIXED;
       DCL B FIXED;
       A=B*SQRT(B);
       END;
*      CREATE  TEXT3
       DCL E1 ENTRY(FIXED);
       DCL E2 ENTRY(FIXED,FIXED);
$      ENDJOB
```

Example 2 - Modification of an INCLUDE File

In this example, line 2 of the macro TEXT1 is replaced and a %INCLUDE statement is inserted in the macro TEXT2 before line 3. Then the macros that have been changed are listed. Note that, since this job involves modification of the INCLUDE file, the work file, WK, must be included in the job.

```
1      8       16
_____________________________________________

$      PROGRAM SRCLIB
$      PRMFL   .L,W,R,MY/INCL
$      FILE    WK,A1R,10L
$      DATA    IN
*      MODIFY  TEXT1
*      ALTER   2,2
       DCL C1 CHAR(3);
*      MODIFY  TEXT2
*      ALTER   3
       %INCLUDE TEXT3;
*      LIST    TEXT1
*      LIST    TEXT2
$      ENDJOB
```

Example 3 - Saving the INCLUDE File

In this example, the INCLUDE file MY-INC1 is saved as the file identified by the file code XY; several experimental changes are made in the file, including the deletion of a macro; and the new INCLUDE file MY-INC2 is saved as the file identified by the file code YZ.

```
1      8       16
_____________________________________________

$      PROGRAM SRCLIB
$      PRMFL   .L,W,R,MY/INCL
$      FILE    WK,A3R,10L
$      TAPE    XY,A1D,,99999,,MY-INC1
$      TAPE    YZ,A2D,,99999,,MY-INC2
$      DATA    IN
*      SAVE    XY,BCD
*      LIST
*      MODIFY  TEXT1
*      ALTER   1,1
*      DELETE  TEXT3
*      SAVE    YZ,BCD
*      LIST
$      ENDJOB
```

Example 4 - Use of a Saved INCLUDE File

In this example, the INCLUDE file saved in the previous run is re-instated and a program referencing the INCLUDE file given.

```
1      8       16
$       PROGRAM SRCLIB
$       PRMFL   .L,W,R,MY/INCL
$       TAPE    IN,A1D,,99999,,MY-INC1
$       PL1     LIST
$       PRMFL   .L,R,R,MY/INCL
 P:   PROC OPTIONS(MAIN);
        %INCLUDE TEXT1;
        X1 = 1;
        D1 = 1;
        %INCLUDE TEXT2;
        X1=2;
      END;
$       ENDJOB
```

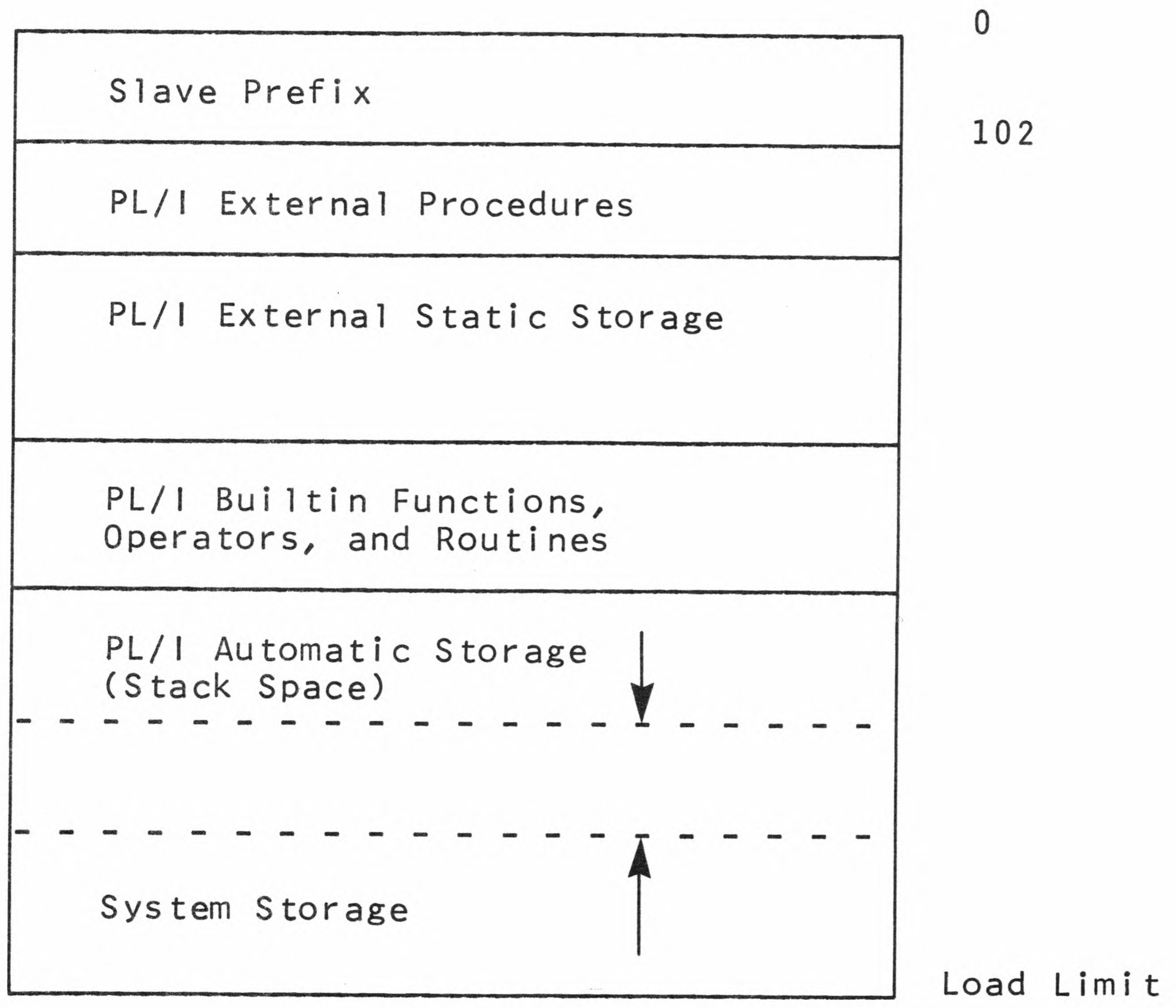
SECTION XIII

DEBUGGING PL/I PROGRAMS

When the execution of a PL/I program terminates abnormally, the execution report produced can be used to obtain useful information about the state of the program upon termination. This section gives general rules for interpreting an execution report and then illustrates the use of some of these rules with an example of a program that terminated on the occurrence of the ZERODIVIDE condition.

MEMORY LAYOUT

The memory layout during the execution of a PL/I program is given here. Later in this section, the actual memory layout for an example is diagrammed.



If the low end of system storage nears the stack frames, then additional memory for system storage is requested from the operating system. When the stack frame space is exhausted or system storage cannot be obtained, the activity is terminated with abort code PC.

ABNORMAL TERMINATION

When a program terminates abnormally, the reason for the termination is indicated either by an abort code or by a message from a default ON unit. Following this identification, an error trace-back is given, which indicates the procedures that were active at the time the termination occurred. Following the error trace-back, a memory dump is listed if the DUMP option is specified on the $ EXECUTE control card.

Abort Codes

If the compilation or execution of a PL/I program terminates abnormally for reasons not handled as conditions, an abort code is listed. Table 13-1 lists the abort codes and gives, for each code, its meaning and the time at which it can occur. If the code can occur during compilation, an X appears in that column; if during execution, an X appears in the execution column.

Table 13-1. PL/I Abort Codes

Code	Meaning	OCCURS DURING	
		Compilation	Execution
PA	Argument and parameter do not match.		X
PC	Core resource exhausted. Try extending the core limit.	X	X
PE	ERROR condition has occurred and user did not supply ON-unit for the condition.		X
PL	Fatal source program error.	X	
PX	Compiler interface detected an unrecoverable error. The system prints a brief comment on the file P*.	X	
S1	Illegal control card on the file A*.	X	
S2	Illegal $ ALTER card on the file A*.	X	
S3	Illegal media code on the file A*.	X	
S4	Illegal media code on the file S*.	X	
S5	Illegal binary card, other than type 5, on the file S*.	X	
S6	Invalid sequence number on the file S*.	X	
S7	Illegal COMDK format on the file S*.	X	

ON Units

Many exceptional conditions can be detected during the execution of a PL/I program. The detection of an enabled condition causes the established ON unit for that condition to be executed. If the user supplies an ON unit for the handling of the condition when it is signalled, that ON unit is executed; otherwise, the system ON unit is executed.

In general the system-supplied ON unit prints an identifying message of the form:

```
**** SIZE CONDITION(ONCODE = 703) OCCURRED.****
```

The message gives the condition name and number. The condition numbers are assigned according to the following list:

Condition Number	Support Routine
1 - 100	Math library
101 - 300	Record and stream I/O
301 - 600	I/O run-time support
601 - 999	PL/I operators
1000	Signal statement
1001 -	Not assigned

804

Appendix I of this manual gives, for each ONCODE number, a more complete description of the reasons for its occurrence. Following the printing of the message, the system-supplied ON unit signals the ERROR condition, which prints the error trace-back and returns to GCOS for the termination of the job.

Error Trace-Back

Following the line that identifies the reason for the abnormal termination of the execution, an error trace-back is given. The error trace-back lists the PROCEDURE blocks that were active at the time the execution terminated, including any PROCEDURE or BEGIN blocks internally created by the compiler. The PROCEDURE blocks are listed in the order in which they were activated, the first being the PROCEDURE with the OPTIONS(MAIN) attribute and the last being the system routine that was activated when the execution was terminated.

For each block the following information is given:

ENTRY NAME — The name of the procedure or block.

LINE # — If the procedure was compiled with the SNUMBER option, the line number in the source listing at which the ERROR condition was signalled is given.

STATEMENT # — If the procedure was compiled with the SNUMBER option, the statement number within the line is given.

LOCATION — The absolute address in memory of the instruction at which either the ERROR condition was signalled or transfer was made to the next block listed in the error trace-back.

OFFSET — The address, on the object listing, of LOCATION relative to the entry point of the activated block.

STACK — The absolute address of the first location of the stack frame assigned to the procedure or block.

The error trace-back is very useful for determining the exact location of the error that terminated the execution and for providing address information to locate PL/I variables. The rules for locating PL/I variables are given in the following paragraphs. Following these rules, a comprehensive example is given that illustrates the use of an error trace-back.

Locating PL/I Variables in Memory

Rules for locating the following types of variables in memory are given in this section:

EXTERNAL STATIC variables
EXTERNAL PROCEDURES
INTERNAL STATIC variables
LABELS
INTERNAL PROCEDURES
AUTOMATIC variables
EXTERNAL PROCEDURE arguments
INTERNAL PROCEDURE arguments

Locating a memory address requires reference to several sections of the compiler output listing. The options given on the $ PL1 control card for the compilation determine the sections of the output listing that are printed. A detailed description of the sections of the compiler output listing and the associated options is given earlier, in the section on the "Compiler".

After the rules for locating the above items are given, a comprehensive example illustrates the location of some PL/I variables in memory by applying these rules to the listings produced from its compilation and execution.

EXTERNAL STATIC VARIABLES

To determine the location in memory of an EXTERNAL STATIC variable, proceed as follows:

1. If the name of the variable exceeds six characters or contains the character '$' or '_', obtain the converted name from the Storage Space and External Symbol section of the compiler output listing. If this listing is not available, convert the name according to the conversion rules given in Appendix F of this manual.

2. Locate the block common with the variable name (or converted variable name) on the Loader Map. The location given to the right of the name is the loaded location for the EXTERNAL STATIC variable. This location will immediately follow the first external procedure in which the variable occurs.

If the EXTERNAL STATIC variable is a structure, continue as follows to locate the members:

3. Obtain the word offset (in octal) and bit offset (in decimal) for the structure member from the Symbol Table section of the compiler output listing.

4. Add the word and bit offset to the origin obtained in Step 2 to locate the member.

EXTERNAL PROCEDURES

To determine the location in memory of an EXTERNAL PROCEDURE, proceed as follows:

1. If the name of the EXTERNAL PROCEDURE exceeds six characters or contains the character '$' or '_', obtain the converted name from the Storage Space and External Symbol section of the compiler output listing. If this listing is not available, convert the name according to the conversion rules given in Appendix F of this manual.

2. Locate the name or converted name on the Loader Map. The location given at the left margin on the Loader Map is the loaded location for the origin of the procedure, including INTERNAL STATIC storage. The location to the right of the name is the loaded location for the entry point to the procedure.

To determine the location of a statement within an EXTERNAL PROCEDURE, continue as follows:

3. Locate the relative location of the statement by consulting the Object Map section of the compiler output listing.

4. Add the relative location for the statement to the procedure origin obtained in Step 2.

When more than one statement is given on a line in the source program, the relative location of the statement can be obtained from the Object Program section of the compiler output listing. The Object Program Listing is annotated for convenient interpretation. The relative location of the instruction is given at the left and the correspondence to the source listing in terms of statement and line number is given on the right of the object code list.

INTERNAL STATIC VARIABLES

To determine the location in memory of an INTERNAL STATIC variable, proceed as follows:

1. Locate the name of the INTERNAL STATIC variable in the Symbol Table section of the compiler output listing to obtain the location of the variable relative to the origin of the procedure.
2. Locate the procedure origin in the Loader Map by following the rules given earlier in this section for locating an external procedure.
3. Add the relative location of the INTERNAL STATIC variable to the procedure origin to obtain the location in memory of the variable.

An INTERNAL STATIC variable is assigned a location in memory only if it is referenced (either explicitly or because an item based on it is referenced). Therefore, INTERNAL STATIC variables that are defined but not referenced do not have a relative location in the Symbol Table section.

LABELS

To determine the location in memory of a LABEL CONSTANT when the Object Program section of the compiler output listing is available, proceed as follows:

1. Locate the label in the Object Program section of the compiler output listing to obtain the location of the label relative to the procedure origin.
2. Locate the procedure origin in the Loader Map by following the rules given earlier in this section for locating an external procedure.
3. Add the relative location of the label to the procedure origin to obtain the location in memory of the instruction so labeled.

If the Object Program section of the compiler output listing is not available, proceed as follows:

1. Locate the label or label array in the Symbol Table section of the compiler output listing under the heading "NAMES DECLARED BY EXPLICIT CONTEXT" to obtain the relative location within the procedure.
2. Locate the procedure origin from the Loader Map by following the rules given earlier in this section for locating an external procedure.
3. If the label is <u>unsubscripted</u>, add the relative location of the label to the procedure origin to obtain the location in memory of the instruction so labeled.

If the label is subscripted and the lower bound of the label array is zero, add the relative location of the label array to the procedure origin to obtain the transfer vector, then add the subscript value to the origin of the transfer vector to obtain the transfer address for the instruction associated with the subscripted label. If the lower bound of the label array is other than zero, a further adjustment must be made.

INTERNAL PROCEDURES

To locate an INTERNAL PROCEDURE, proceed as follows:

1. Locate the relative location of the INTERNAL PROCEDURE in the Symbol Table section of the compiler output listing under the heading "NAMES DECLARED BY EXPLICIT CONTEXT".

2. Locate the origin of the external procedure in the Loader Map following the rules given earlier in this section for locating external procedures.

3. Add the relative location of the internal procedure to the procedure origin of the external procedure to obtain the location in memory for the internal procedure.

AUTOMATIC VARIABLES

To locate the relative location of an AUTOMATIC variable, proceed as follows:

1. Locate the relative location of the AUTOMATIC variable in the Symbol Table section of the compiler output listing.

2. Locate the origin of the stack frame for the current invocation of the procedure from the error trace-back.

3. Add the relative location of the AUTOMATIC variable to the stack frame origin to obtain the location in memory of the AUTOMATIC variable.

If the error trace-back is not available, the stack frame for the current invocation of the procedure can be obtained by following stack frame linkages. The stack frame linkages can be followed either from the first stack frame in a forward direction or from the last stack frame in a backward direction. If a procedure has several active invocations, the current active invocation can be obtained most efficiently by beginning from the last stack frame and working backwards.

The format of the stack frame is as follows:

	0	18
0	AP	LP
1	OC	EL
2	CS	SF
3	SB	TO
4	Temporary Storage for Operators	
5		
6	First word of AUTOMATIC Storage	
	. . .	
	Last word of AUTOMATIC Storage	

where: AP is the location of the calling sequence.

LP is the location of call + 1.

OC is the offset relative to stack frame of the enabled condition chain.

EL is the location of the entry + 1.

CS is the location of the stack frame header for caller (index register SP = 2).

SF is the location of the stack frame header for the last invocation of the enclosing procedure.

SB is the last location in this stack frame (including temporary).

TO is the first location used for temporary (frame extension).

For a more detailed explanation of the stack frame format, refer to Detailed Stack Frame Format, later in this section.

To follow stack frame linkages in a forward direction, proceed as follows:

1. Locate the address of the first stack frame from the upper half of word 37 in the memory dump.

2. Examine the lower half of word 1 (EL) to obtain the location+1 of the entry point of the associated procedure. Compare this address+1 with the address obtained on the Loader Map for the entry to the procedure in question. If the addresses agree, the stack frame for the procedure is located. The stack frame for the current invocation of the procedure is the last stack frame located for the procedure using this method.

3. If the addresses do not agree, pick up the location of the next stack frame from the first half of word 3 (SB). If this address is not an even address, round up to an even address.

4. Return to Step 2.

To follow the stack frame linkages in a backward direction, proceed as follows:

1. Locate the address of the last stack frame from index register 2. If the abnormal termination occurred within a GFRC routine, however, index register 2 no longer has the last stack frame and the stack frame linkages must be followed in a forward direction.

2. Compare the addresses as in Step 2 for forward linking.

3. If the addresses do not agree, pick up the location of the preceding stack frame from the first half of word 2 (CS).

The address obtained from the stack frame field EL can be used to obtain the name of the external procedure from the memory dump. Consider the sequence of instructions preceding the procedure entry for a procedure named CALCULATE.

```
e-4    ASCII CALC
e-3    ASCII ULAT
e-2    ASCII E
e-1    ZERO  number of parameters, number of characters in name (= 9)
e      TSX0  .P0090 (external entry operator)
e+1    ZERO  0, number of words of automatic storage used
e+2    instructions for first executable statement
```

The field EL in the stack frame contains the address e+1. Subtracting 2 from the address in EL gives the location in which the number of characters in the name is stored. The number of characters in the name determines the number of words used to store the name, and thus the name of the external procedure can be obtained.

EXTERNAL PROCEDURE ARGUMENTS

To determine the location in memory of an argument of an EXTERNAL PROCEDURE, proceed as follows:

1. Locate the relative location of the argument list within the stack frame of the calling procedure. This can be done in one of two ways, depending on the availability of the object program listing.

   If the Object Program section of the compiler output listing for the calling program is available, the relative location can be obtained from the generated code as follows:

   ```
   .
   .
   .
   EAX6     n,SP
   TSXLP    procedure
   .
   .
   .
   ```

   The number n is the relative location (in decimal) within the stack frame of the calling procedure for the argument list for the called procedure. Add the relative location to the stack frame origin of the calling procedure.

   If the Object Program listing is not available, information can be obtained from a memory dump. The upper half of word 0 of the stack frame of the called procedure gives the location of the argument list within the stack frame of the calling procedure.

2. The format of the argument list is described in detail earlier, in the Section on "Linking PL/I and Other Languages". Given the origin of the argument list, the description of Section on "Linking PL/I and Other Languages" gives the information necessary to locate any argument within that list.

The location of an argument of an external procedure is illustrated in the examples that conclude this section.

INTERNAL PROCEDURE ARGUMENTS

To determine the location in memory of an argument of an INTERNAL PROCEDURE, consult the Storage and External Symbol section of the compiler output listing to see whether or not the INTERNAL PROCEDURE shares the stack frame of the enclosing procedure. Then proceed as follows:

1. If a stack frame is created when the internal procedure is called, then the rules for locating an argument are identical to the rules for locating an argument of an external procedure.

2. However, if the stack frame is shared, the location within the stack frame of the calling procedure for the argument list can be obtained in one of two ways, depending on the availability of the Object Program listing.

   If the Object Program section of the compiler output listing for the calling procedure is available, the location can be obtained from the first instructions generated for a shared frame internal procedure, namely:

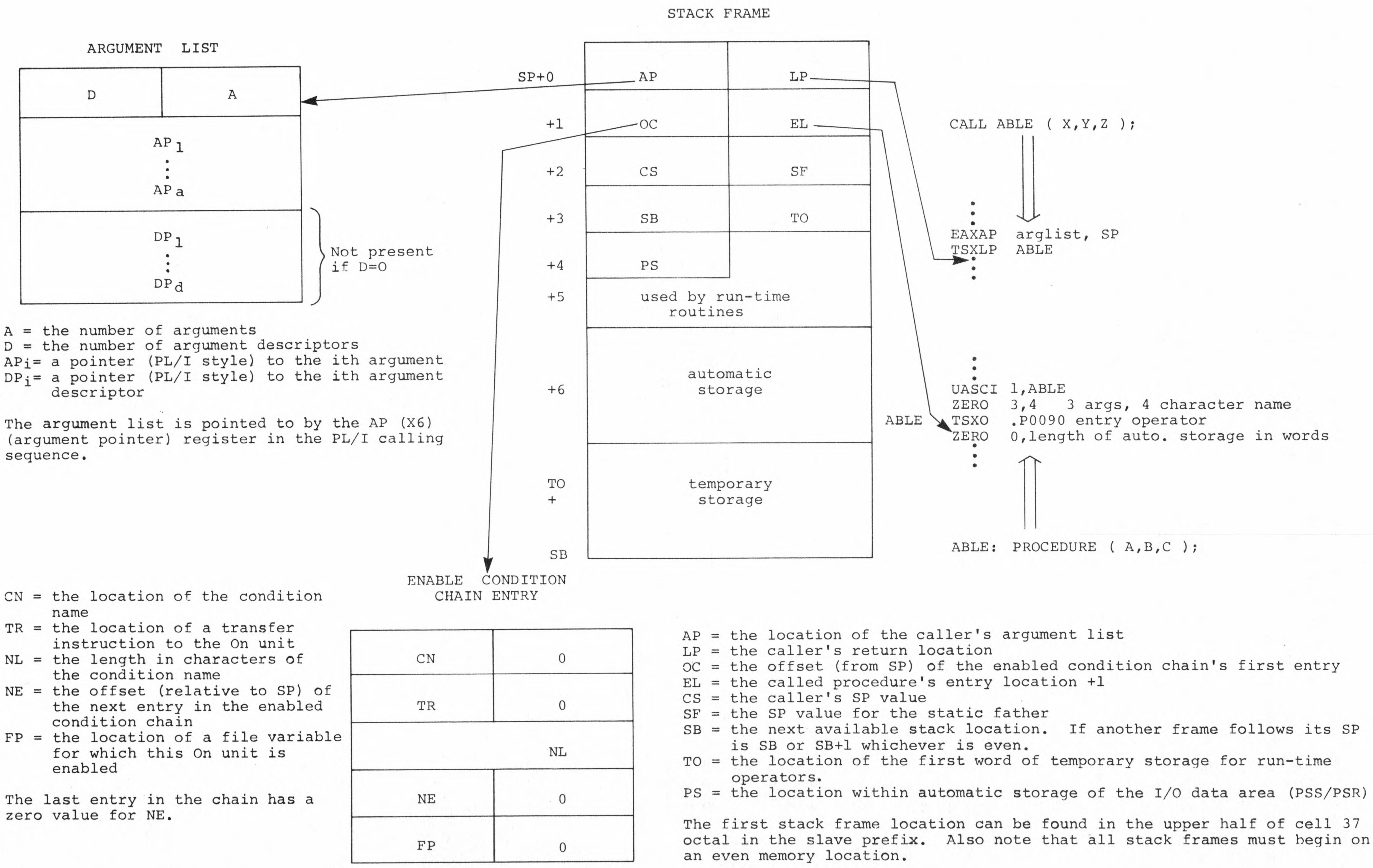
   ```
   STXAP   n,SP
   SXLLP   n,SP
   ```

   The number n is the relative decimal location in the stack frame of the enclosing procedure for the word whose upper half contains the loaded location of the argument list of the called procedure.

   If the Object Program Listing is not available for the calling procedure, the above instructions can be located by consulting the Object Program Map Listing. The source line in the map for the internal procedure locates the above instructions. The loaded location of these instructions, and hence the value of n, can then be obtained from a memory dump.

Detailed Stack Frame Format

A detailed diagram of the stack frame format is given in Figure 13-1. The relationship of the stack frame to the argument list, calling sequence, and enabled condition names is also illustrated in this figure.



A = the number of arguments
D = the number of argument descriptors
APi= a pointer (PL/I style) to the ith argument
DPi= a pointer (PL/I style) to the ith argument
descriptor

The argument list is pointed to by the AP (X6) (argument pointer) register in the PL/I calling sequence.

CN = the location of the condition name
TR = the location of a transfer instruction to the On unit
NL = the length in characters of the condition name
NE = the offset (relative to SP) of the next entry in the enabled condition chain
FP = the location of a file variable for which this On unit is enabled

The last entry in the chain has a zero value for NE.

AP = the location of the caller's argument list
LP = the caller's return location
OC = the offset (from SP) of the enabled condition chain's first entry
EL = the called procedure's entry location +1
CS = the caller's SP value
SF = the SP value for the static father
SB = the next available stack location. If another frame follows its SP is SB or SB+1 whichever is even.
TO = the location of the first word of temporary storage for run-time operators.
PS = the location within automatic storage of the I/O data area (PSS/PSR)

The first stack frame location can be found in the upper half of cell 37 octal in the slave prefix. Also note that all stack frames must begin on an even memory location.

Figure 13-1. Detailed Stack Frame Diagram

The PL/I condition signalling mechanism makes use of several external variables which may be of interest. Their converted names and uses are listed in Table 13-2. Their location in memory may be determined from the load map where their Block Common items usually first appear in relation to module ZLKV. These Block Commons will contain values related to the most recently signalled condition to which each is pertinent.

Table 13-2. Frequently-Used Block Common Items

Name	Data Type	Value
OCODE	FIXED BINARY	the oncode.
9ONDEX	FIXED BINARY	index in 6OURCE of the offending character.
6OURCE	CHARACTER(256) VARYING	bad string causing conversion error.
4OFILE	CHARACTER(32) VARYING	file name for which the CONVERSION, NAME, ENDFILE, TRANSMIT, RECORD, KEY, or UNDEFINEDFILE condition has been signalled.
3ONLOC	CHARACTER(256) VARYING	a character string containing the name of the faulting procedure.
7OIELD	CHARACTER(256) VARYING	the bad identifier in the GET DATA statement.
3ONKEY	CHARACTER(256) VARYING	the character string containing the key of the record for which the ENDFILE, TRANSMIT, or ONKEY condition has been signalled.

EXAMPLE

The following example is intended to illustrate the output resulting from an abnormal termination of a PL/I program. The computations being performed in the program are of no interest, except that the third execution of P2A is intended to abort the third time through to provide the execution report.

The job consists of the compilation of four separate external procedures and the execution of the results of the compilations. Figure 13-2 gives the control cards and input cards for this job.

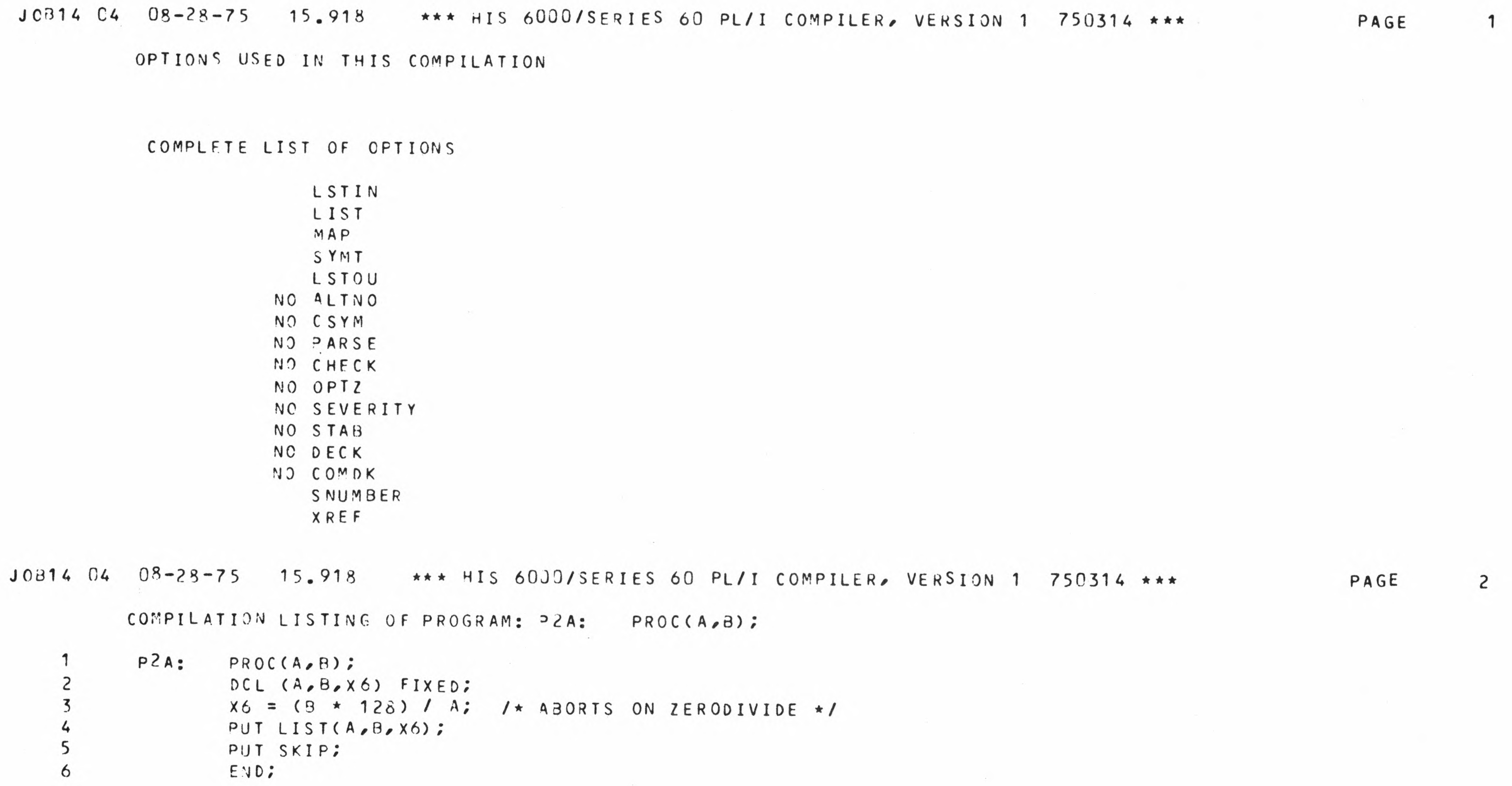
```
1      8       16

$      SNUMB   JOB14
$      IDENT
$      OPTION  PL1
$      PL1     LIST,SNUMBER
P:    PROC OPTIONS(MAIN);
       DCL P1 ENTRY(FIXED);
       DCL P2 ENTRY(FIXED,FIXED);
       DCL (X1,X2,X3) FIXED;
       DO X1 = 1 BY 1 TO 5;
            X2 = X1**2;
            CALL P1(X2);
            END;
       DO X3 = 2 BY 2 TO 50;
            X2 = X3 - 6;
            CALL P2(X2,X3);  /* ABORTS THIRD TIME THROUGH */
            END;
      END;
$      PL1     LIST
P1:   PROC(A);
       DCL (A,X4) FIXED;
       X4 = SQRT(A);
       PUT LIST(X4,A);
       PUT SKIP;
      END;
$      PL1     LIST,SNUMBER
P2:   PROC(A,B);
       DCL (A,B,X5) FIXED;
       DCL P2A ENTRY(FIXED,FIXED);
       X5 = A**2;
       CALL P2A(X5,B);
      END;
$      PL1     LIST,SNUMBER
P2A:  PROC(A,B);
       DCL (A,B,X6) FIXED;
       X6 = (B * 128) / A;  /* ABORTS ON ZERODIVIDE */
       PUT LIST(A,B,X6);
       PUT SKIP;
      END;
$      EXECUTE DUMP
$      LIMITS  10,40K,-2K
$      ENDJOB
***EOF
```

Figure 13-2. Deck Setup for Example

For each external procedure, the option LIST is specified on the $ PL1 control card. Therefore, the compiler output listing for each procedure contains the following sections:

Option listing
Source Program listing
Symbol Table listing
Storage and External Symbol listing
Object Program Map listing
Object Program listing

Figure 13-3 contains the compiler output listing, composed of these sections, for the external procedure P2A. Each of the other external procedures produces the same logical sections of the compiler output listing. The listing, as it appears in Figure 13-3, is somewhat compressed, but all the information is retained. This listing is used later in the section to locate some variables in memory.



```
JOB14 C4  08-28-75    15.918     *** HIS 6000/SERIES 60 PL/I COMPILER, VERSION 1  750314 ***            PAGE      1
          OPTIONS USED IN THIS COMPILATION


           COMPLETE LIST OF OPTIONS

                          LSTIN
                          LIST
                          MAP
                          SYMT
                          LSTOU
                       NO ALTNO
                       NO CSYM
                       NO PARSE
                       NO CHECK
                       NO OPTZ
                       NO SEVERITY
                       NO STAB
                       NO DECK
                       NO COMDK
                          SNUMBER
                          XREF

JOB14 04  08-28-75    15.918     *** HIS 6000/SERIES 60 PL/I COMPILER, VERSION 1  750314 ***            PAGE      2

          COMPILATION LISTING OF PROGRAM: P2A:    PROC(A,B);

     1    P2A:     PROC(A,B);
     2             DCL (A,B,X6) FIXED;
     3             X6 = (B * 128) / A;   /* ABORTS ON ZERODIVIDE */
     4             PUT LIST(A,B,X6);
     5             PUT SKIP;
     6             END;
```

Figure 13-3. Compiler Output Listing for Example

```
IDENTIFIER              OFFSET      LOC STORAGE CLASS         DATA TYPE           ATTRIBUTES AND REFERENCES

*NAMES DECLARED BY DECLARE STATEMENT*
A                                       PARAMETER       FIXED BIN(17,0)           DCL 2 REF 1 3 4
B                                       PARAMETER       FIXED BIN(17,0)           DCL 2 REF 1 3 4
X6                               000006 AUTOMATIC       FIXED BIN(17,0)           DCL 2 SET REF 3 4

*NAME DECLARED BY EXPLICIT CONTEXT*
P2A                              000051 CONSTANT        ENTRY                     EXTERNAL DCL 1 REF 1

*NAME DECLARED BY CONTEXT OR IMPLICATION*
SYSPRINT                         000002 CONSTANT        FILE                      SET REF 0 4 5


J0614 04  08-28-75   15.918     *** HIS 6000/SERIES 60 PL/I COMPILER, VERSION 1   750314 ***              PAGE      4

         COMPILATION LISTING OF PROGRAM: P2A:    PROC(A,B);


*STORAGE REQUIREMENTS FOR THIS PROGRAM*

OBJECT PROGRAM SIZE IS 89 WORDS. (V COUNT 5)

EXTERNAL PROCEDURE 'P2A' USES 50 WORDS OF AUTOMATIC STORAGE

*THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM*
FX1←TO←FL2           EXT←ENTRY           RETURN←MAC           PUT←LIST←NP←AL      PUT←TERMINATE        FL2←TO←FXSCALED
PUT←PREP

*NO EXTERNAL ENTRIES ARE CALLED BY THIS PROGRAM*

*THE FOLLOWING EXTERNAL VARIABLES ARE USED BY THIS PROGRAM*
SYSPRINT             SYSPRINT#

*EXTERNAL NAMES AND CONVERTED NAMES OF THEM*

SYSPRINT                                                      8SRINT

*OBJECT MAP*

LINE SIZE  LOC    LINE SIZE  LOC    LINE SIZE  LOC    LINE SIZE  LOC    LINE SIZE  LOC    LINE SIZE  LOC    LINE SIZE  LOC
   1  4 000047       0  3 000057       3 13 000064       4  5 000101       5  7 000120       6  1 000130
```

Figure 13-3 (cont). Compiler Output Listing for Example

```
        COMPILATION LISTING OF PROGRAM: P2A:    PROC(A,B);

SOURCE ID. TABLE
    000000  000000000005     000
    000001  000047000021     010
    000002  000064000061     010
    000003  000101000101     010
    000004  000120000121     010
    000005  000130000141     010
    000006  000000000000     000

CONSTANTS
    000007  010000000021     000     DESC-RXBS

    000010  000000000000     000     BIT STRG
    000011  123131123120     000
    000012  122111116124     000
    000013  040040040040     000
    NEXT 5 WORDS ARE SAME AS ABOVE
    000021  000000000000     000
    NEXT 1 WORD IS SAME AS ABOVE

SYMBOL TABLES
    000023  000035000033     010
    000024  000036000473     010
    000025  061057064057     000
    000026  065000000000     000
    000027  060070057062     000
    000030  070057067065     000
    000031  000000000000     011
    NEXT 2 WORDS ARE SAME AS ABOVE
    000034  000047000000     010
    000035  160154061002     000
    000036  150151163040     000
    000037  066060060060     000
    000040  057163145162     000
    000041  151145163040     000
    000042  066060040160     000
    000043  154057151056     000
    000044  040166145162     000
    000045  163151157156     000
    000046  040061056000     000
```

Figure 13-3 (cont). Compiler Output Listing for Example

```
BEGIN PROCEDURE 'P2A'
ENTRY TO 'P2A'                                                                   STATEMENT 1 ON LINE 1
     000047  120 062 101 040  000     P2A
     000050  000002 000003    000     ZERO     2,3
     000051  030000 7000 00   030     TSXBP    .P0090                           EXT+ENTRY
     000052  000023 000062    010     ZERO     19,50
     000053  000022 4500 12   000     STZ      18,SP
     000054  000022 7420 12   000     STXSP    18,SP
     000055  000007 6200 12   000     EAXBP    7,SP




JOB14 04  08-23-75   15.918    *** HIS 6000/SERIES 60 PL/I COMPILER, VERSION 1  750314 ***           PAGE     6

          COMPILATION LISTING OF PROGRAM: P2A:    PROC(A,B);

     000056  000004 7400 12   000     STXBP    4,SP
     000057  010000 6200 00   030     EAXBP    SYSPRINT#
     000060  000000 6360 10   000     EAQ      0,BP
     000061  020000 7560 00   030     STQ      SYSPRINT
     000062  777726 6360 04   000     EAQ      -42,IC                    000010
     000063  010003 7560 00   030     STQ      SYSPRINT#+3
                                                                         STATEMENT 1 ON LINE 3
     000064  000000 2210 12   000     LDXLP    0,SP
     000065  000001 2360 31   000     LDQ      1,LP*
     000066  040000 7010 00   030     TSXLP    .P0032                    FX1+TO+FL2
     000067  000060 4570 12   000     DFST     48,SP
     000070  000000 2210 12   000     LDXLP    0,SP
     000071  000002 2360 31   000     LDQ      2,LP*
     000072  000007 7360 00   000     QLS      7
     000073  040000 7010 00   030     TSXLP    .P0032                    FX1+TO+FL2
     000074  000060 5670 12   000     DFDV     48,SP
     000075  050000 7010 00   030     TSXLP    .P0276                    FL2+TO+FXSCALED
     000076  072000 0000 00   000     ARG      29696
     000077  000052 7330 00   000     LRS      42
     000100  000006 7560 12   000     STQ      6,SP
```

Figure 13-3 (cont). Compiler Output Listing for Example

```
                                                                      STATEMENT 1 ON LINE 4
  000101  020000 6360 00  030     EAQ      SYSPRINT
  000102  000020 7560 12  000     STQ      16,SP
  000103  412000 2350 03  000     LDA      136192,DU
  000104  000014 6200 04  000     EAXBP    12,IC                      000120
  000105  060000 7010 00  030     TSXLP    .P0386                     PUT+PREP
  000106  777701 2360 04  000     LDQ      -63,IC                     000007
  000107  000000 2210 12  000     LDXLP    0,SP
  000110  000001 6200 31  000     EAXBP    1,LP*
  000111  070000 7010 00  030     TSXLP    .P0115                     PUT+LIST+NP+AL
  000112  000000 2210 12  000     LDXLP    0,SP
  000113  000002 6200 31  000     EAXBP    2,LP*
  000114  070000 7010 00  030     TSXLP    .P0115                     PUT+LIST+NP+AL
  000115  000006 6200 12  000     EAXBP    6,SP
  000116  070000 7010 00  030     TSXLP    .P0115                     PUT+LIST+NP+AL
  000117  100000 7010 00  030     TSXLP    .P0249                     PUT+TERMINATE
                                                                      STATEMENT 1 ON LINE 5
  000120  020000 6360 00  030     EAQ      SYSPRINT
  000121  000020 7560 12  000     STQ      16,SP
  000122  000001 2360 07  000     LDQ      1,DL
  000123  000017 7560 12  000     STQ      15,SP
  000124  402200 2350 03  000     LDA      132224,DU
  000125  000003 6200 04  000     EAXBP    3,IC                       000130
  000126  060000 7010 00  030     TSXLP    .P0386                     PUT+PREP
  000127  100000 7010 00  030     TSXLP    .P0249                     PUT+TERMINATE
                                                                      STATEMENT 1 ON LINE 6
  000130  110000 7100 00  030     TRA      .P0098                     RETURN+MAC
END PROCEDURE 'P2A'

        * * * * * END ENDE FIM FIN FIN FINE FINIS OWARI * * * * *


JOB14 04  08-23-75   15.918    *** HIS 6000/SERIES 60 PL/I COMPILER, VERSION 1  750314 ***          PAGE     7

        COMPILATION LISTING OF PROGRAM: P2A:   PROC(A,B);


WARNING 75
THE UNDECLARED IDENTIFIER 'SYSPRINT' HAS BEEN CONTEXTUALLY DECLARED AS A FILE CONSTANT. IT WILL ACQUIRE DEFAULT
ATTRIBUTES.

WARNING 495
IMPLEMENTATION RESTRICTION: LONG EXTERNAL NAME 'SYSPRINT' HAS BEEN CONVERTED TO A 6 CHARACTER NAME. RESTRICTIONS ARE:
EXTERNAL FILE NAME SIZE <= 5 AND OTHER EXTERNAL NAME SIZE <= 6.


**  66K WAS USED TO COMPILE THIS PROGRAM.
```

Figure 13-3 (cont). Compiler Output Listing for Example

Figure 13-4 contains the relevant portion of the Loader Map. This listing is used later in the section to obtain the loaded locations for the origin and entry points of the external procedures.

```
JOB14  05 08-23-75   15.922                                                                            PAGE      1

   ORIGIN   DATE MODULE ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION

                              SUBPROGRAMS INCLJDED IN DECK.

                        $      OPTIONS PL1
   000102 75/08/28 0000 ...... 000143     P      000143
   000226 75/08/28 0000 P1     000244
         BLOCK COMMON   9SINT# 000324     8SRINT 000420
   000422 75/08/28 0000 P2     000456
   000510 75/08/28 0000 P2A    000561
         BLOCK COMMON   9SINT# 000324     8SRINT 000420

                              SUBPROGRAMS OBTAINED FROM SYSTEM LIBRARY
   000642 12/02/74 ZLAU SQRT.  000655
JOB14  05 08-28-75   15.922                                                                            PAGE      5

   ORIGIN   DATE MODULE ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION


   050746 04/25/73 G50R .GR250 050746
   051022 75/02/03 G27R .GR275 051022
   051274 75/02/05 G37R .GR377 051332     .GR375 051274     .GR37X 051351     .GR390 051371
   051456 74/12/03 G60R .GR960 051463     .GABTB 051456
   051500 75/02/05 G80R .GR980 051500     .GR979 051603     .GR99X 051504     .GR984 051551     .GR985 051603
                        .GR999 051517     ASCII  051601     ASCRPT 051606     NORRPT 052066
   052074 75/04/11 G90R .GR990 052074     .GR991 052115
   052126 04/29/72 GLAB .GINHD 052133     .GOUTH 052132     .GINTL 052131     .GOUTL 052130     .GUSWH 052127
                        .GOVRL 052134     .GLREA 052224     .GRCVY 052126     .GRPRV 052163
   052766 03/07/66 GINI .GINID 052766
                                                RANGE              SIZE
                        ALLOCATED CORE    000000 THRU 117777      120000
                           RELOCATABLE    000100 THRU 052767      052670

                        FCB AND BUFFER SPACE
                           AVAILABLE      052770 THRU 117777      045010
                           FILE CTRL BLKS 052770 THRU 053120      000131
                           MAXIMUM BUFFER SPACE REQUIRED          001202

                            23K, IS THE MINIMUM MEMORY NEEDED TO LOAD THIS ACTIVITY WITH ALL FILES OPEN     740808 2/H
                        002074 LOCATIONS REQUIRED FOR LOAD TABLE
                        EXECUTION PROGRAM ENTERED AT  000143   THROUGH  .PSETU
```

Figure 13-4. Loader Map for Example

Figure 13-5 contains the execution report produced upon the abnormal termination of the program. As planned, the termination occurred on the third execution of the external procedure P2A.

First, the system ON unit for ZERODIVIDE prints an identifying message, then signals the ERROR condition. The system ON unit for ERROR prints the error trace-back and returns control to GCOS. Since the DUMP option is given on the $ EXECUTE control card for the run, a memory dump is produced. The memory dump in Figure 13-4 has been edited for inclusion in this manual so that only the relevant portions appear.

Following the memory dump, the information output by the program on SYSPRINT is listed (see last page of Figure 13-5).

```
**** ZERODIVIDE CONDITION(ONCODE = 800) OCCURRED. ****

  * REGISTERS AT SIGNAL TIME *

  EI 000060567012  OI 002622701000  IC 000605  IR 402000  BA 320120  ER 024  AR 300000000000  QR 000000000000  TR 00013374
  X0 000324  X1 000604  X2 053154  X3 000000  X4 000000  X5 000000  X6 053150  X7 000000

  * TRACE OF CALLS IN FORWORD ORDER *

      *  ENTRY NAME                                            LINE #   STATEMENT #   LOCATION   OFFSET    STACK

         P                                                         11             1     000217    000054    053120
         P2                                                         5             1     000505    000027    053140
         P2A                                                        3             1     000604    000023    053154
         PL1+SIGNAL+                                                                    005132    001263    053250

      *  END OF TRACE


  MA 000114  MB 000114  BE 000000
  EI 204725236007  OI 000010001000  IC 015125  IR 002001  BA 320120  ER 000  AR 040040040040  QR 000000204725  TR 00013764
  X0 010025  X1 011230  X2 054770  X3 000000  X4 055163  X5 000000  X6 012444  X7 000000

*EIS REGISTERS*

000000      000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
000010      000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000004  000000000000


000000S     000000000000  001454710000  000000000000  000000000000  000000000000  001466710000  000605402000  001236710000
000010      000000000000  001257710000  000000000000  015125024725  000000000000  000045062114  000000000000  053064000001
000020      000000400000  000000000000  015125002001  000000000000  000143400000  052771000000  000000000000  002665000000
000030      000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  054146220104  053120117777
000040      002364002020  002364203753  777212204000  000020000000  021567200200  000175000001  776000000000  000136623000
000050      014576000001  000001014560  014571000000  014506000000  000000000000  014576014576  000000000000  000111137000
```

Figure 13-5. Execution Report for Example

```
000110     000177000221  000205000241  000210000261  000220000301  000223000321  000127000033  000130000473  061057064057
000120     065000000000  060070057062  070057067065  000102000102  000102000000  000000000000  000141000000  160154061002
000130     150151163040  066060060060  057163145162  151145163040  066060040160  154057151056  040166145162  163151157156
000140     040061056003  120040040040  000000000001  002051700000  000115000020  000001236007  000006756012  000006236012
000150     000005116007  000002600004  000025605004  000006636012  000013756012  000047636004  000014756012  000011636012
000160     000015756012  000003236007  000012756012  000012626012  000770701000  000011236012  000007756012  000007636012
000170     000017756012  000001236007  000016756012  000016626012  000244701000  000006054012  777751710004  000002236007
000200     000010756012  000010236012  000062116007  000002600004  000017605004  000010236012  000006176007  000007756012
000210     000007636012  000013756012  000010636012  000014756012  000002236007  000012756012  000012626012  000456701000
000220     000002236007  000010056012  777757710004  002040710000  000000000002  000000000000  010000000021  000000000000
000230     123131123120  122111116124  040040040040  040040040040  040040040040  040040040040  040040040040  040040040040
000240     000000000000  000000000000  120061040040  000001000002  002051700000  000000000067  000022450012  000022742012
000250     000007620012  000004740012  000324620000  000000636010  000420756000  777752636004  000327756000  000000221012
000260     000001236031  003502701000  000057470012  000057636012  000061756012  000066636012  000062756012  000002236007
000270     000060756012  000060626012  000655701000  000066431012  002606701000  000006756012  000420636000  000020756012
000300     412000235003  000011620004  001722701000  777723236004  000006620012  001707701000  000000221012  000001620031
000310     001707701000  001714701000  000420636000  000020756012  000001236007  000017756012  402200235003  000003620004
000320     001722701000  001714701000  002040710000  000000000000  565000000200  000000000000  400000000000  000520000000
000330     777777000000  123131123120  122111116124  040040040040  040040040040  040040040040  040040040040  040040040040
000340     040040040040  120052000000  117172000000  000000000011  000000000103  053064000000  000000000000  000000000000
000350     000000000002  000000000000  117024000000  000000000001  000000000000  000000000204  000000000010  000000000067
000360     000000000001  000000000041  000000000000  000000000003  000000000001  000000000000  000000000000  000000000000
000370     000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
000420*    000324000000  000000000000  000000000004  000454000021  000460000101  000475000121  000506000141  010000000021
000430     000442000033  000443000473  061057064057  065000000000  060070057062  070057067065  000422000422  000422000000
000440     000000000000  000454000000  160154061002  150151163040  066060060060  057163145162  151145163040  066060040160
000450     154057151056  040166145162  163151157156  040061056003  120062040040  000002000002  002051700000  000430000014
000460     000000221012  000001636031  000011756012  000024636004  000012756012  000007636012  000013756012  000003236007
000470     000010756012  000010626012  000770701000  000007236012  000006756012  000006636012  000011756012  000000221012
000500     000002636031  000012756012  000002236007  000010756012  000010626012  000561701000  002040710000  000000000002
000510     000000000005  000557000021  000574000061  000611000101  000630000121  000640000141  000000000000  010000000021
000520     000000000000  123131123120  122111116124  040040040040  040040040040  040040040040  040040040040  040040040040
000530     040040040040  000000000000  000000000000  000545000033  000546000473  061057064057  065000000000  060070057062
000540     070057067065  000510000510  000510000000  000000000000  000557000000  160154061002  150151163040  066060060060
000550     057163145162  151145163040  066060040160  154057151056  040166145162  163151157156  040061056000  120062101040
000560     000002000003  002051700000  000533000062  000022450012  000022742012  000007620012  000004740012  000324620000
000570     000000636010  000420756000  777726636004  000327756000  000000221012  000001236031  003502701000  000060457012
000600     000000221012  000002236031  000007736000  003502701000  000060567012  002622701000  072000000000  000052733000
000610     000006756012  000420636000  000020756012  412000235003  000014620004  001722701000  777701236004  000000221012
000620     000001620031  001707701000  000000221012  000002620031  001707701000  000006620012  001707701000  001714701000
000630     000420636000  000020756012  000001236007  000017756012  402200235003  000003620004  001722701000  001714701000
000640     002040710000  000000000000  002416227677  000462556202  030000000033  000000000000  000552023631  356047675131
000650     776456125464  631101467257  123121122124  137040040040  000002000005  002051700000  000000000023  000000221012
000660     000001431031  400000475003  000006457012  000006433012  000073600004  000006433012  000014605004  000015236007
```

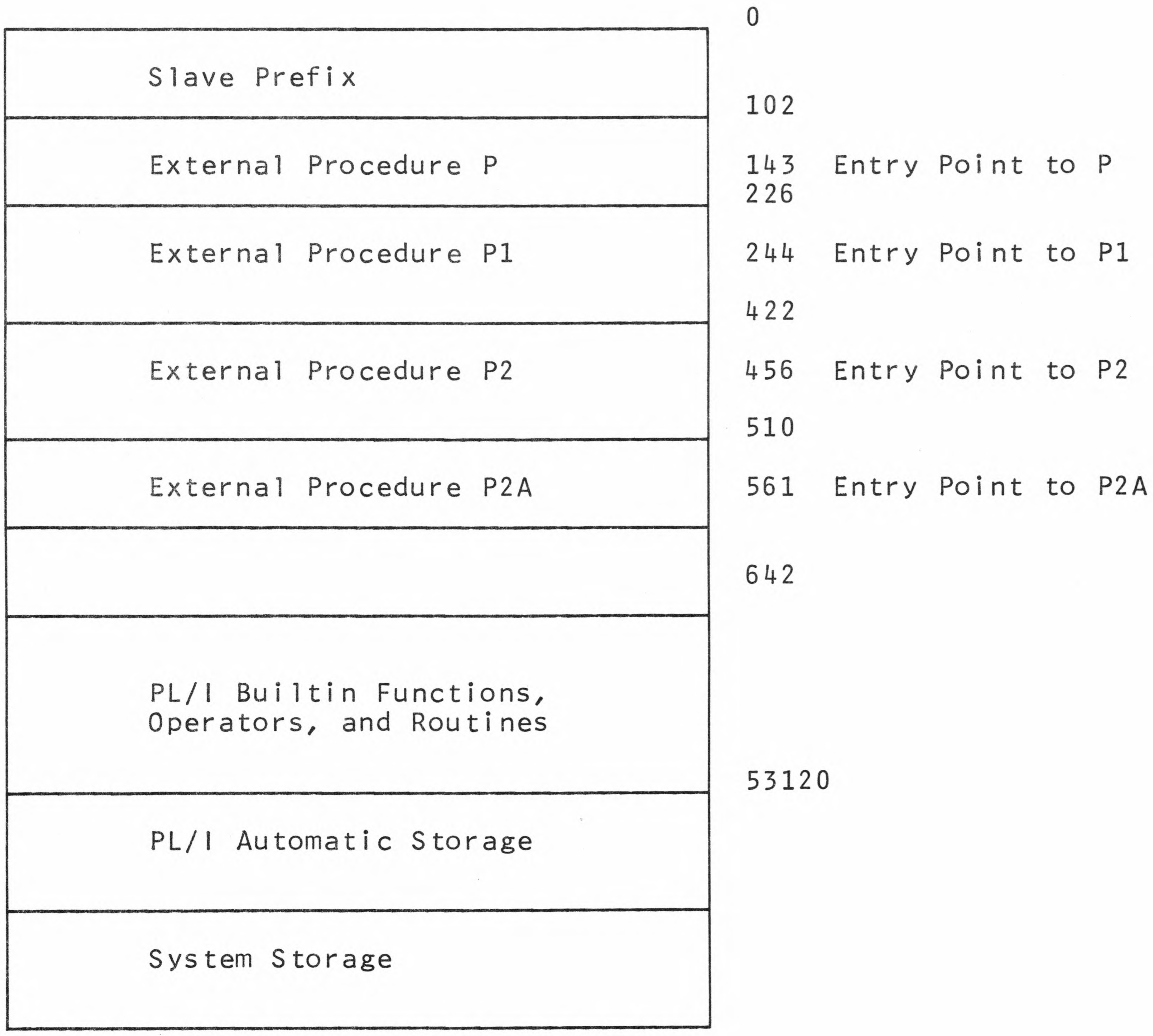
Figure 13-5 (cont). Execution Report for Example

```
053110       000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000500000000  000000000000
053120       001225001223  000000000144  000000000000  053140053140  000000000000  000000000000  000000000006  000000000000
053130       000000000006  000000000031  000000000002  053127000000  053130000000  053131000000  000000000001  053127000000
053140       053132000220  000000000457  053120000000  053154053154  000000000000  000000000000  000000000000  000000000000
053150       000000000002  053146000000  053130000000  053147000000  053150000506  000000000562  053140000000  053247053236
053160       053163000000  000000000605  000000000001  402200000000  000000000000  000000000002  015152000000  000000000000
053170       000640053154  053162000000  000000000000  000000000001  000420000000  053127000000  053154000000  000000000000
053200       000000000000  000000000000  000000000000  000000000000  010000000021  000000000000  000000000000  000000000000
053210       020000000000  053162000000  010000000021  000000000000  000000000000  000000000000  000000000000  012620000000
053220       000000000002  053217000000  053226000000  000000000000  000000000000  000000000000  006500000000  000000000001
053230       053144000000  000322000322  053140000000  000000000000  400000000000  000000000000  000004000004  001437000000
053240       001622000000  001604000000  001623000000  001600000000  001601000000  001602000000  001601000000  000324000000
053250       053236001567  000000003650  053154000000  054434054426  000000000000  000000000000  053250000000  777777000000
053260       777777000000  000000000000  000000000000  000000000000  000000000002  053256000000  053173000000  000000000000
053270       053264026264  000000027124  053250000000  053346053346  000000000000  000000000000  053163000000  062070000000
053300       000324000000  000000000000  053264015334  000000027704  053250000000  053360053360  000000000000  000000000000
053310       053163000000  000000000000  000324000000  000352000000  000000000042  002530000000  000000000001  000000000011
000030       000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  054146220104  053120117777
000040       400000202130  202401203253  000150204000  000020000000  202203002600  000271000001  000000000000  000132607000
000050       400000202130  202401203253  000150204000  000020000000  202203002600  000271000001  000000000000  000132607000
000060       000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  202020440204  232107050706
000070       732343446123  202343204421  414655624231  736263216331  464520232020  202020202020  202020202020  202020202020

**  24K WAS USED TO EXECUTE THIS PROGRAM.



  SNUMB = JOB14, ACTIVITY # = 05, , REPORT CODE = 01, RECORD COUNT = 000007

         1                   1
         2                   4
         3                   9
         4                  16
         5                  25
        16                   2                  16
         4                   4                 128
```

Figure 13-5 (cont). Execution Report for Example

Gross Memory Layout for the Example

The Loader Map, given in Figure 13-3, lists the loaded locations of the user-defined external procedures and support system routines. From this information, the gross memory layout for the job can be diagrammed, as follows:



Error Trace-Back for the Example

The error trace-back for this example (Figure 13-4) indicates that the external procedures P, P2, and P2A were active at the time the ZERODIVIDE condition was signalled. The system routine PL1_SIGNAL_, the last routine executed, handled the condition.

Since the external procedures P, P2, and P2A were compiled with the SNUMBER option, the line and statement number currently being executed are given in the error trace-back. Line 11 in the external procedure P is the call to P2; line 5 in the external procedure P2 is the call to P2A; and line 3 in the external procedure P2A, as can be seen in the Source Program section of the compiler output listing in Figure 13-3, is the calculation that caused the ZERODIVIDE condition to be signalled.

Locating an AUTOMATIC Variable

Consider the location of the variable X6 in the external procedure P2A. X6 is an AUTOMATIC variable, so the rules for locating an AUTOMATIC variable are applied, as follows:

1. The relative location of the variable X6 in the Symbol Table listing of Figure 13-3 is 000006.

2. The origin of the stack frame for the current invocation of P2A from the error trace-back is 053154.

3. The location of the AUTOMATIC variable X6 in memory is then:

```
  053154
+ 000006
  ------
  053162
```

Examination of the dump of Figure 13-5 indicates that location 053162 contains the value 000000000001. Note that this value is meaningless because X6 was not evaluated (and stored) when the ZERODIVIDE condition was signalled. Since X6 is an AUTOMATIC variable, its space may be otherwise used between calls to P2A.

Current Stack Frames for the Example

If the error trace-back for this example were not available, the stack frame for an external procedure could be located by either of the methods described earlier in this section. If the forward method is applied, the location of the first stack frame is obtained from the upper half of word 37 and the results of following the links can be diagrammed, as follows:

53120	1225	1223	Stack Frame for P
	0	144	
	0	0	
	53140	53140	

⋮

53140	53132	220	Stack Frame for P2
	0	457	
	53120	0	
	53154	53154	

⋮

53154	53150	506	Stack Frame for P2A
	0	562	
	53140	0	
	53247	53236	

Note that the lower half of word 1 contains the entry+1 to the associated procedure. Consider the stack frame for P2A. The lower half of word 1 contains 562. The entry to P2A can be obtained from the Loader Map and is found to be 561. Therefore, in the absence of other information, the procedure associated with the stack frame can be determined in this way.

Locating an Argument List

Consider the location of the argument list for the external procedure P2A. The rules for locating an external procedure argument are applied, as follows:

1. Suppose that the Object Program Listing for the calling procedure, P2, is not available.

   The upper half of word 0 of the stack frame for P2A gives the loaded location (053150) within the stack frame of P2 for the argument list.

2. The argument list from the memory dump is:

```
000000000002 - indicating two arguments
053146000000 - location of first argument (A)
053130000000 - location of second argument (B)
```

   From the memory dump the value of the arguments are, as follows:

```
053146  000000000000    (A)
053130  000000000006    (B)
```

Although the parameters are declared to be 17 bits in length, the default alignment assumption of ALIGNED causes them to be represented for ease of access in a full word.

SECTION XIV

EFFICIENCY CONSIDERATIONS

Several measures of efficiency can be applied to a computer program. The program can be efficient in terms of execution time, storage space, or clarity of expression. Sometimes these different measures of efficiency are compatible; sometimes, however, one measure of efficiency must be sacrificed to increase another.

The rules for clarity of expression apply in a general way to all programming languages and do not change from one implementation of PL/I to another. Therefore, rules for clarity of expression are not discussed here. On the other hand, the rules for obtaining efficiency of time and storage are closely related to the design of the host computer and the way the language is implemented. Therefore, these efficiency rules are discussed in this section.

First, some general rules are given for the efficient use of PL/I. Then rules are given that increase one measure of efficiency at the expense of the other measure.

GENERAL RULES FOR IMPROVING EFFICIENCY

The rules given in this section can be applied to improve the general efficiency of a PL/I program.

Data Types

The data type should be chosen to suit the type of operation. In general, the following rules apply:

- Integer values (such as subscripts, counters, and indexes) should be declared FIXED BINARY.

- Noninteger values should be declared FLOAT BINARY, except in the case where exceptional precision is required; for exceptional precision, FLOAT DECIMAL should be used. (However, be aware of potential incompatibility with a future PL/I system. See Section X at the end of the paragraphs on DATA.)

- A numeric picture variable should not be used in a complicated arithmetic calculation. Picture variables are intended for use in situations in which input-output is important and calculations are simple.

A detailed set of guidelines for the choice of data types can be found in the PL/I Reference Manual.

Data Conversions

Data conversions are time-consuming and should be avoided whenever possible. Some hints for avoiding data conversions follow:

- Avoid unnecessary conversions by carefully matching the data types of variables. Even a difference in the number-of-digits or scale-factor in the precision attribute can cause a conversion to occur.
- Avoid unnecessary assignment to a target that requires promotion of the aggregate type on the right-hand side.
- Avoid unnecessary conversion of arguments in a procedure call or function reference by using by-reference arguments rather than by-value arguments. In order for an argument to be handled by-reference every detail of the storage type of the argument must match the storage type of the corresponding parameter.
- Avoid excessive conversion of pictured values to arithmetic values. As noted in the above paragraph on "Data Types" picture variables are intended for use in situations where input-output is important and calculations are simple.

Varying Strings

The handling of NONVARYING strings is more efficient than the handling of VARYING strings. The use of VARYING strings increases both the amount of storage that must be allocated for the string and the amount of object code that must be generated to handle the string.

Debugging Constructs

Constructs used for debugging should be removed before the program enters production. These constructs include:

- Data-directed stream input-output statements.
- The SNUMBER option, used for error trace-back information.
- Condition prefixes for the SIZE, SUBSCRIPTRANGE, STRINGRANGE, and STRINGSIZE conditions.

All of these features are costly in terms of execution time and storage use.

RULES FOR IMPROVING TIME EFFICIENCY

The rules in this section are useful for improving the time efficiency of a PL/I program. Some of these rules improve the execution time of a program at the expense of storage; others, at the expense of program clarity.

Alignment of Structures

Variables with the ALIGNED attribute are stored for efficiency of access. Therefore, if a frequently-accessed variable is normally assigned the default attribute UNALIGNED, the variable should be declared ALIGNED. UNALIGNED is the normal default alignment for nonvarying strings and structures. More efficient code can be generated if level 01 structures are declared ALIGNED.

If a frequently-accessed variable must be declared UNALIGNED, then the value of the variable can be moved to an ALIGNED temporary for access. Consider the variable X in the structure TABLE:

```
DCL  01  TABLE,
         02 X FIXED UNAL,
         02 C1 CHAR(6),
         02 C2 CHAR(8);
```

The structure TABLE occupies four words. The access of X is accomplished by assigning X to the ALIGNED temporary TEMP, as follows:

```
DCL TEMP FIXED;
TEMP=X;
.
.
.
Y=TEMP;
.
.
.
Z=TEMP;
.
.
.
W=TEMP;
.
.
.
```

The subsequent accesses of TEMP are more efficient than accesses of X. However, if the value of TEMP is changed, it must be assigned to X before any access of the variable TABLE.

Blocks and ON Units

BEGIN blocks and ON units involve considerable overhead at activation and termination. Extensive use of such block structure in a program should be avoided if time efficiency is the principal consideration. Internal PROCEDURE blocks, however, are reasonably efficient provided they are not used recursively.

String Assignment

The use of the STRING built-in function to assign string constants to a contiguously stored series of bit strings is more efficient than assignment on an element-by-element basis. Consider the following structure:

```
DCL 01 STR ALIGNED,
       02 X FIXED,
       02 B,
          03 B1 BIT(1) UNAL,
          03 B2 BIT(1) UNAL,
          03 B3 BIT(2) UNAL;
```

The following assignment statement

```
STRING(B)='0101'B;
```

is equivalent to and more efficient than the element-by-element assignment:

```
B1='0'B;
B2='1'B;
B3='01'B;
```

The declaration of a constant with a descriptive name, in this case, is sometimes clearer:

```
DCL CLEAR_MASK BIT (4) INIT ("0101"B);
 .
 .
 .
 .
STRING (B) = CLEAR_MASK;
```

Fixed-Point Multiplication and Division

The use of the MULTIPLY and DIVIDE built-in functions sometimes reduces the number of instructions required for the evaluation of an expression. In particular, the use of the built-in functions is efficient when the operands are single precision FIXED BINARY variables and the result of the operation is also a single precision FIXED BINARY variable. Consider the following statements:

```
DCL (I,J,K,M) FIXED BINARY(18);
 .
 .
 .
M = I/J + K;
M = I*K + J;
```

The following use of the built-in functions is more efficient for this case:

```
M = DIVIDE(I,J,18,0) + K;
M = MULTIPLY(I,K,30,0) + J;
```

Fixed-Point Addition and Subtraction

The precision of intermediate results of arithmetic operations can have an adverse effect upon efficiency. Consider the following example:

```
DCL (I,J,K) FIXED(35);
 .
 .
 .
K = I + J;
```

According to the rules of PL/I, the number-of-digits of the intermediate result of the addition or subtraction operation is one more than the maximum number-of-digits of the operands. In this example, the number-of-digits of the intermediate result requires a double precision number. Therefore, object code is required to convert from single to double precision for the intermediate result and from double to single precision for the assignment to K.

Suppose I, J, and K are declared in the following way:

```
DCL (I,J,K) FIXED(30);
 .
 .
 .
K = I + J;
```

Here, the intermediate result is a single precision number and no conversion is necessary.

Scale-Factor Conversion

Scale-factor conversion can be avoided in the addition and subtraction of decimal fixed-point numbers by declaring the same scale-factor for each variable. For example, consider the following addition:

```
DCL X FIXED DEC(5,2);
DCL Y FIXED DEC(6,1);
DCL Z FIXED DEC(7,3);
 .
 .
 .
Z = X + Y;
```

The following version of these statements is more efficient:

```
DCL X FIXED DEC(6,3);
DCL Y FIXED DEC(8,3);
DCL Z FIXED DEC(7,3);
 .
 .
 .
Z = X + Y;
```

Address Calculation

If a reference with multiple locator-qualifiers is used frequently, the introduction of temporary storage increases the efficiency of the program by eliminating the need for repeated complex address calculations.

Assume that BASE1 and BASE2 are based variables and P1, Q1, and R1 are pointers, and consider the following program fragment:

```
R1->BASE1.R2->BASE2.X = P1->BASE1.P2->BASE2.X
                      + Q1->BASE1.Q2->BASE2.X;
R1->BASE1.R2->BASE2.Y = P1->BASE1.P2->BASE2.Y
                      - Q1->BASE1.Q2->BASE2.Y;
```

The addressing can be made more efficient, in this case, by the use of the pointers, P, Q, and R, as follows:

```
P = P1->BASE1.P2;
Q = Q1->BASE1.Q2;
R = R1->BASE1.R2;

R->BASE2.X = P->BASE2.X + Q->BASE2.X;
R->BASE2.Y = P->BASE2.Y - Q->BASE2.Y;
```

Another example of the effective use of temporary storage follows. Consider, first, the following program fragment:

```
IF SUBSTR(S,I,1)="A" !
   SUBSTR(S,I,1)="B" !
   SUBSTR(S,I,1)="C"
   THEN GOTO L1;
.
.
.
ST = SUBSTR(S,I,1) !! "D";
```

If the result of the SUBSTR function is assigned to a temporary, the resulting program is more efficient:

```
CHAR1 = SUBSTR(S,I,1);
IF CHAR1="A" ! CHAR1="B" ! CHAR1="C"
   THEN GOTO L1;
.
.
.
ST = CHAR1 !!  "D" ;
```

Logical Expressions

The use of general logical expressions involving relational operators should be avoided. A series of simple IF statements is more efficient than a single IF statement with a multiple condition test.

For example, the statement

```
IF X=C1
   THEN IF Y=C2
        THEN IF Z=C3
             THEN GOTO EXIT;
```

is more efficient than

```
IF X=C1 & Y=C2 & Z=C3 THEN GOTO EXIT;
```

Logical expressions involving only BIT variables, however, are efficient.

```
IF B1 & B2 & B3 THEN GOTO EXIT;
```

Tests

The choice of a test can influence the economy with which a program can be stated. For example:

```
IF X  = Y
   THEN DO;
        X = X + 1;
        Y = Y - 1;
        END;
   .
   .
   .
```

is more efficient than

```
IF X=Y
   THEN GOTO L1;
        X = X + 1;
        Y = Y - 1;
L1:
   .
   .
   .
```

Invariant Computations

Any operations that are not associated with the control variable of a loop should be moved outside the loop. For example:

```
DO I = 1 TO 10;
   DO J = 1 TO 100;
      C(I,J) = A(I,J) + B(I,J);
      Z(I) = X(I) + Y(I);
      END;
   END;
```

should be rewritten as

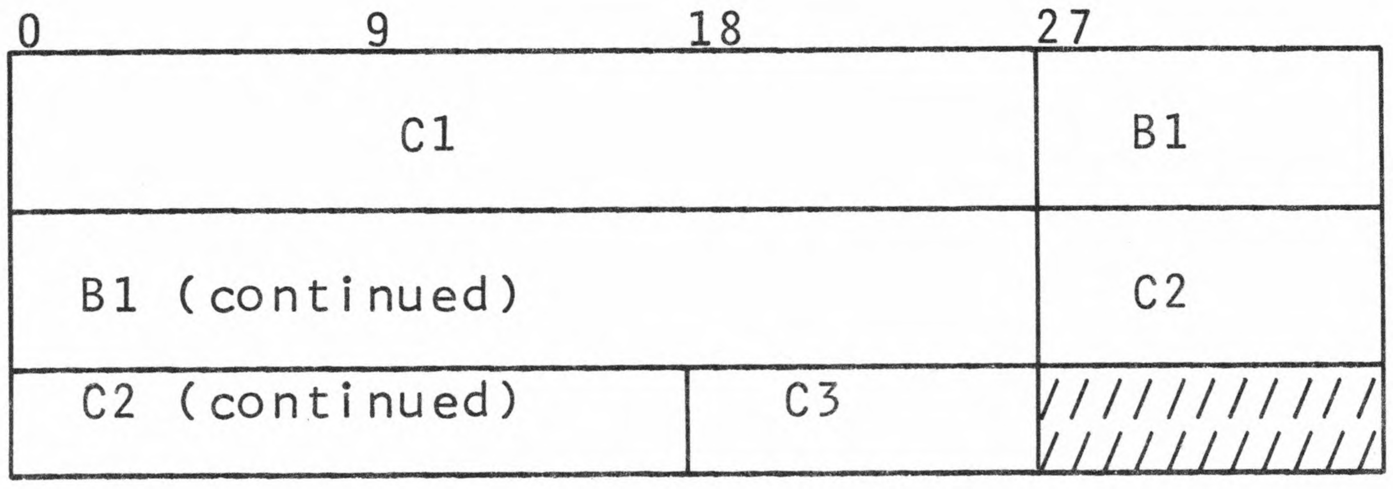
```
DO I = 1 TO 10;
   Z(I) = X(I) + Y(I);
   DO J = 1 TO 100;
      C(I,J) = A(I,J) + B(I,J);
         END;
   END;
```

Structure Layout

If a CHARACTER or BIT string variable within a structure is frequently accessed, it should not share a word with another variable. For example, consider the following structure of string variables:

```
DCL 01 BC ALIGNED,
       02 C1 CHAR(3) UNAL,
       02 B1 BIT(36) UNAL,
       02 C2 CHAR(3) UNAL,
       02 C3 CHAR(1) UNAL;
```

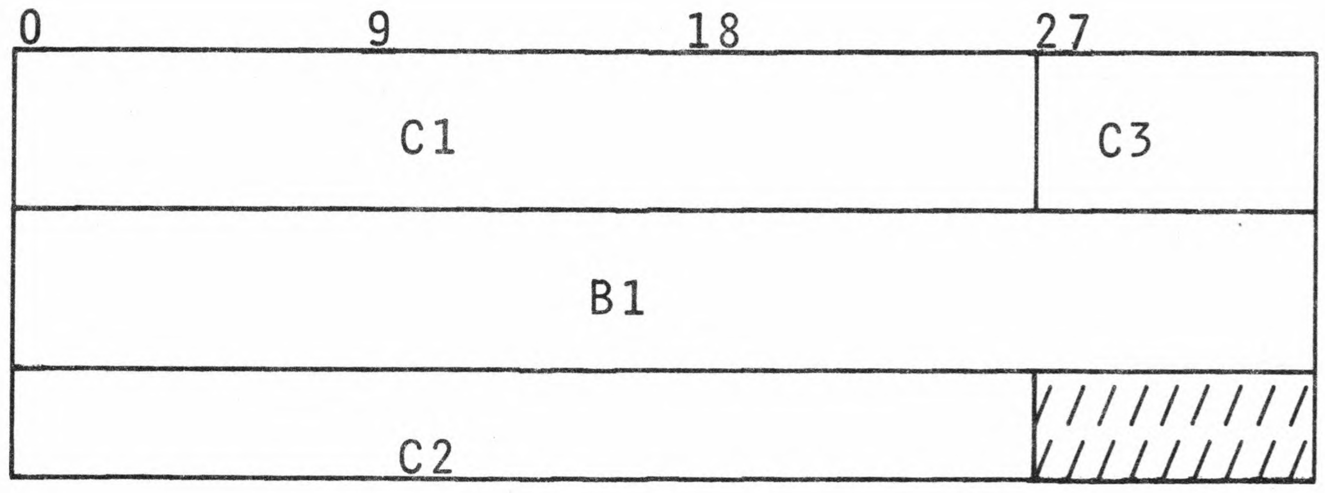
The structure BC is represented in storage in the following way:



The variables share words with one another. Assuming that the variables B1 and C2 are frequently accessed, rewriting the structure in the following way improves the efficiency of the program:

```
DCL 01 BC ALIGNED,
       02 C1 CHAR(3) UNAL,
       02 C3 CHAR(1) UNAL,
       02 B1 BIT(36) UNAL,
       02 C2 CHAR(3) UNAL;
```

The structure BC is now represented in storage in the following way:



Access of the variables B1 and C2 in this representation is more efficient than in the previous representation.

The rules for laying out aggregates in memory are given earlier, in the section on "Internal Representation of PL/I Data".

<u>Variable Extents</u>

The use of strings whose maximum length is not known at compile time or arrays whose bounds are not known should be avoided whenever possible. If such a variable must be used within a structure, it should be the last member of the structure, as follows:

```
DCL 01 C ALIGNED,
       02 C1 CHAR(8),
       02 C2 CHAR(4),
       02 C3 CHAR(4),
       02 C4 CHAR(N);
```

Static Global Variables

A variable that is declared in an outer block and frequently accessed in the inner blocks should be declared STATIC. For example, in the program fragment:

```
E1: BEGIN;
        DCL (X,Y) FIXED;
        Y = X;
E2:     BEGIN;
            .
            .
            .
            X = X + 1;
            .
            .
            .
E3:         BEGIN;
                .
                .
                .
                Z = X;
                .
                .
                .
                END;
            .
            .
            .
            END;
        .
        .
        .
        END;
```

If X is declared as a STATIC variable, the amount of object code for the above program is reduced. The declaration of X and Y should be rewritten as:

```
DCL X FIXED STATIC;
DCL Y FIXED;
```

Global and Parameter Variable References

When an automatic variable declared in a calling block is used frequently in the called block, the variable should be assigned to an AUTOMATIC temporary declared in the called block.

Parameters passed by a CALL statement or a function reference, if frequently accessed, should be assigned to a temporary AUTOMATIC variable within the called procedure. The use of temporary storage for parameters is especially effective if the parameters are declared with the UNALIGNED attribute.

Constant Arguments

The use of named constants as arguments of a CALL statement is more efficient than the use of literal constants. Consider the following program fragment:

```
DCL SUB2 ENTRY(FIXED,FIXED);
DCL CONST2 FIXED INT STATIC INIT(2),
    CONST3 FIXED INT STATIC INIT(3);


CALL SUB2(CONST2,CONST3);


CALL SUB2(2,3);
```

The statement with the named constant arguments CONST2 and CONST3 is more efficient than the statement with the literal constant arguments.

Initialization

If a variable is initialized, it should be declared as STATIC if possible. Code for the initialization of AUTOMATIC variables must be executed on each entry to the block or procedure.

Labels

Avoid the use of unnecessary labels in a program. Labels are sometimes used to indicate a program note rather than a transfer point. The compiler can perform better optimization on a series of statements if the statements are not broken up by labels.

Concatenation

Unnecessary use of concatenation should be avoided because concatenation operations are time-consuming.

Stream Input-Output

In stream input-output, use one statement with a long data list rather than several input-output statements. Each input-output statement requires linkage and, therefore, has an associated overhead.

Temporary Work Files

The use of ASCII is more efficient than the use of BCD in temporary stream work files.

Edit-Directed Input-Output

For stream input-output, edit-directed input-output is more efficient than either list- or data-directed input-output if the GET or PUT statement specifies more than one item.

Stream Data List

In stream input-output, a single long item rather than a sequence of short items, should be used in the data list whenever possible. Consider, for example, the following program fragment:

```
DCL 01 C ALIGNED,
       02 C1 CHAR(2)  UNAL,
       02 C2 CHAR(6)  UNAL,
       02 C3 CHAR(16) UNAL,
       02 C4 CHAR(20) UNAL,
       02 C5 CHAR(36) UNAL;
DCL STR CHAR(80) ALIGNED DEF(C);


GET LIST(STR);


GET LIST(C1,C2,C3,C4,C5);
```

Buffers

For INDEXED or REGIONAL file organization, the number of buffers allocated determines the actual amount of data transmitted to and from external files. The allocation of sufficient buffer space minimizes the amount of time spent transferring data.

RULES FOR IMPROVING STORAGE EFFICIENCY

The rules in this section are useful for improving the storage efficiency of a PL/I program.

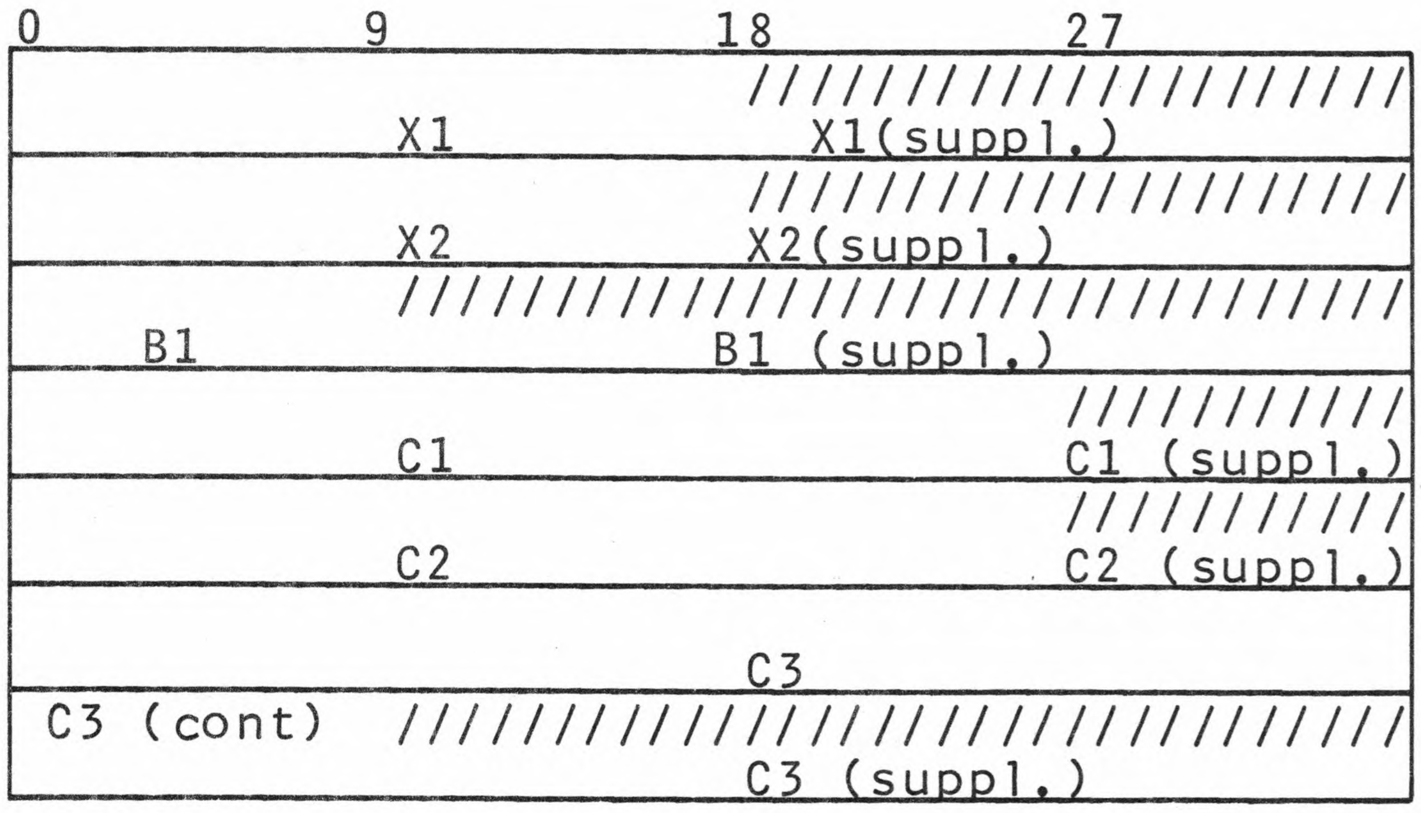
Alignment

In order to minimize the amount of storage used (at the expense of access time), use the UNALIGNED attribute for variables within a structure.

Consider the case in which a large number of tables are allocated in storage at execution time. The table has the following declaration:

```
DCL 01 TABLE ALIGNED BASED,
       02 X1 FIXED,
       02 X2 FIXED,
       02 B1 BIT(9),
       02 C1 CHAR(3),
       02 C2 CHAR(3),
       02 C3 CHAR(5);
```

The storage layout for the above structure occupies seven words, as follows:



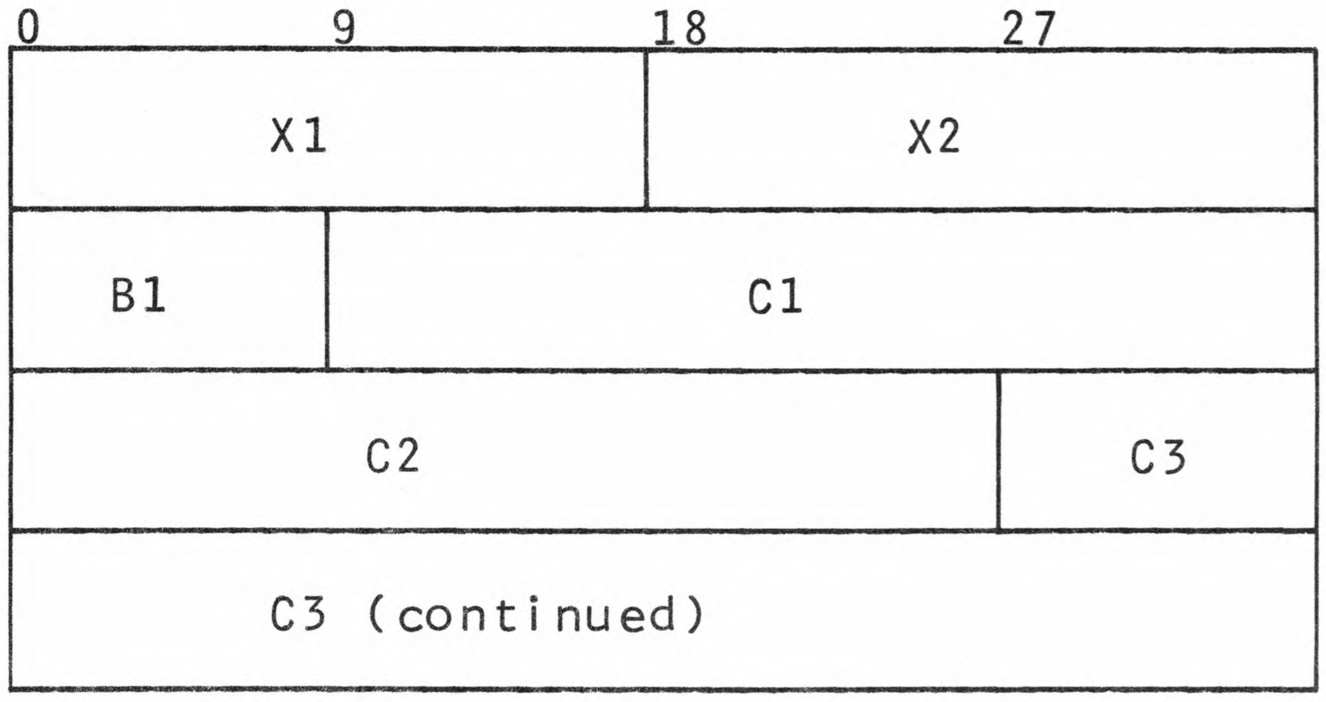
Each execution of the statement

```
ALLOCATE TABLE SET(P);
```

allocates seven words and sets the pointer variable P to point to the starting address. However, if the variables of the structure TABLE are declared to be UNALIGNED, the variables are stored to minimize storage:

```
DCL 01 TABLE ALIGNED BASED,
       02 X1 FIXED UNAL,
       02 X2 FIXED UNAL,
       02 B1 BIT(9) UNAL,
       02 C1 CHAR(3) UNAL,
       02 C2 CHAR(3) UNAL,
       02 C3 CHAR(5) UNAL;
```

The storage layout for the above declaration occupies only four words, as follows:



The second representation of the structure TABLE saves three words per structure. If many TABLEs are to be allocated, then the saving is substantial.

Detailed rules for the storage layout of variables are given earlier in the section on the "Internal Representation of PL/I Data".

Static Variables

Variables declared with the STATIC attribute are allocated when the object program is loaded and remain allocated throughout the activity. The use of the STATIC attribute, therefore, should be avoided whenever possible if storage efficiency is the principal consideration.

File Organization

If a program uses only one or two of the three possible types of file organization, then the specification of the file organization in the ENVIRONMENT option of the file declaration is more efficient than the specification of the organization on control cards at execution time.

External Variables

The cost of binding and allocating each external variable is high. This cost can be reduced by gathering several external variables together into a structure. Consider the following declarations:

```
DCL (X1,X2,X3) FIXED EXT STATIC;
DCL (P1,P2,P3,P4) PTR EXT STATIC;
```

A more efficient representation of the above is:

```
DCL 01 LINK EXT STATIC,
       02 X1 FIXED,
       02 X2 FIXED,
       02 X3 FIXED,
       02 P1 PTR,
       02 P2 PTR,
       02 P3 PTR,
       02 P4 PTR;
```

This also reduces the number of separate labeled common regions because a separate labeled common region is created for each external static declaration statement. A maximum of 63 labeled common regions is permitted per external procedure.

Data-Directed Input-Output

The use of the DATA option in stream input-output without an explicit list of variables requires the entire symbol table to be kept in storage during the execution of the program; therefore, it should never be used without a list. Even with an explicit list of variables, the cost is considerable. The use of the DATA option should be confined to debugging.

Input-Output Interfacing

Whenever possible, input-output statements should be confined to a single block in a program. The system allocates storage within each block containing input-output statements for an input-output interfacing facility. If input-output statements appear in more than one block, storage is allocated in each block for the interfacing facility, and the resulting amount of storage for the program is increased.

Work Regions for Files

When an INDEXED or REGIONAL file is used, the proper size for the work region should be calculated, using the formulas given earlier in the sections on "INDEXED Organization" and "REGIONAL Organization" respectively. The specification of the proper size for the work region on the $ USE control card avoids the allocation of unnecessary space.

SECTION XV

COMMON PROGRAMMING ERRORS

This section contains remarks on some of the most common programming errors made in the use of PL/I. Some of the errors described here are detected by the compiler. However, some of the errors are undetectable at compile time, and their occurrence during the execution of the program produces invalid results or interruption of the flow of control of the program. Often, this type of error arises from a misunderstanding of the rules of PL/I and is, therefore, difficult to resolve.

The common mistakes are listed according to the following classifications:

Program constructs
Program structure
Program control
Initialization
Evaluation
Conversion
Procedure calls
Input-output

The errors in each of the above classifications are described and, where necessary, illustrated by an example. The classification given here is intended to aid the reader in locating a topic of interest; however, like most classifications, the above one is somewhat arbitrary. Moreover, the list of common errors given here is not to be considered, in any sense, complete.

Discussions of programming style appear throughout the PL/I Reference Manual, usually under headings of the form "Guidelines for ... ". Some of the discussions point out features of PL/I that are especially susceptible to programming errors.

PROGRAM CONSTRUCTS

Some reminders related to the basic constructs of a PL/I program are given in the following paragraphs.

Special Characters

The representations of some characters in the PL/I character set depend upon the code used. The representations for these characters in ASCII, BCD, and EBCDIC are given in the following list:

	Representation		
Character	ASCII	BCD	EBCDIC
OR	!	!	|
AND	&	&	&
NOT	∧	↑	¬
break	_	←	_
quote	' or "	' or "	'
concatenation	!!	!!	||

Reserved Character Combination

A program or data card with the character '$' in Column 1 and the blank character in Column 2 can be mistaken for a GCOS control card. Therefore, this sequence of characters in the first two columns of a card should be avoided.

Confusion Between Break and Minus

The statement

```
MACHINE = H-6000;
```

is interpreted as the assignment to the variable MACHINE of the difference between the variable H and the constant 6000. If H-6000 is to be interpreted as a variable name containing a break character, the statement must be written as follows:

```
MACHINE = H←6000;   (in BCD)
MACHINE = H_6000;   (in ASCII and EBCDIC)
```

Confusion Between Assignment and Comparison Operators

The character '=' is used both for assignment and for comparison. In the following statement, the first character '=' is used for assignment and the second, then, for comparison:

```
A = B = C;
```

The above statement is equivalent to the statement:

```
IF B = C THEN A = '1'B ELSE A = '0'B;
```

Multiple assignment is accomplished by commas separating the identifiers to be assigned. If the above statement is to be a multiple assignment, it would be written:

```
          A,B = C;
```

This has the effect of assigning the value in C to both A and B.

Picture Characters

Alphabetic PICTURE characters must be given in upper case even when the program is input directly in ASCII from a terminal. For example, the following declaration is incorrect:

```
          DCL X PIC 'aaaa';
```

The correct form of the above declaration requires the PICTURE characters to be in upper case, as follows:

```
          DCL X PIC 'AAAA';
```

Decimal Point in a Pictured Character String

The PICTURE character 'V' indicates a scale factor and does not. occupy storage. On the other hand, the PICTURE character '.' is an editing character and does not indicate the scale factor. Consider, first, a program fragment that uses the 'V' character in the declaration of CHARGE:

```
               DCL CHARGE PICTURE "S999V9";
               CHARGE = 123.4;
               PUT LIST(CHARGE);
```

The execution of the PUT statement produces the following output:

```
          +1234
```

Consider, next, a program fragment that uses the '.' character in the declaration of CHARGE:

```
               DCL CHARGE PICTURE "S999.9";
               CHARGE = 123.4;
               PUT LIST(CHARGE);
```

The execution of the PUT statement produces the following output:

```
          +012.3
```

To output the true value of CHARGE, the characters 'V' and '.' must be adjacent in the declaration. For example:

```
        DCL CHARGE PICTURE "S999V.9";
        CHARGE = 123.4;
        PUT LIST(CHARGE);
```

The execution of the PUT statement, in this case, produces the correct result, namely:

```
        +123.4
```

Restrictions on Identifiers

The characters '#' and '@' cannot be used in an identifier. The length of an external name is limited to six characters. The length of a file name is limited to five characters.

Conflict Between Built-In Function and Procedure Names

If a built-in function is used and not declared, a conflict can occur, as follows:

```
    TRUNC: PROC(Z) RETURNS(FIXED);
                 .
                 .
                 .
                 Y = TRUNC(X);
                 .
                 .
                 .
                 END;
```

Since the built-in funtion TRUNC is not declared, the procedure TRUNC in this example is assumed to be recursive.

PROGRAM STRUCTURE

The unintentional omission of a delimiter is a common error in program structure. The effect of a missing comma, semicolon, or parenthesis is well known. A missing or misplaced END statement affects the entire program meaning. The effect of the omission of comment delimiters, quotation marks, and ELSE clauses are described in the following paragraphs and a recommendation about the use of the END statement for the multiple closure of blocks is made.

Unmatched Comment Delimiters

Unmatched comment delimiters can produce an unexpected interpretation of the program by the compiler. Consider the following example:

```
/* COMMENT:   CALCULATE SPEED
   X = Y**Z**2;
/* COMMENT:   CALCULATE TIME  */
```

The unintentional omission of the closing comment delimiter '*/' on the first line of this fragment results in all three lines being taken as comment. The assignment on the second line, therefore, is never performed.

Nested comments are not allowed. The closing comment delimiter of the nested comment prematurely terminates the enclosing comment:

```
/* COMMENT: IN THIS CASE
   X   /* THE SPEED  */
   IS REPLACED BY AN ESTIMATED VALUE */
```

The comment is terminated at the end of the second line of the above fragment. The third line is not considered to be a comment and the PL/I compiler attempts to interpret that line as a PL/I statement.

Quotes

In order to preserve the pairing of quotation mark delimiters for character strings, any quote within the string must be replaced by a double quote. Consider the following text:

```
HE SAID, "THE CAR WON'T GO".
```

When this text is represented as a character string constant, the internal quotes are replaced by double quotes as follows:

```
"HE SAID, ""THE CAR WON''T GO""."
```

Thus, the pairing of the quote delimiters is preserved.

Matching ELSE Clauses

The compiler associates an ELSE clause with the closest previous unmatched IF statement. To get the correct sequence of control, it is sometimes necessary to include a null ELSE clause. For example, to assign the value 1 to X if both B1 and B2 are true and the value 2 to X if B1 is false, a null ELSE clause is required as follows:

```
IF B1
     THEN IF B2
          THEN X = 1;
          ELSE;
     ELSE X = 2;
```

If the null ELSE clause is omitted, then the ELSE clause is associated with the closest previous IF, as follows:

```
IF B1
     THEN IF B2
          THEN X = 1;
          ELSE X = 2;
```

When the ELSE clause is associated in this way, the value 1 is assigned to X if B1 and B2 are both true and the value 2 to X if B1 is true and B2 is false.

Multiple Closure of Blocks

The use of an END statement with a label to close a series of nested PROCEDURE blocks, BEGIN blocks, or DO-groups can introduce an obscure error. Consider the following program fragment:

```
P1:  PROC;
       .
       .
       .

P2:    PROC;
          .
          .
          .
          DO I = 1 TO N;
               A(I) = B(I+1);
               C(I) = C(I)**2;
          X = X + 1;
          .
          .
          .
       END P1;
```

The END statement with the label P1 is intended to close the PROCEDURE block P1 and the internal procedure P2. However, the END statement for the DO-group was inadvertently omitted, and, therefore, the END statement closes the DO-group as well as the above mentioned procedure blocks.

If blocks are closed explicitly by END statements without a label, any missing END statement is detected by the compiler. The use of END statements for multiple closure of blocks tends to obscure the structure of the program and is, therefore, not recommended.

PROGRAM CONTROL

Some critical features of program control are described in the following paragraphs.

OPTIONS(MAIN) Attribute

Only one procedure in a program can have the OPTIONS(MAIN) attribute. At execution time, control is passed to the procedure declared with that attribute.

Transfer of Control

A GOTO statement that is outside a given PROCEDURE block, BEGIN block, or iterative DO-group cannot transfer to a label that is inside the block or group. The following fragment contains such a transfer and is, therefore, invalid.

```
P1:  PROC;
        .
        .
        .
        GOTO A;
        .
        .
        .
P2:     PROC;
             .
             .
             .
A:           .
             .
             .
             END;
        END;
```

A PROCEDURE block is executed only when it is invoked by a CALL statement or a function reference. A BEGIN block is executed when control reaches the BEGIN statement, either from the preceding statement or from a transfer to the label of the BEGIN statement. Consider the following program fragment:

```
P1:   PROC;
          .
          .
          .
          GOTO A;
          GOTO B;
          .
          .
          .
A:        BEGIN;
               .
               .
               .
               END;
B:        PROC;
               .
               .
               .
               END;
          END;
```

The first transfer statement is correct and as a result of its execution control is transferred to the block labeled A. The second transfer statement is incorrect. To execute the procedure B, a CALL statement must be used. If control is not explicitly transferred to the BEGIN block in the above example, it is executed when control passes to it from the preceding statement. The PROCEDURE block, however, is not executed unless it is explicitly invoked.

Changing the Index within a DO-Group

If the index of an iterative DO-group is changed within the DO-group, the index may never exceed the limit and the program may loop. Consider the following program fragment:

```
DO I = 1 TO 20;
     .
     .
     .
     PUT LIST((X(I) DO I = 1 TO 10));
     .
     .
     .
     END;
```

The index I in this example is reset on each execution of the group and, therefore, never exceeds the limit.

LABEL and ENTRY Variables

Misuse of LABEL or ENTRY variables can cause errors that are difficult to trace. Consider the following program fragment, in which the use of ENTRY variables causes the mechanism for block activation to be disrupted:

```
P1:    PROC OPTIONS(MAIN);
          DCL EV ENTRY VARIABLE;
          .
          .
          .
          CALL E2;
          .
          .
          .
OUT1:     CALL EV;
          .
          .
          .
E2:       PROC;
              .
              .
              .
              EV = E3;
              .
              .
              .
              GOTO OUT1;
              .
              .
              .
E3:           PROC;
                 .
                 .
                 .
                 END;
              .
              .
              .
              END;
          .
          .
          .
          END;
```

The statement

    CALL EV;

attempts to call the procedure E3, which is nested within the procedure E2. However, at the time of the call, E2 is no longer active and therefore, the call is not valid.

INITIALIZATION

Misunderstanding of the effect of initialization and the time at which initialization activities occur often gives rise to programming errors. Some remarks on this type of error are given in the following paragraphs.

Initialization of Variables

If a variable is accessed before a value has been assigned to the variable, the program is in error and its continued execution is undefined.

No variable should be expected to be initialized unless it is declared with the INITIAL attribute. If an AUTOMATIC variable is declared with an INITIAL attribute, then the variable is initialized to that value upon each activation of the block in which it is declared. If an AUTOMATIC variable is declared without an INITIAL attribute, then the value of the variable is undefined upon each activation of the block.

An attempt to write out, using STREAM-oriented data transmission, variables that have not been initialized can cause a CONVERSION error to occur.

Allocation of Variables

An AUTOMATIC variable in a block is allocated when the block is activated. Consider the following program fragment:

```
P:    PROC;
         DCL N FIXED;
         .
         .
         .
         N = 10;
P2:      BEGIN;
               DCL C1 CHAR(N);
               DCL C2 CHAR(N) BASED;
               N = 20;
               ALLOCATE C2 SET(P);
               .
               .
               .
               END;
      .
      .
      .
      END;
```

The AUTOMATIC variable C1 is allocated when the block is activated. Since the value of N is 10 when the block is activated, ten characters are allocated for C1 and remain the allocation for C1 throughout the block's execution. The variable C2 is allocated by the ALLOCATE statement. Since the value of N is 20 when the ALLOCATE statement is executed, twenty characters are allocated for C2.

Evaluation of Increments and Limits for DO-Groups

Increments and limits for DO-groups are computed upon entry to the DO-group. Consider the following program fragment:

```
     P:   PROC;
             .
             .
             .
             J = 10;
             DO I = 1 TO J;
                  .
                  .
                  .
                  J = 20;
                  .
                  .
                  .
                  END;
             END;
```

The limit of the DO-group is evaluated upon entry and is, therefore, 10. Although the value of J is changed within the DO-group, the value of the limit is unchanged and consequently, the DO-group is executed ten times.

External Names

The declarations of an external name must be identical in all external procedures which become part of a single execution unit. Consider, for example, the declaration of the external variable, M, in two external procedures:

```
     P1:  PROC;
             DCL M FIXED EXT INIT(1);
             .
             .
             .
             END;

     P2:  PROC;
             DCL M FIXED EXT;
             .
             .
             .
             END;
```

The two declarations of M differ: in one case M is initialized and in the other, it is not. The above procedures, therefore, are incorrect.

Extent Expressions for BASED Variables

The extents of BASED variables must be known at the time the variable is allocated. Consider the following program fragment:

```
P1:  PROC;
        DCL 01 BV ALIGNED BASED(P),
                  02 N FIXED,
                  02 CH CHAR(N);
        .
        .
        .
        ALLOCATE BV;
        .
        .
        .
        END;
```

The length of the character string variable CH is given by N, a member of the same structure. Therefore, when the ALLOCATE statement is executed, the value of N and the length of the character string CH are undefined. To obtain the correct result, the REFER option should be used, as follows:

```
DCL 01 BV ALIGNED BASED(P),
       02 N FIXED,
       02 CH CHAR(M REFER(BV.N));
```

The value of M determines the maximum length of the character string and is assigned to BV.N at the same time that BV is allocated. Subsequent references to the BASED variable CH make use of the value of BV.N.

Replication Factors in INITIAL Attributes

In an INITIAL attribute, a parenthesized expression can be used to treat a single value as a sequence of values. Consider, for example, the declaration:

```
DCL X(3) FIXED INIT((3)1);
```

This declaration initializes each of the three elements of X to the value one.

However, a problem arises when the initial value is a string constant. Consider the declaration:

```
DCL Y(3) CHAR(20) INIT((3)'X');
```

The parenthesized expression in this declaration is a factor that denotes the repetition of the sequence in the string constant the specified number of times to derive the complete constant. The above declaration is equivalent to:

```
DCL Y(3) CHAR(20) INIT('XXX');
```

Since this declaration provides only one initial value when three initial values are required, the declaration is invalid. To initialize the three values to a single X, the following declaration must be used:

```
        DCL Y(3) CHAR(20) INIT((3)(1)'X');
```

In this statement, the parenthesized expression '(1)' is the factor treated as part of the string constant, and the parenthesized expression '(3)' is treated as a replication factor for the initial values of the three-element array.

EVALUATION

The following paragraphs list errors that arise from misunderstanding of the order of evaluation or the behavior of the SUBSTR built-in function.

Multiple Assignments

Misunderstanding of the order in which multiple assignments are executed sometimes creates a programming error that is difficult to trace. Consider the following program fragment:

```
    P1:   PROC;
            DCL (A(5),I) FIXED;
            .
            .
            .
            A = 0;
            I = 1;
            A(I),I,A(I) = I + 1;
            .
            .
            .
            END;
```

After the execution of the multiple assignment statement, the array elements have the following values:

```
        A(1)   2
        A(2)   2
        A(3)   0
        A(4)   0
        A(5)   0
```

The multiple assignment statement is equivalent to the following sequence of statements:

```
        TEMP = I + 1;
        A(I) = TEMP;
        I    = TEMP;
        A(I) = TEMP;
```

Evaluation Order

Misunderstanding of the order in which expressions are evaluated can lead to errors that are difficult to find. The following list gives some common unparenthesized expressions. The order of evaluation for these expressions is made explicit by the use of parentheses.

Expression	Evaluation Order
A = B ! C	(A = B) ! C
B ! C & D	B ! (C & D)
A > B ! C	(A > B) ! C
A > B > C	(A > B) > C
A**B**C	A**(B**C)

Parentheses can be used to change the order of evaluation when required.

SUBSTR Built-In Function Arguments

The arguments of the SUBSTR built-in function that specify the starting position and the length must be valid for the specified string. Consider the following example:

```
P:    PROC;
         DCL C1 CHAR(12);
         DCL C2(10) CHAR(4);
         .
         .
         .
         DO I = 1 TO 10
               C2(I) = SUBSTR(C1,I,4);
               END;
         END;
```

The assignment statement in the above example is valid for the values of I from one to nine. However, when I has the value ten, the SUBSTR function attempts to access the thirteenth character of a twelve-character string and the SUBSCRIPTRANGE condition is raised if the condition is enabled. If the SUBSCRIPTRANGE condition is not enabled, the program is in error and its continued execution undefined.

SUBSTR Function and Varying Strings

An assignment to a varying string by the SUBSTR pseudo-variable does not alter the control word that holds the current length of the string. Consider the following example:

```
DCL C1 CHAR(6) VARYING;
.
.
.
C1 = "A";
SUBSTR(C1,2,3) = "BCD";
PUT LIST(C1);
```

The length of the varying string C1 is determined to be one by the first assignment statement. Since the SUBSTR pseudo-variable does not alter that length in the second statement, the PUT statement outputs the single character 'A'.

Only another assignment into or concatenation onto a varying string will modify the current size of the string.

CONVERSION

Some of the most difficult errors in PL/I programs are related to the effect of conversion. Conversions should be avoided, whenever possible. The resulting program is clearer and more efficient. Some conversion problems are described in the following paragraphs.

Fixed-Point Division

The precision of a constant affects the precision of the intermediate results. Consider the following program fragment:

```
DCL X FIXED DEC(3,1);
X = 10 + 1/3;
```

The precision of the intermediate result is (59,58). In PL/I, this precision causes the FIXEDOVERFLOW condition to occur. For general fixed-point arithmetic expressions, the DIVIDE built-in function, which permits the programmer to give the precision of the result, should be used. For constant values, a single constant with a decimal point should be used. Thus, the above assignment statement should be written as:

```
X = 10.333333
```

Loss of Precision in Conversion

According to the rules of PL/I, conversion occurs whenever the data types of the operands of an expression differ. In some cases, the conversion results in the loss of the value of the result of the operation through truncation. Consider the following program fragment:

```
DCL X FIXED(4) INIT(1);
DCL (B1, B2) BIT(4);
.
.
.
B1 = X;
B2 = X + 1;
.
.
.
```

The execution of the first assignment statement assigns the value '0001' to B1. The execution of the second assignment statement involves addition. The precision of the decimal constant, 1, converted to binary is (5,0). The precision of the result of the addition operation, then, is (6,0). The value of the addition operation '000010' is truncated to four bits for assignment to B2. Therefore, the value '0000' is assigned to B2 as a result of the execution of the second assignment statement.

To obtain the correct result the built-in functions FIXED or BIT can be used, as follows:

```
B2 = FIXED(X+1,4);
```

or

```
B2 = BIT(FIXED(X+1,3),4);
```

Fixed-Point Arithmetic to Character Conversion

Assignment of a fixed-point arithmetic value to a character string requires three more characters in the string than there are digit positions in the value. Consider the following example:

```
DCL C CHAR(6) STATIC INIT(123456);
```

The initial value of C, as specified in the above declaration, is:

```
'bbb123'
```

where b indicates the blank character.

To obtain the desired result, the initial value should be given as a character string constant, as follows:

```
DCL C CHAR(6) STATIC INIT("123456");
```

PROCEDURE CALLS

Some common programming errors made in the invocation of procedures and function references are described in the following paragraphs.

By-Value Arguments

If the storage type of an argument in a CALL statement or function reference is not identical to the storage type of the parameter, the argument is passed by-value. The value of a by-value argument cannot be changed by any action of the procedures. Consider the following program fragment:

```
E1:    PROC;
          DCL X BIT(3);
          DCL E2 EXT ENTRY(BIT(3) ALIGNED);
          X = "001"B;
          CALL E2(X);
          IF X = "101"B THEN GOTO NEXT;
          .
          .
          .
NEXT:     END;
E2:    PROC(B);
          DCL B BIT(3) ALIGNED;
          .
          .
          .
          B = "101"B;
          RETURN;
          END;
```

Since no alignment is specified for the variable X in its declaration, it acquires the default alignment UNALIGNED. Therefore, the storage type of the argument X is not identical to the storage type of the parameter B. The argument is passed by-value and after the execution of the procedure E2, the value of X is unchanged. The test on X following the procedure call, therefore, fails.

Parenthesized Arguments

If an argument is enclosed in parentheses, it is passed by-value. Since the generation of storage associated with the parameter is not the generation occupied by the original argument, any assignment to the parameter by the procedure has no effect upon the value of the argument.

Consider the following program fragment:

```
P1:  PROC;
        DCL P PTR ALIGNED;
        .
        .
        .
        CALL SUB((P));
        .
        .
        .
SUB:    PROC(Q);
           DCL (Q,R) PTR ALIGNED;
           .
           .
           .
           Q = R;
           END;
        .
        .
        .
        END;
```

The execution of the procedure SUB does not affect the value of the pointer variable P.

Function References without Arguments

A function reference without an argument list must be followed by a pair of parentheses, indicating an empty list. In this way a function reference is distinguished from an entry constant. Consider the following program fragment:

```
DCL FUNC ENTRY() RETURNS(CHAR(10));
DCL V ENTRY;
DCL C CHAR(10);
.
.
.
V = FUNC;
C = FUNC();
```

The first assignment statement assigns the entry constant FUNC to the entry variable V. The second assignment assigns the result of the function FUNC to the CHARACTER string variable C.

Multiple Entry Points

When a procedure has more than one entry point and the different entry points have their own parameter lists, any statement within the procedure that refers to parameters from different lists is in error. For example, consider the following program fragment:

```
     G1:  PROC(A,B);
               .
               .
               .
     G2:  ENTRY(X,Y);
               .
               .
               .
               C = A + X;
               .
               .
               .
               END;
```

The assignment statement in the above example is in error because it refers to a parameter, A, from one list and a parameter, X, from another list.

Parameter Extents

The extent of a parameter can be declared with either a constant or an asterisk. The following declaration of a parameter is wrong because the parameter extent is declared to be a variable:

```
     P1:  PROC(A,N);
               DCL A(N) FIXED DEC(6);
               .
               .
               .
               END;
```

INPUT-OUTPUT

Common errors related to input-output are described in the following paragraphs.

Input-Output Lists

An iterated input-output list must be parenthesized. The following GET statement is syntactically incorrect.

```
          GET LIST(A(I), I DO I = 1 TO N);
```

The above statement should be written as:

```
          GET LIST((A(I), I DO I = 1 TO N));
```

Control Format Items

A control format item is executed only if it precedes a data format item that is paired to a data item. Consider the statement:

```
          PUT EDIT(A,B)
                  (F(4),X(5),F(5),X(3));
```

In this statement, the control format item, X(3), is not executed.

Control Options

Control options are always executed before data transmission regardless of their position with respect to the data specifications. For example, the statements:

```
          PUT SKIP EDIT(X,Y)
                  (F(3),X(2),F(5));
```

and

```
          PUT EDIT(X,Y)
             (F(3),X(2),F(5)) SKIP;
```

are equivalent. Both statements skip a line before printing X and Y in the EDIT-directed format.

Input Strings

A CHARACTER or BIT string format item must include the string size on input. The size specification is not required for output.

Mixed Transmission

If a file is read using both list- and edit-directed data transmission, the format of the file must be taken into account. Consider, for example, the following program fragment, which contains two types of data transmission:

```
DCL X FIXED DEC(6);
DCL Z CHAR(4);
.
.
.
GET LIST(X);
GET EDIT(Z)
        (A(4));
.
.
.
```

When this program reads the input stream:

```
123bb,bABC
```

the following results are obtained for X and Z:

```
X    123
Z    'b,bA'
```

That is, after execution of the first list-directed GET statement, the next data item will begin with the first character following the blank or comma that separates it from the previous data item.

Page and Line Size

The specifications PAGESIZE and LINESIZE for a file are given as options on the OPEN statement for the file. These specifications are not part of the file declaration.

BCD Devices

In RECORD-oriented input-output, code conversion is not performed. Since the internal code for PL/I is ASCII, RECORD-oriented input-output cannot be used to transmit data to peripheral devices that accept only BCD. Therefore, STREAM-oriented input-output must be used to transmit data from the card reader or to the line printer.

Control Cards for INDEXED and REGIONAL Files

If a file with INDEXED or REGIONAL organization is used in a program, the work region and file parameters must be given at execution time on control cards. The required control cards are described in the sections of this manual dealing with file organization.

SECTION XVI

SOLUTION OF A PROBLEM IN PL/I

The variety of language features available in PL/I gives the user great flexibility in the solution of problems. To solve a given problem, a number of different approaches can be taken.

This section defines a problem and then illustrates two possible programs for the solution of the problem. The first solution is a hastily-written program for the programmer's own use. The second solution is a program developed for use in a production environment.

The problem is simple and the programs are short. Aside from this unreal simplicity, the problem represents a realistic application of PL/I. The programs illustrate the features of the language and the freedom the programmer has in the solution of a problem. In addition to the solution programs given here, there are other equally valid solutions to the problem. The suitability of a program depends upon the constraints under which it is written.

DEFINITION OF THE PROBLEM

The problem to be solved is defined as follows:

> Input a value, calculate the exact value of its factorial,
> and print the value with the calculated factorial.

The factorial is a mathematical function that is defined by the following formula:

factorial(n) = n(n-1)(n-2)...1

where n is an integer greater than zero, and

factorial(0) = 1

The calculation of factorials is a simple problem with some interesting properties:

- The factorial is defined only for non-negative integers.
- The factorial increases rapidly.

The two programs given here calculate the factorial in approximately the same way. Both programs use binary arithmetic to obtain an efficient solution and fixed-point arithmetic to obtain maximum binary capacity. The programs, however, are very different in their handling of input-output.

FIRST SOLUTION

The first solution of the defined problem is a program written for the programmer's own use. The program is written using list-directed input to obtain the values for which the factorial is to be calculated. Since the only user of the program is the programmer himself and since the program is to be run only a few times, no special provisions are made to detect and handle illegal input. If a condition occurs as a result of an input value, the default ON unit is invoked and the execution of the program terminated. For convenience, the programmer makes use of the default ON unit for the ENDFILE condition to terminate the run when the input values are exhausted.

Deck Setup

The deck setup for the first solution is given in Figure 16-1. Since only the output values are of interest to the programmer, the optional listings are omitted and warning messages suppressed.

```
1      8      16
_______________________________________________

$      SNUMB   JOB16
$      IDENT
$      USERID
$      OPTION  PL1,NOMAP
$      PL1     SEVERITY2

P1:    PROC OPTIONS(MAIN);
         DCL I FIXED;
         DCL (SYSIN,SYSPRINT) FILE;
         PUT PAGE LIST("EXACT VALUES OF FACTORIALS");
         PUT SKIP LIST("I","FACTORIAL(I)");
LOOP:    GET LIST(I);
         PUT SKIP LIST(I,FACT(I));
         GOTO LOOP;
FACT:  PROC(ARG) RETURNS(FIXED(71));
         DCL ARG FIXED;
         DCL RES FIXED(71);
         DCL I FIXED;
         IF ARG = 0 THEN RETURN(1);
         RES = 1;
         DO I = 1 TO ARG;
              RES = RES*I;
              END;
         RETURN(RES);
         END;
       END;

$      EXECUTE
$      LIMITS  2,30K,-2K
$      DATA    I*
6 0 20 5 9 21 22
$      ENDJOB
***EOF
```

Figure 16-1. Deck Setup for First Solution

Output Listing

The complete output listing for this run is given in Figure 16-2. Because the programmer made use of the default ON unit for the ENDFILE condition, an error trace-back and register list are given at the conclusion of the execution of the program. Following this information, the information output on SYSPRINT is listed.

```
$$     JOB16 ENTERED 7.1*D  AT 11.358 FROM SYSTEM-0 CD RDR  0-24-01

0001  $       SNUMB    JOB16
0002  $       IDENT
0003  $$      USERID
0004  $       OPTION   PL1,NOMAP
0005 A$       PL1      SEVERITY2
0006 A$       EXECUTE
0007  $       LIMITS   2,30K,-2K
0008  $       DATA     I*
0009  $       ENDJOB
  TOTAL CARD COUNT THIS JOB = 000030

* ACTY-01  $CARD #0005  PL1       09/15/75    SW=210200000000
  * NORMAL TERMINATION      AT 020437 I=4020 SW=210200000000

     START 11.359      LINES     73       PROC  0.0003        I/O    0.001     IU  5    MEMORY    90K
     STOP  11.360      LIMIT  12000       LIMIT 0.1500        LIMIT            CU  5    M*T       340
     SWAP   0.000
     LAPSE  0.001 FC D    TYPE     BUSY     IP/AT      FP/RT      IS/#C MS/#E     ADDRESS T#/PK#

                  S* R    D400 *     15         0          1          1     1     0-08-03
                  D* R    D400 *     38         0          1          1     1     0-08-03
                  P*     SYOUT
                  *3 R    D400 *      0         0          0         72    72R    0-08-07
                  B* S    D400 *      6         0          1         36    36     0-08-01
                  K*     SYOUT
                  C*     SYOUT

      LIST     73 LINES

* ACTY-02  $CARD #0006  GELOAD    09/15/75    SW=000000000000
  * USERS PE MME GEBORT     AT 014545 I=0020 SW=000000000000

  * WRAPUP BEGUN
  * NORMAL TERMINATION      AT 002335 I=0020 SW=000000000000

     START 11.360      LINES    109       PROC  0.0007        I/O    0.001     IU  5    MEMORY    30K
     STOP  11.362      LIMIT   5000       LIMIT 0.0200        LIMIT            CU  5    M*T       295
     SWAP   0.000
     LAPSE  0.002 FC D    TYPE     BUSY     IP/AT      FP/RT      IS/#C MS/#E     ADDRESS T#/PK#

                  B* R    D400 *     19         1          1         36    36     0-08-01
                  I* R    D400 *     30         0          0          1     1     0-08-03
                  R* R    D400 *     38         0          0          1     1     0-08-03
                  P*     SYOUT
                  L* R    D400 *    455         0          0        750   750R    0-08-07
                  *L R    D400 *   2247         0          0        950   950R    0-08-09

      LIST     99 LINES
      RC-01    10 LINES
```

Figure 16-2. Complete Output Listing for First Solution

```
SNUMB = JOB16, ACTIVITY # = 01, , REPORT CODE = 74, RECORD COUNT = 000073
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
JOB16 01  09-15-75    11.359     *** HIS 6000/SERIES 60 PL/I COMPILER, VERSION 1  750314 ***              PAGE       1
          OPTIONS USED IN THIS COMPILATION


           COMPLETE LIST OF OPTIONS

                       LSTIN
                    NO LIST
                    NO MAP
                    NO SYMT
                    NO LSTOU
                    NO ALTNO
                    NO CSYM
                    NO PARSE
                    NO CHECK
                    NO OPTZ
                       SEVERITY
                    NO STAB
                    NO DECK
                    NO COMDK
                    NO SNUMBER
                    NO XREF
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
JOP16 01  09-15-75   11.359    *** HIS 6000/SERIES 60 PL/I COMPILER, VERSION 1  750314 ***             PAGE      2

        COMPILATION LISTING OF PROGRAM: P1:     PROC OPTIONS(MAIN);

    1       P1:     PROC OPTIONS(MAIN);
    2                 DCL I FIXED;
    3                 DCL (SYSIN,SYSPRINT) FILE;
    4                 PUT PAGE LIST("EXACT VALUES OF FACTORIALS");
    5                 PUT SKIP LIST("I","FACTORIAL(I)");
    6       LOOP:     GET LIST(I);
    7                 PUT SKIP LIST(I,FACT(I));
    8                 GOTO LOOP;
    9       FACT:     PROC(ARG) RETURNS(FIXED(71));
   10                   DCL ARG FIXED;
   11                   DCL RES FIXED(71);
   12                   DCL I FIXED;
   13                   IF ARG = 0 THEN RETURN(1);
   14                   RES = 1;
   15                   DO I = 1 TO ARG;
   16   1                    RES = RES*I;
   17   1                    END;
   18                   RETURN(RES);
   19                   END;
   20                 END;
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
J0316 01  09-15-75    11.359     *** HIS 6000/SERIES 60 PL/I COMPILER, VERSION 1  750314 ***            PAGE      3

         COMPILATION LISTING OF PROGRAM: P1:     PROC OPTIONS(MAIN);


*STORAGE REQUIREMENTS FOR THIS PROGRAM*

OBJECT PROGRAM SIZE IS 145 WORDS. (V COUNT 5)

EXTERNAL PROCEDURE 'P1' USES 58 WORDS OF AUTOMATIC STORAGE
INTERNAL PROCEDURE 'FACT' SHARES STACK FRAME OF EXTERNAL PROCEDURE 'P1'

*THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM*
GET←LIST←NP←AL      EXT←ENTRY           PUT←LIST←NP←AL      MPFX2               GET←TERMINATE       PUT←TERMINATE
GET←PREP            PUT←PREP

*NO EXTERNAL ENTRIES ARE CALLED BY THIS PROGRAM*

*THE FOLLOWING EXTERNAL VARIABLES ARE USED BY THIS PROGRAM*
SYSIN               SYSIN#              SYSPRINT            SYSPRINT#

*EXTERNAL NAMES AND CONVERTED NAMES OF THEM*

SYSPRINT                                                    8SRINT


**  68K WAS USED TO COMPILE THIS PROGRAM.
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
SNUMB = JOB16, ACTIVITY # = 02, , REPORT CODE = 74, RECORD COUNT = 000099
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
J0816 02  09-15-75    11.360                                                                                PAGE       1

   ORIGIN    DATE MODULE ENTRY LOCATION     ENTRY LOCATION     ENTRY LOCATION     ENTRY LOCATION     ENTRY LOCATION

                   SUBPROGRAMS INCLUDED IN DECK.

          $        OPTION  PL1,NOMAP

                   SUBPROGRAMS OBTAINED FROM SYSTEM LIBRARY

                                  RANGE                SIZE
          ALLOCATED CORE     000000 THRU 073777       074000
             RELOCATABLE     000100 THRU 062647       062550
          $        DATA     I*

          FCB AND BUFFER SPACE
             AVAILABLE       062650 THRU 073777       011130
             FILE CTRL BLKS 062650 THRU 063000        000131
             MAXIMUM BUFFER SPACE REQUIRED            001202

             27K, IS THE MINIMUM MEMORY NEEDED TO LOAD THIS ACTIVITY WITH ALL FILES OPEN      740808 2/H
          002162 LOCATIONS REQUIRED FOR LOAD TABLE
          EXECUTION PROGRAM ENTERED AT  000154   THROUGH  .PSETU
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
JOB16 02  09-15-75   11.362                PL/I EXECUTION REPORT                                                PAGE      1


**** ENDFILE CONDITION(ONCODE = 363) OCCURRED. ****

  * TRACE OF CALLS IN FORWORD ORDER *

      *  ENTRY NAME                                           LINE #   STATEMENT #   LOCATION    OFFSET     STACK

         P1                                                                         000232      000056     063000
         PLIO+$GET+VALUE+LIST+                                                      027210      000015     063104
         PLIO+$GET+FIELD+LIST+                                                      026700      000367     063232
         PLIO+SIGNAL+$S+R+                                                          015742      000510     063306
         PL1+SIGNAL+$HELP+PLIO+                                                     004474      000671     063474
         SIGNAL+$IO+SIGNAL                                                          007667      000170     064656
         PL1+SIGNAL+$DEFAULT+HANDLER                                                011143      000577     064736

      *  END OF TRACE ( IF YOU WANT TO GET THE LINE AND STATEMENT #, RECOMPILE THE PROGRAM WITH 'SNUMBER' OPTION. )


 MA 000374   MB 000374   BE  CCOCOO
 FI 204725236007   OI 000010001000   IC 014545   IR 002001   BA 600074   ER 000   AR 040040040040   QR 000000204725   TR 00013764
 X0 007447   X1 010652   X2 065210   X3 000040   X4 000000   X5 000000   X6 012066   X7 000000


*FIS REGISTERS*

000000      000000000000  0C00C0000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
000010      000000000000  0CC000000000  000000000000  000000000000  000000000000  000000000000  000000000004  000000000000


*  UPPER SSA

777000M     202003777700  203030000110  014545002001  000010000000  045071204200  045071203225  045071200200  000000000000
777010      000000000000  0CC000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
777020      000000777000  442020010133  644747204000  000020000000  777777000004  203000001100  000000000000  000374000000
777030      007447010652  065210000040  000000000000  012066000000  040040040040  000000204725  000000000000  000137645000
777040      410002040400  777777777774  000264000002  000000000000  000000000000  202000011256  202000011254  202000011251
777050      202001011244  202001011253  202001011251  600074600672  600074574000  000000004000  000000732105  377777045672
777060      000000000000  000000504122  010757775042  023400260476  000000761456  520002000002  272543462124  040000000000
777070      000012000000  777360000000  777225000005  013560000071  052002776216  000001000000  022500000000  000000000000
777100      004000000000  022403230374  022400000070  000000000000  000000000000  000000000000  000000000000  000000000000
777110      000000000000  0C0000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
777120      000000000000  000000000000  000000000000  036012000000  036062000000  777777770000  000401010071  000472010022
777130      000514010002  001124000063  002000000000  002000000000  002000000000  000000000323  000000000050  000000000100
777140      000000000000  0CC0C0000000  000000000000  000000000000  204040073200  202000630251  000000000000  777777777777
777150      000000000000  000000026061  141000005305  000000000010  400000000000  001000040000  000000000003  000000000063
777160      203102230374  203131000003  000353000523  774005000000  203101500000  014211000000  000000004754  777777770000
777170      000000000000  000000000072  203102400374  203127000001  000072000140  277777000000  203101500000  014211000000
777200      202003777700  014216014216  000000370000  000000010323  000014010006  001207010002  000000030070  000015000763
777210      000000000000  000000000000  117733000000  000000000000  002360002110  777733002020  137000304257  001203000001
777220      777220776000  250000240002  777204000000  777154000000  000000010323  000000000000  000000000000  000000000000
777230      000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
777310*     000000000000  000000000000  000000000000  000000000000  002354143343  000000000000  002344002254  000000002325
777320      004104000000  0C11C0000001  777777777777  777777777777  777777777777  777777777777  777777777777  777777777777
777330      777777777777  777777777777  777777777777  002354000000  000000003616  004400777777  000020000001  000001143350
777340      777777777777  002354143343  000000004627  004400777777  000020000000  000001143351  002350000000  000000070671
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
777350      005400000000  027340000000  001356000010  002404000000  000000430726  005400000000  035540000000  001666030240
777360      421006000001  202004777600  014545002001  202234002200  020311000200  020643200200  045071200200  000000000000
777370      000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
777710*     000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000042354  537353005443
777720      537346004354  000000044754  477341005154  437333003154  637316402254  777735006361  777737006354  777741005463
777730      777743006163  777745000000  777760000000  777753000000  777765000000  002174637001  000000000000  002174637002
777740      000151044477  002174637003  000000000000  002174637004  000071412433  002404143343  000000042553  001400000000
777750      000000000002  003244032614  000000000000  002360000000  000000107643  001000000000  000000000000  003720045454
777760      002354000000  000000016247  005000000000  000014000000  000014131544  002410777225  000000005072  002410000643
777770      022204230036  022203230036  200240000400  000000000000  000020744025  000000000000  272563464520  202020202020
000000      000000000000  001036710000  000000000000  001043710000  000000000000  001050710000  000000000000  000620710000
000010      000000000000  000641710000  000000000000  014545024725  000000000000  000045062114  000000000000  062744000002
000020      000000400000  000000000000  014545002001  000000000000  000154400000  062651000000  000000000000  002275000000
000030      000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  024146220106  063000073777
000040      002360002020  002360203753  777212204000  000020000000  020641200200  000175000001  776000000000  000136314000
000050      014216000001  000001014200  014211000000  014126000000  000000000000  014216014216  000000000000  000113560000
000060      000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  202020440204  232107050706
000070      732343446123  202343204421  414655624231  736263216331  464520232020  202020202020  202020202020  202020202020

**  28K WAS USED TO EXECUTE THIS PROGRAM.
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
SNUMB = JOB16, ACTIVITY # = 02, , REPORT CODE = 01, RECORD COUNT = 000010
```

Figure 16-2 (cont). Complete Output Listing for First Solution

```
EXACT VALUES OF FACTORIALS
I          FACTORIAL(I)
         6                                720
         0                                  1
        20                2432902008176640000
         5                                120
         9                             362880
        21               51090942171709440000
        22             1124000727777607680000
```

Figure 16-2 (cont). Complete Output Listing for First Solution

Discussion

This solution to the problem, although satisfactory for its stated purpose, has defects. A positive non-integer value is converted to an integer value. For example, the input value 8.83 produces the following output line:

```
         9          362880
```

Negative values are accepted and processed. Since the program is written only for positive integers, the following erroneous output line is produced for the input value -6:

```
        -6                1
```

Input values that are invalid or out of range terminate the run. If legitimate input values follow such invalid inputs, the legitimate values are not processed. Some examples of values that terminate processing are given here:

Input Value	Condition Raised
A1	CONVERSION
23	FIXEDOVERFLOW
131073	SIZE (in the runtime input routines)

In addition to the defects noted above in input processing, the output, although unambiguous, is not well formatted. Yet, clearly, the defects mentioned are not important if the purpose of the program is to compute several factorials for the programmer.

SECOND SOLUTION

The second solution to the defined problem is a program to be released for general use. The input format, therefore, must be carefully defined and any departure from the input format reported. Invalid input values must be detected, reported, and skipped so that the processing of valid values can continue.

The programmer defines the format to be one input value per card. The input value occupies the first two columns of the card and the remaining columns of the card must be blank. This input format is quite rigid, but it guarantees that only legal input values are processed. Invalid input values are detected and a descriptive message printed.

Deck Setup

Because the program is destined for production use, the programmer obtains a complete set of output listings for his files. The deck setup for the second solution is given in Figure 16-3.

```
1        8        16
$        SNUMB    JOB18
$        IDENT
$        OPTION   PL1
$        PL1      LIST

/* FACTORIAL PROGRAM

   PROGRAM TO COMPUTE THE EXACT VALUES OF GIVEN FACTORIALS.
   INPUT VALUE MUST BE BETWEEN 0 AND 22, INCLUSIVE.
   INPUT CARD MUST CONTAIN TWO DIGITS FOLLOWED BY 78 BLANKS.
   INVALID INPUT IS DETECTED AND NOTED IN THE OUTPUT.
   EXECUTION CONTINUES AFTER INVALID INPUT CARD.
   NORMAL TERMINATION IS AT THE INPUT END-OF-FILE.  */

P2:      PROC OPTIONS(MAIN);
           DCL  I FIXED;
           DCL  S CHAR(78);
           DCL  (SYSIN,SYSPRINT) FILE;
           DCL  (ENDFILE,CONVERSION) CONDITION;
           /*   ESTABLISH ON-UNITS */
           ON ENDFILE(SYSIN) GOTO EXIT;
           ON CONVERSION BEGIN;
                PUT EDIT("--","-- (BAD INPUT VALUE SKIPPED)")
                        (SKIP,A(2),X(21),A);
                GOTO LOOP;
                END;

           /* PRINT TITLE AND COLUMN HEADS */
           PUT EDIT("EXACT VALUES OF FACTORIALS")
                   (PAGE,A);
           PUT EDIT("I","FACTORIAL(I)")
                   (SKIP(2),X(1),A(1),X(11),A(12));
           PUT SKIP;

           /* PROCESS GIVEN VALUES */
LOOP:      GET EDIT(I,S)
                   (SKIP,P"99",A(78));
           IF 0 <= I & I <= 22
                THEN PUT EDIT(I,FACT(I))
                             (SKIP,P"Z9",X(1),P"(21)Z9");
                ELSE PUT EDIT(I,"-- (INPUT OUT OF RANGE)")
                             (SKIP,P"Z9",X(21),A);
           IF S  = (78)" "
                THEN PUT EDIT("(WARNING: COL 3-80 NOT BLANK)")
                             (X(1),A);
           GOTO LOOP;
           /* PRINT CLOSING MESSAGE  */
EXIT:      PUT EDIT("END OF FACTORIAL OUTPUT")
                   (SKIP,A);

           /* CALCULATE A FACTORIAL  */
FACT:      PROC(ARG) RETURNS(FIXED(71));
                DCL ARG FIXED;
                DCL RES FIXED(71);
                DCL K FIXED;
                RES = 1;
                DO K = 2 TO ARG;
                     RES = K*RES;
                     END;
                RETURN(RES);
                END;
           END;
```

Figure 16-3. Deck Setup for Second Solution

(cont)

```
1      8      16
$      EXECUTE
$      LIMITS  2,40K,-2K
$      DATA    I*

06
10
12
00
-5
22
23
8.83
202
X1
 4
4
04
$      ENDJOB
***EOF
```

Figure 16-3 (cont). Deck Setup for Second Solution

Output Listing

The output listing for the second solution consists of 26 pages because it includes additional listings requested by the LIST option on the $ PL1 control card. The complete output listing is, therefore, not reproduced in this manual. However, Figure 16-4 gives the information output on SYSPRINT as a result of the execution of the program.

```
EXACT VALUES OF FACTORIALS

 I            FACTORIAL(I)

 6                     720
10                 3628800
12               479001600
 0                       1
--                      -- (BAD INPUT VALUE SKIPPED)
22  1124000727777607680000
23                      -- (INPUT OUT OF RANGE)
--                      -- (BAD INPUT VALUE SKIPPED)
20     2432902008176640000 (WARNING: COL 3-80 NOT BLANK)
--                      -- (BAD INPUT VALUE SKIPPED)
--                      -- (BAD INPUT VALUE SKIPPED)
--                      -- (BAD INPUT VALUE SKIPPED)
 4                      24
END OF FACTORIAL OUTPUT
```

Figure 16-4. Output of Second Solution

Input values that are invalid are not printed because such a value could, in fact, be nonprintable. In this run, the invalid input values were the following:

```
1
_____________

-5
8.83
X1
 4
4
```

As previously noted, the input format is very rigid. To enter the input value '4', a card with a '0' in column 1 and a '4' in column 2 must be given.

The input value '202' is assumed to be the value '20', but a warning is issued that the remaining columns of the card are not blank. The input value '23' is detected by the program as being out of range.

Discussion

The second solution to the defined problem is a foolproof program in the following sense:

- Any departure from the specified input format is noted and a warning message printed.
- The program recovers from invalid input to continue processing valid inputs.

These assertions are supported by the following features of the program:

- The only value allowed by the picture is a non-negative integer.
- Columns 3 - 80 of the card are checked and any nonblank character in those columns causes a warning message to be printed.
- An ON-unit is provided to handle the CONVERSION condition, which can be raised by invalid input values.
- The range of the positive integer value is checked in the program to prevent the occurrence of the FIXEDOVERFLOW condition.

APPENDIX A

SERIES 60 (LEVEL 66)/6000 PL/I RESTRICTIONS

The following restrictions are present in the implementation of PL/I in the GCOS environment.

Identifiers

An identifier cannot contain either the character '#' or '@'.

External Names

The General Loader requires that an external name must be six or less characters in length. Any external name of more than six characters will be converted by the system (see Appendix F).

File Names

A file name must be one to five characters.

Picture Characters

Picture characters must be given as capital letters.

Peripheral Devices

The paper tape reader or paper tape punch cannot be used.

Line Size

The line size of a file with the PRINT attribute cannot exceed 136 characters.

BCD Files

BCD files cannot be handled with RECORD-oriented input-output. RECORD-oriented input-output, therefore, cannot be used for the card reader or line printer since these devices accept only BCD code.

REWRITE Statement

The REWRITE statement cannot be used for SEQUENTIAL files.

INDEXED Files

Embedded keys are the only type of keys in an INDEXED file.

REGIONAL Files

The value of a key in a REGIONAL file must be a positive integer.

The length of the character string appearing in the KEYTO option must be exactly 32 characters.

DECIMAL Attribute

In the interest of future file compatibility, the use of DECIMAL data in RECORD I/O content should be avoided. It is expected that a future version of PL/I will use a different hardware representation for DECIMAL variables.

INDEXED and REGIONAL Files

A future version of PL/I will be supported by UFAS (refer to the _UFAS_ manual) which utilizes different file formats. Present files will require conversion for compatibility.

ASCII Files

ASCII stream files written by the PL/I system and used by the COPY and SAVE directives of SRCLIB are presently not compatible with time sharing subsystems and other users of ASCII files.

APPENDIX B

COMPARISON OF SERIES 60 (LEVEL 66)/6000 PL/I AND STANDARD PL/I

This appendix lists all known deviations of the Series 60 (Level 66)/6000 PL/I language from the draft standard as of March 1974. When this PL/I language was specified, the ANSI/ECMA PL/I standardization committee had not completed its definition of PL/I. Any language issue not then resolved by the standards committee is marked by the symbol '*'.

Four types of departure from the standard are covered in this appendix, namely:

Features of standard PL/I not in this PL/I
Features restricted in this PL/I
Features implemented at variance with the standard
Extensions

The reader is assumed to be familiar with the work of the ANSI/ECMA standardization committee.

The Multics PL/I Language Manual (Order No. AG94) specifies a language very close to the proposed standard. It is a semi-formal definition of the Multics PL/I language from which this PL/I system was derived. The terminology of this appendix is consistent with that of the Multics PL/I Language Manual.

FEATURES OF STANDARD PL/I NOT IN SERIES 60 (LEVEL 66)/6000 PL/I

The following features are part of standard PL/I, but are not included in this PL/I.

1. The BYNAME option in the assignment statement.

2. The 'T', 'I', and 'R' picture characters.

3. The TAB option and TAB format item. In this PL/I the same effect can be obtained by control cards at execution time.

FEATURES RESTRICTED IN PL/I

The following features are restricted in this PL/I.

1. A literal constant cannot contain a scale factor (F±n) or a default suppression character (P).

2. A bit string constant cannot be expressed in octal or hexadecimal, and the bit string format has no provision for processing octal or hexadecimal.

3. Only one prefix subscript is permitted in a label prefix.

4. The condition names defined by the language are reserved. A user-defined condition cannot have the same name as a language-defined condition.

5. A condition name cannot have INTERNAL scope.

6. Only one attribute set is allowed in a DEFAULT statement.

7. The extents of variables with the STATIC attribute must be decimal integers. The expressions in the INITIAL attribute for a STATIC variable are restricted to optionally signed literal constants, pairs of real and imaginary signed literal constants, or the NULL and EMPTY built-in functions.

8. The label prefix of a PROCEDURE ENTRY, or FORMAT statement cannot contain a prefix subscript.

9. The STRING built-in function requires that its argument be a scalar or an aggregate consisting of either packed bit string data or packed character string data.

10. If two structures share storage, their alignment attributes must match.

11. Only one condition name is allowed in an ON statement.

12. All condition prefixes of a statement must precede any label prefixes of the statement.

13. An AREA variable cannot be used as the index of a DO statement.

14. DEFINED variables whose DEFINED attribute contains either ISUBS or asterisks cannot be input or output by a GET or PUT statement that specifies data directed transmission.

15. File constants cannot have the DIMENSION attribute.

16. If the expression of an assignment statement is a reference that identifies a scalar string variable, then no target of the assignment statement can identify a generation of storage that overlaps the generation of storage of the string variable, unless it is exactly the same generation.

17. Asterisk extents must be used when passing an unconnected array as an array parameter. An unconnected array is an array whose elements are separated from one another in storage by other values.

18. When one array shares storage with another array by simple defining, the base reference must contain an asterisk for each dimension of the DEFINED array.

19. The pointer value yielded by the ADDR built-in function applied to a parameter is valid only as long as the block activation to which the corresponding argument was passed is still active. This restriction applies to the case in which the standard option OPTZ is given on the $ PL1 control card.

20. The standard allows an array of scalars to be promoted to an array of structures, but this PL/I does not allow this promotion.

21. A simple or ISUB defined variable must have extents that equal the corresponding extents of the base variable on which it is defined. The standard allows the extents to be less than or equal to the extents of the base variable.

22. In structure promotion of the form S=R or S+R, this PL/I requires that the aggregate type of each member of S match the aggregate type of the corresponding member of R. The standard performs aggregate promotion for members that do not match.

*23. The DOT built-in function requires that the precision of its result be given in the function reference.

24. Both the IGNORE option and the KEY option cannot be given in the same READ statement.

25. The INTO option of a READ statement may not reference a VARYING string.

FEATURES IMPLEMENTED AT VARIANCE WITH THE STANDARD

The implementation of the following features produces a different effect than specified in the standard.

1. Return from an ON-unit entered by a signal of the AREA condition causes the allocation to be re-attempted in the original area, without reevaluation of the IN option.

2. The bounds of an evaluated array expression are always normalized such that each lower bound is one and each upper bound is the number of elements in the dimension.

3. A mismatch between the alignment attributes of a structure and a structure parameter descriptor causes the argument to be passed by-value rather than by-reference. The standard ignores the alignment attributes of structures.

4. The STRINGSIZE condition is disabled by default in this PL/I, but enabled in standard PL/I.

EXTENSIONS

The following features are included in this PL/I but are not part of standard PL/I.

1. An identifier can contain the special character '$'. In the case of external names, this character has additional semantics.

2. Varying length strings can be used in simple and ISUB defining.

3. The base variable identified by a DEFINED attribute can be a BASED variable.

4. Most restrictions on the REFER option are removed.

5. Several new built-in functions are implemented.

6. The INCLUDE macro is implemented.

7. The LOCAL attribute is allowed in all descriptors.

8. BASED variables can be output by a PUT statement that specifies data-directed output.

9. An IN option is not required in a FREE statement when freeing a generation of storage allocated in an area.

10. The RECURSIVE keyword is never required in a PROCEDURE statement. The system always generates code for a procedure that allows recursive calls.

11. The UNSPEC pseudo variable allows aggregate arguments.

12. Assignments and infix operations can be performed on two arrays of unequal bounds if the number of dimensions is equal and the number of elements in each dimension of one array is equal to the number of elements in the corresponding dimension of the other array.

13. A replication factor in a PICTURE can be zero. A zero replication factor indicates that the picture character to which it applies is to be deleted from the normal picture produced by the translation of the PICTURE.

14. A name declared with the ENVIRONMENT attribute acquires the FILE attribute by default. A name declared with the OPTIONS attribute acquires the ENTRY attribute by default. The standard does not give defaults for these cases.

15. The COLUMN option can be used by a GET or PUT statement containing the STRING option.

16. The standard considers the case in which an array is passed as an argument to an array parameter that has different bounds but equal extents to be an error. This PL/I assigns the argument to an array temporary whose bounds are equal to the bounds of the array parameter.

17. A picture scale factor is allowed for floating point pictures.

18. The REDUCIBLE and IRREDUCIBLE attributes are allowed.

19. No delimiter is required between the keywords PICTURE or PIC and the quoted picture in a picture attribute. No delimiter is required between the letter P and the quoted picture in a picture format.

APPENDIX C

MEMORY REQUIREMENTS

MEMORY ESTIMATION

To estimate the memory size required for the execution of a program compiled by this PL/I compiler, the following items must be considered:

The size of the object program to be allocated.

Storage allocated dynamically at execution time for:

AUTOMATIC variables
BASED variables
CONTROLLED variables
ON-units

Library routines provided by the system at execution time.

The memory required is calculated by adding the requirements of the above items.

RECOMMENDED PROCEDURE FOR STORAGE USE

It is often difficult to predict the required memory size due to the program logic and input data. However, after the job is executed, the system prints the memory size that was actually used, as follows:

```
**31K WAS USED TO EXECUTE THIS PROGRAM.
```

Therefore, the recommended procedure is to specify a slightly oversized memory requirement for the first execution of the job on the $ LIMIT control card and, after execution, to replace that estimate by the actual memory used.

MINIMUM MEMORY REQUIREMENTS

Approximate minimum memory requirements for different job types are given below:

Job Type	Minimum Memory
STREAM list-directed transmission	27K
STREAM edit-directed transmission	28K
STREAM data-directed transmission	30K
RECORD CONSECUTIVE organization	16K
RECORD INDEXED organization	20K
RECORD REGIONAL organization	22K

The minimums are composed of basic run-time support routines as well as input-output support routines. Therefore, job type combinations cannot be predicted by simple addition. A job having files of both STREAM list-directed and RECORD REGIONAL takes 29K while the combination of STREAM list-directed and RECORD INDEXED takes 36K. A combination of STREAM list-directed and data-directed needs at least 32K and a combination of STREAM data-directed with RECORD CONSECUTIVE needs 34K.

APPENDIX D

CHARACTER CONVERSION TABLES

This section contains three character conversion tables:

IBMEL to ASCII Conversion Table (Table D-1)
GBCD to ASCII Conversion Table (Table D-2)
ASCII to GBCD and IBMEL Conversion Table (Table D-3)

The first two tables provide the input conversion rules that are used for creating a PL/I source program and for input data. The third table provides the output conversion rules that are used for writing data out to external media.

Table D-1. Character Conversion Table (IBMEL to ASCII)

IBMEL Character	IBMEL Card Punch	GBCD Character	GBCD Card Punch	GBCD Internal Code	ASCII Character	ASCII Internal Code
blank	blank	blank	blank	20	blank	040
!	11-8-2(11-0)	!	0-7-8	77	!	041
'	8-7	'	0-6-8	76	"	042
#	8-3	#	3-8	13	#	043
$	11-8-3	$	11-3-8	53	$	044
%	0-8-4	%	0-4-8	74	%	045
&	12	&	12	32	&	046
'	8-5	'	11-7-8	57	'	047
(	12-8-5	(	12-5-8	35	(	050
)	11-8-5	)	11-5-8	55	)	051
*	11-8-4	*	11-4-8	54	*	052
+	12-8-6	+	12-0	60	+	053
,	0-8-3	,	0-3-8	73	,	054
-	11	-	11	52	-	055
.	12-8-3	.	12-3-8	33	.	056
/	0-1	/	0-1	61	/	057
0	0	0	0	00	0	060
1	1	1	1	01	1	061
2	2	2	2	02	2	062
3	3	3	3	03	3	063
4	4	4	4	04	4	064
5	5	5	5	05	5	065
6	6	6	6	06	6	066
7	7	7	7	07	7	067
8	8	8	8	10	8	070
9	9	9	9	11	9	071
:	8-2	:	5-8	15	:	072
;	11-8-6	;	11-6-8	56	;	073
<	12-8-4	<	12-6-8	36	<	074
=	8-6	=	0-5-8	75	=	075
>	0-8-6	>	6-8	16	>	076
?	0-8-7	?	7-8	17	?	077
@	8-4	@	4-8	14	@	100
A	12-1	A	12-1	21	A	101
B	12-2	B	12-2	22	B	102
C	12-3	C	12-3	23	C	103
D	12-4	D	12-4	24	D	104
E	12-5	E	12-5	25	E	105
F	12-6	F	12-6	26	F	106
G	12-7	G	12-7	27	G	107
H	12-8	H	12-8	30	H	110
I	12-9	I	12-9	31	I	111
J	11-1	J	11-1	41	J	112
K	11-2	K	11-2	42	K	113
L	11-3	L	11-3	43	L	114
M	11-4	M	11-4	44	M	115
N	11-5	N	11-5	45	N	116
O	11-6	O	11-6	46	O	117
P	11-7	P	11-7	47	P	120
Q	11-8	Q	11-8	50	Q	121

Table D-1 (cont). Character Conversion Table (IBMEL to ASCII)

IBMEL Character	IBMEL Card Punch	GBCD Character	GBCD Card Punch	GBCD Internal Code	ASCII Character	ASCII Internal Code
R	11-9	R	11-9	51	R	122
S	0-2	S	0-2	62	S	123
T	0-3	T	0-3	63	T	124
U	0-4	U	0-4	64	U	125
V	0-5	V	0-5	65	V	126
W	0-6	W	0-6	66	W	127
X	0-7	X	0-7	67	X	130
Y	0-8	Y	0-8	70	Y	131
Z	0-9	Z	0-9	71	Z	132
¢	12-8-2(12-0)	[	2-8	12	[	133
	0-8-2	\	12-7-8	37	\	134
|	12-8-7	]	12-4-8	34	]	135
¬	11-8-7	↑	11-0	40	^	136
_	0-8-5	←	0-2-8	72	_	137

Table D-2. Character Conversion Table (GBCD to ASCII)

GBCD Character	GBCD Card Punch	GBCD Internal Code	ASCII Character	ASCII Internal Code	IBMEL Character	IBMEL Card Punch
0	0	00	0	060	0	0
1	1	01	1	061	1	1
2	2	02	2	062	2	2
3	3	03	3	063	3	3
4	4	04	4	064	4	4
5	5	05	5	065	5	5
6	6	06	6	066	6	6
7	7	07	7	067	7	7
8	8	10	8	070	8	8
9	9	11	9	071	9	9
[	2-8	12		133	¢	12-8-2(12-0)
#	3-8	13	#	043	#	8-3
@	4-8	14	@	100	@	8-4
:	5-8	15	:	072	:	8-2
>	6-8	16	>	076	>	0-8-6
?	7-8	17	?	077	?	0-8-7
blank	blank	20	blank	040	blank	blank
A	12-1	21	A	101	A	12-1
B	12-2	22	B	102	B	12-2
C	12-3	23	C	103	C	12-3
D	12-4	24	D	104	D	12-4
E	12-5	25	E	105	E	12-5
F	12-6	26	F	106	F	12-6
G	12-7	27	G	107	G	12-7
H	12-8	30	H	110	H	12-8
I	12-9	31	I	111	I	12-9
&	12	32	&	046	&	12
.	12-3-8	33	.	056	.	12-8-3
]	12-4-8	34	]	135	|	12-8-7
(	12-5-8	35	(	050	(	12-8-5
<	12-6-8	36	<	074	<	12-8-4
\	12-7-8	37	\	134		0-8-2
↑	11-0	40	^	136	¬	11-8-7
J	11-1	41	J	112	J	11-1
K	11-2	42	K	113	K	11-2
L	11-3	43	L	114	L	11-3
M	11-4	44	M	115	M	11-4
N	11-5	45	N	116	N	11-5
O	11-6	46	O	117	O	11-6
P	11-7	47	P	120	P	11-7
Q	11-8	50	Q	121	Q	11-8
R	11-9	51	R	122	R	11-9
-	11	52	-	055	-	11
$	11-3-8	53	$	044	$	11-8-3
*	11-4-8	54	*	052	*	11-8-4

Table D-2 (cont). Character Conversion Table (GBCD to ASCII)

GBCD Character	GBCD Card Punch	GBCD Internal Code	ASCII Character	ASCII Internal Code	IBMEL Character	IBMEL Card Punch
)	11-5-8	55	)	051	)	11-8-5
;	11-6-8	56	;	073	;	11-8-6
'	11-7-8	57	'	047	'	8-5
+	12-0	60	+	053	+	12-8-6
/	0-1	61	/	057	/	0-1
S	0-2	62	S	123	S	0-2
T	0-3	63	T	124	T	0-3
U	0-4	64	U	125	U	0-4
V	0-5	65	V	126	V	0-5
W	0-6	66	W	127	W	0-6
X	0-7	67	X	130	X	0-7
Y	0-8	70	Y	131	Y	0-8
Z	0-9	71	Z	132	Z	0-9
<-	0-2-8	72	_	137	_	0-8-5
,	0-3-8	73	,	054	,	0-8-3
%	0-4-8	74	%	045	%	0-8-4
=	0-5-8	75	=	075	=	8-6
'	0-6-8	76	"	042	'	8-7
!	0-7-8	77	!	041	!	11-8-2(11-0)

Table D-3. Character Conversion Table (ASCII to GBCD and IBMEL)

ASCII Character	ASCII Internal Code	GBCD Character	GBCD Card Punch	GBCD Internal Code	IBMEL Character	IBMEL Card Punch
blank	040	blank	blank	20	blank	blank
!	041	!	0-7-8	77	!	11-8-2(11-0)
"	042	'	0-6-8	76	'	8-7
#	043	#	3-8	13	#	8-3
$	044	$	11-3-8	53	$	11-8-3
%	045	%	0-4-8	74	%	0-8-4
&	046	&	12	32	&	12
'	047	'	11-7-8	57	'	8-5
(	050	(	12-5-8	35	(	12-8-5
)	051	)	11-5-8	55	)	11-8-5
*	052	*	11-4-8	54	*	11-8-4
+	053	+	12-0	60	+	12-8-6
,	054	,	0-3-8	73	,	0-8-3
-	055	-	11	52	-	11
.	056	.	12-3-8	33	.	12-8-3
/	057	/	0-1	61	/	0-1
0	060	0	0	00	0	0
1	061	1	1	01	1	1
2	062	2	2	02	2	2
3	063	3	3	03	3	3
4	064	4	4	04	4	4
5	065	5	5	05	5	5
6	066	6	6	06	6	6
7	067	7	7	07	7	7
8	070	8	8	10	8	8
9	071	9	9	11	9	9
:	072	:	5-8	15	:	8-2
;	073	;	11-6-8	56	;	11-8-6
<	074	<	12-6-8	36	<	12-8-4
=	075	=	0-5-8	75	=	8-6
>	076	>	6-8	16	>	0-8-6
?	077	?	7-8	17	?	0-8-7
@	100	@	4-8	14	@	8-4
A	101	A	12-1	21	A	12-1
B	102	B	12-2	22	B	12-2
C	103	C	12-3	23	C	12-3
D	104	D	12-4	24	D	12-4
E	105	E	12-5	25	E	12-5
F	106	F	12-6	26	F	12-6
G	107	G	12-7	27	G	12-7
H	110	H	12-8	30	H	12-8
I	111	I	12-9	31	I	12-9
J	112	J	11-1	41	J	11-1
K	113	K	11-2	42	K	11-2
L	114	L	11-3	43	L	11-3
M	115	M	11-4	44	M	11-4
N	116	N	11-5	45	N	11-5
O	117	O	11-6	46	O	11-6

Table D-3 (cont). Character Conversion Table (ASCII to GBCD and IBMEL)

ASCII Character	ASCII Internal Code	GBCD Character	GBCD Card Punch	GBCD Internal Code	IBMEL Character	IBMEL Card Punch
P	120	P	11-7	47	P	11-7
Q	121	Q	11-8	50	Q	11-8
R	122	R	11-9	51	R	11-9
S	123	S	0-2	62	S	0-2
T	124	T	0-3	63	T	0-3
U	125	U	0-4	64	U	0-4
V	126	V	0-6	66	V	0-6
W	127	W	0-6	66	W	0-6
X	130	X	0-7	67	X	0-7
Y	131	Y	0-8	70	Y	0-8
Z	132	Z	0-9	71	Z	0-9
[	133	[	2-8	12	¢	12-8-2(12-0)
\	134	\	12-7-8	37		0-8-2
]	135	]	12-4-8	34	|	12-8-7
^	136	↑	11-0	40	¬	11-8-7
_	137	←	0-2-8	72	_	0-8-5
'	140	*	11-4-8	54	*	11-8-4
a	141	A	12-1	21	A	12-1
b	142	B	12-2	22	B	12-2
c	143	C	12-3	23	C	12-3
d	144	D	12-4	24	D	12-4
e	145	E	12-5	25	E	12-5
f	146	F	12-6	26	F	12-6
g	147	G	12-7	27	G	12-7
h	150	H	12-8	30	H	12-8
i	151	I	12-9	31	I	12-9
j	152	J	11-1	41	J	11-1
k	153	K	11-2	42	K	11-2
l	154	L	11-3	43	L	11-3
m	155	M	11-4	44	M	11-4
n	156	N	11-5	45	N	11-5
o	157	O	11-6	46	O	11-6
p	160	P	11-7	47	P	11-7
q	161	Q	11-8	50	Q	11-8
r	162	R	11-9	51	R	11-9
s	163	S	0-2	62	S	0-2
t	164	T	0-3	63	T	0-3
u	165	U	0-4	64	U	0-4
v	166	V	0-5	65	V	0-5
w	167	W	0-6	66	W	0-6
x	170	X	0-7	67	X	0-7
y	171	Y	0-8	70	Y	0-8
z	172	Z	0-9	71	Z	0-9
{	173	\	12-7-8	37		0-8-2
¦	174	!	0-7-8	77	!	12-8-7
}	175	\	11-4-8	54		0-8-2
~	176	\	11-4-8	54		0-8-2
°	241	\	11-4-8	54		0-8-2
⌐	242	\	11-4-8	54		0-8-2
┘	243	\	11-4-8	54		0-8-2
ˊ	244	\	11-4-8	54		0-8-2
·	245	\	11-4-8	54		0-8-2

APPENDIX E

INTERNAL REPRESENTATION OF PL/I DATA TYPES

This appendix gives the internal representation for each of the PL/I data types in both the UNALIGNED and ALIGNED cases. The data types are given by classification, as follows:

Arithmetic
String
Address
Area

The boundary requirement and default alignment are summarized for each data type in the classification. Diagrams for the UNALIGNED and ALIGNED representation for each data type then follow.

ARITHMETIC DATA TYPES

The arithmetic data types, their boundary requirements, and their default alignment are listed here:

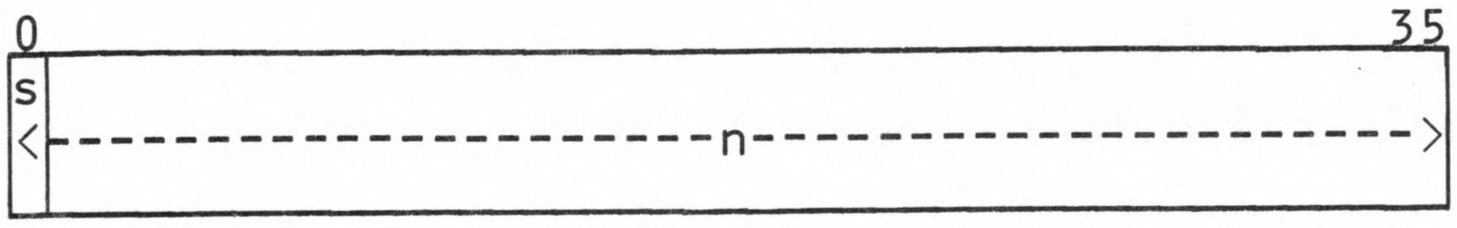
		Boundary Required		Default
Data Type	Prec.	UNALIGNED	ALIGNED	Alignment
REAL FIXED BINARY	single	bit	word	ALIGNED
	double	bit	even-word	ALIGNED
REAL FIXED DECIMAL		byte	word	ALIGNED
REAL FLOAT BINARY	single	bit	word	ALIGNED
	double	bit	even-word	ALIGNED
REAL FLOAT DECIMAL		byte	word	ALIGNED
COMPLEX FIXED BINARY	single	bit	even-word	ALIGNED
	double	bit	even-word	ALIGNED
COMPLEX FIXED DECIMAL		byte	word	ALIGNED
COMPLEX FLOAT BINARY	single	bit	even-word	ALIGNED
	double	bit	even-word	ALIGNED
COMPLEX FLOAT DECIMAL		byte	word	ALIGNED

Real Fixed-Point Binary

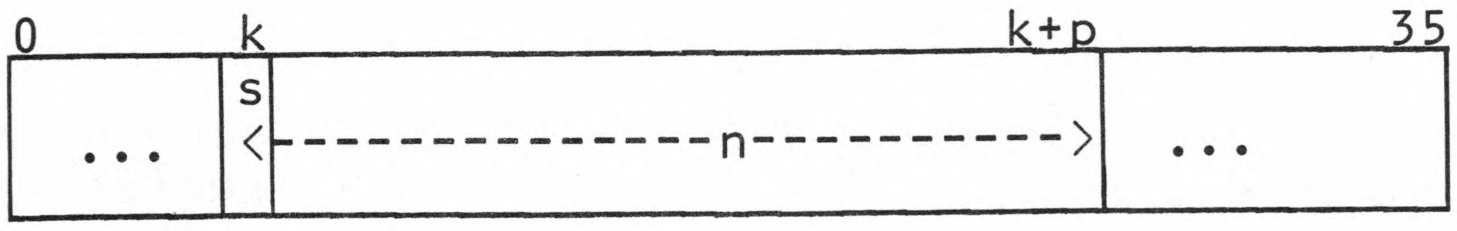
A real fixed-point binary number of precision (p,q) is stored as a two's complement binary number, n, as follows:

SINGLE PRECISION $0<p<36$

If ALIGNED, the number is positioned at a word boundary and occupies one word, as follows:

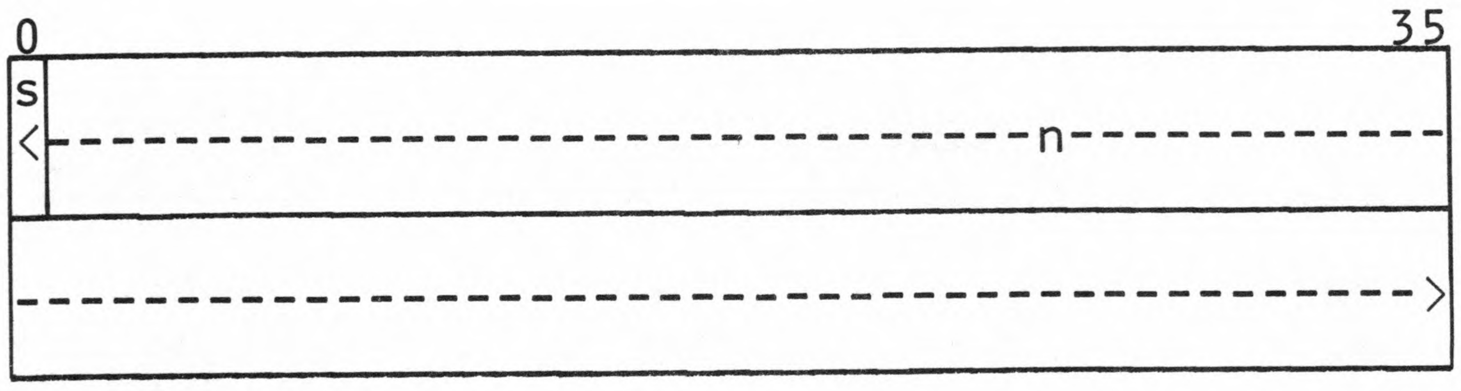


If UNALIGNED within a structure, the number is positioned at a bit boundary and occupies p+1 bits, as follows:

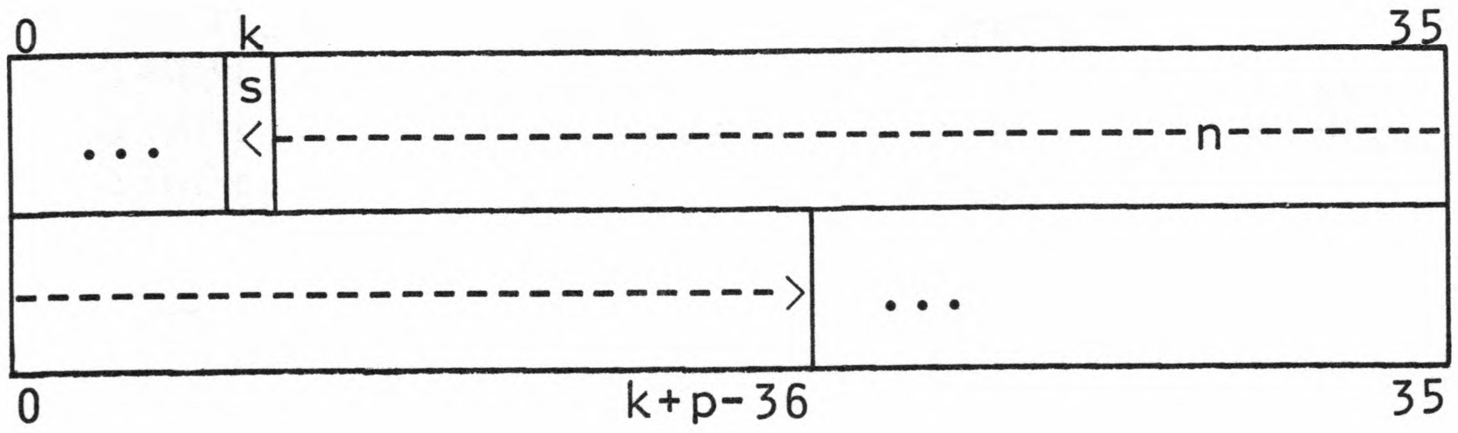


DOUBLE PRECISION $35<p<72$

If ALIGNED, the number is positioned at an even-word boundary and occupies two words, as follows:



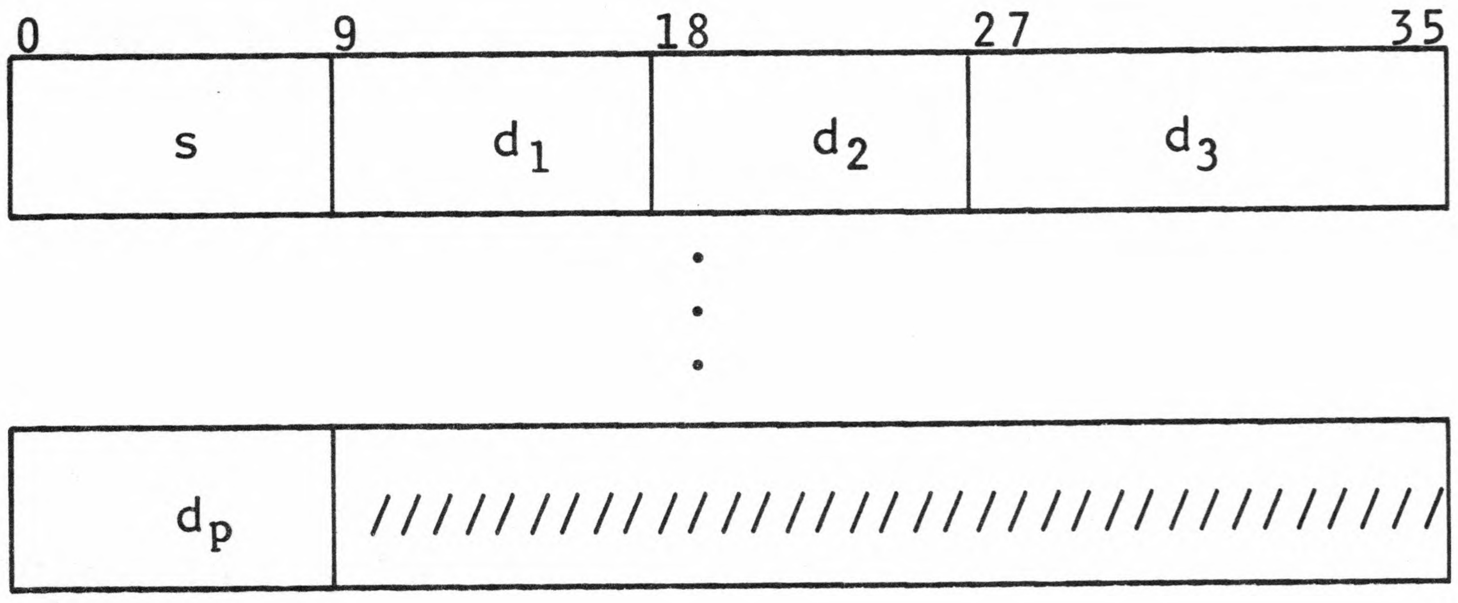
If UNALIGNED within a structure, the number is positioned at a bit boundary and occupies p+1 bits, as follows:



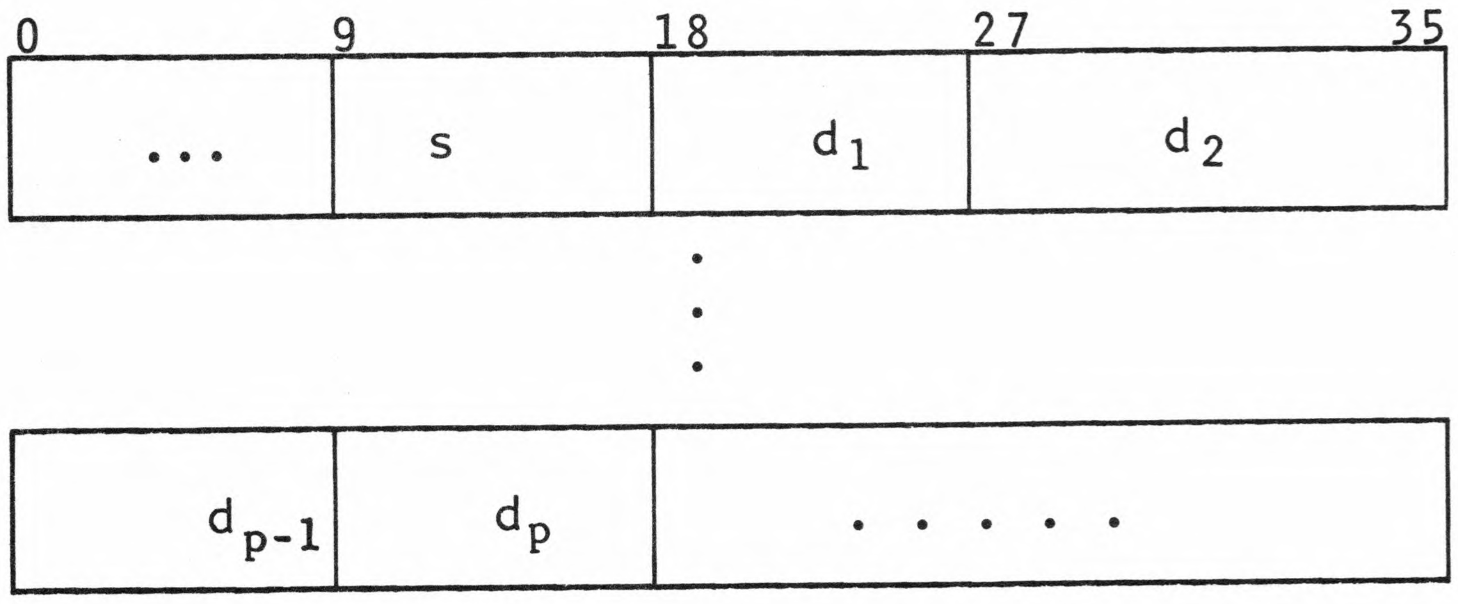
Real Fixed-Point Decimal

A real fixed-point decimal number of precision p is stored as a string of p+1 characters, as follows:

If ALIGNED, the number is positioned at a word boundary and occupies an integral number of words; some trailing bytes may be unused.



If UNALIGNED within a structure, the number is positioned at a byte boundary and occupies p+1 bytes, as follows:



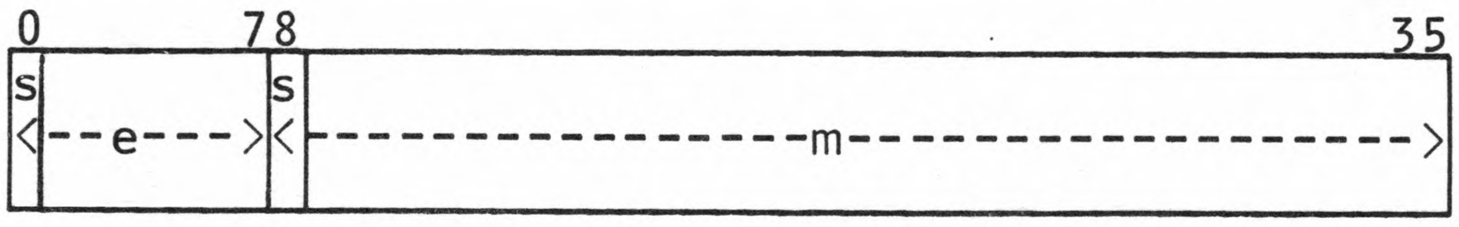
The left most character is the sign, either '+' or '-', and the remaining characters are from the set '0123456789'.

Real Floating-Point Binary

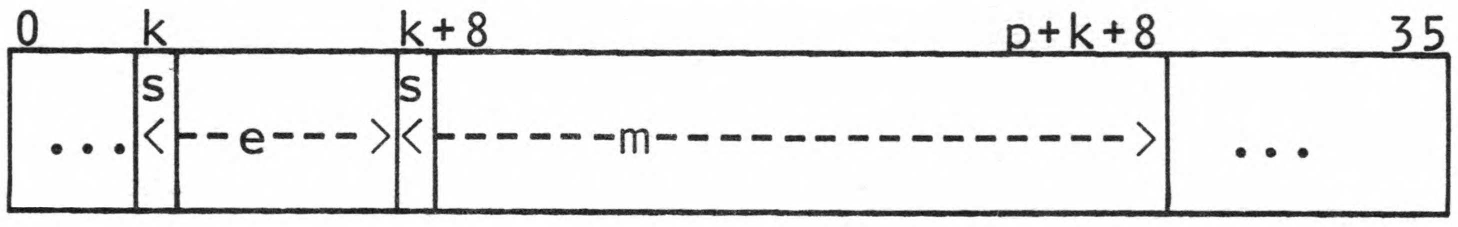
A real floating-point binary number of precision (p,q) is stored as a two's complement binary fractional mantissa, m, and a two's complement binary integer exponent, e, as follows:

SINGLE PRECISION 0<p<28

If ALIGNED, the number is positioned at a word boundary and occupies one word, as follows:

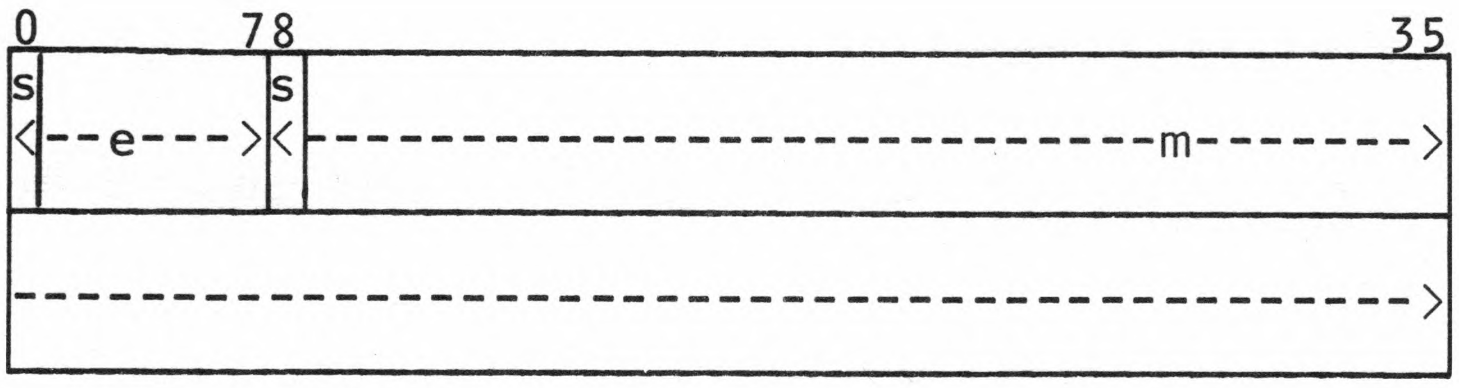


If UNALIGNED within a structure, the number is positioned at a bit boundary and occupies p+9 bits, as follows:

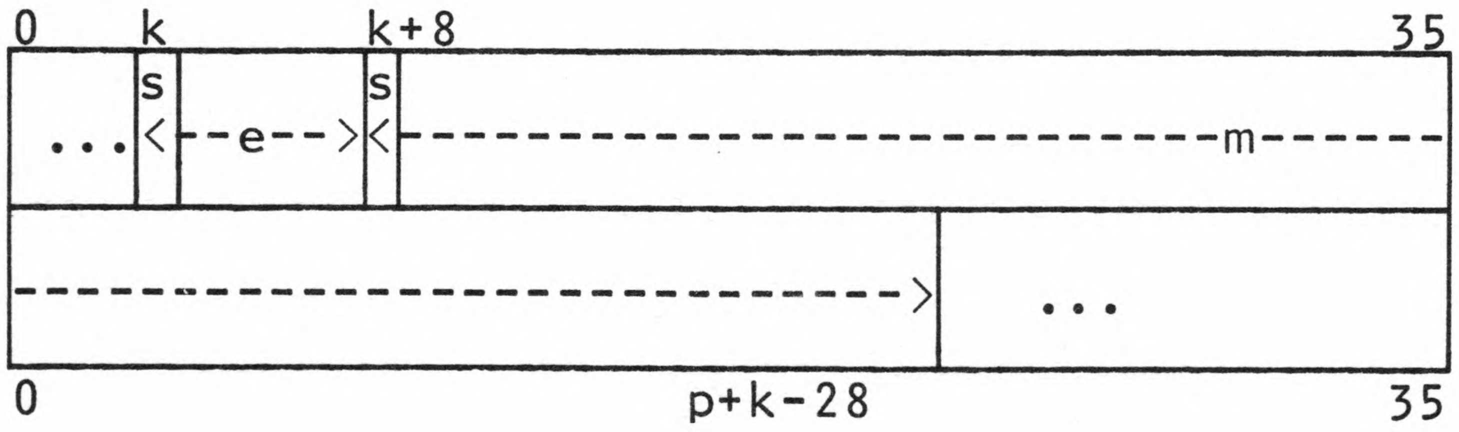


DOUBLE PRECISION 27<p<64

If ALIGNED, the number is positioned at an even-word boundary and occupies two words, as follows:



If UNALIGNED within a structure, the number is positioned at a bit boundary and occupies p+9 bits, as follows:

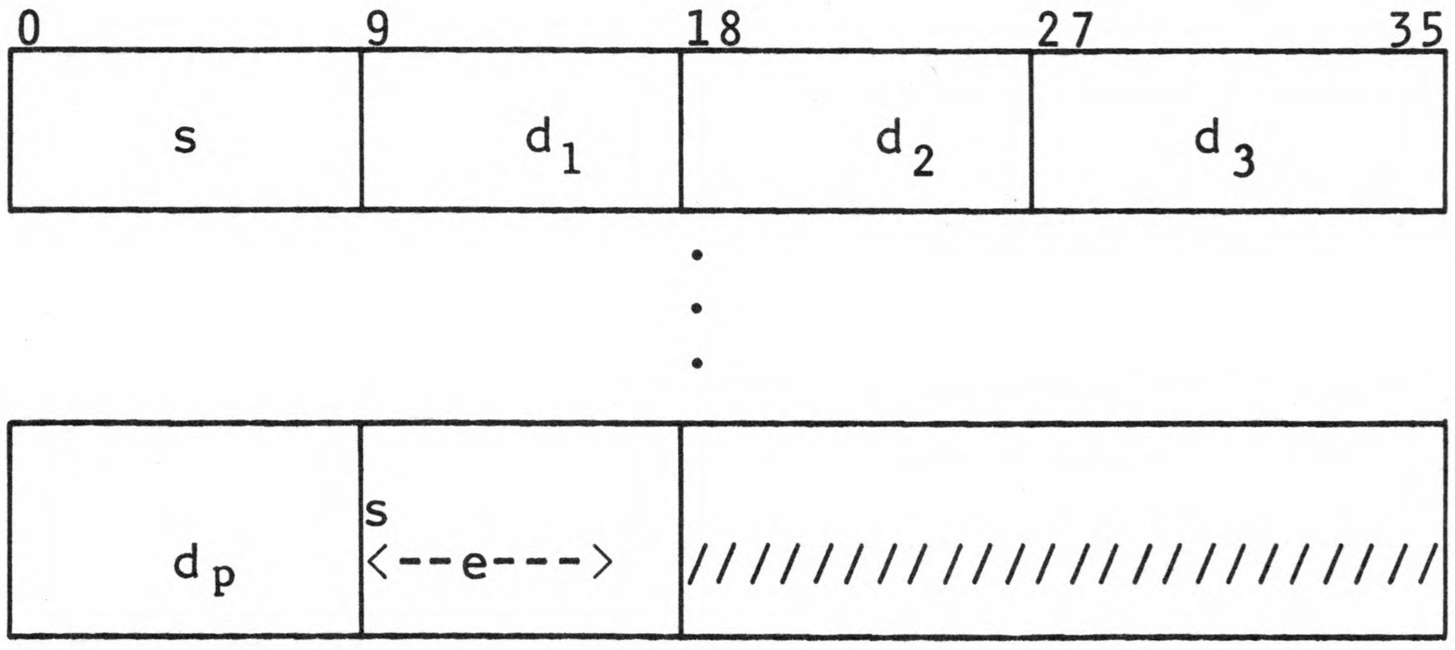


The value zero is represented as m=0 and e=-128.

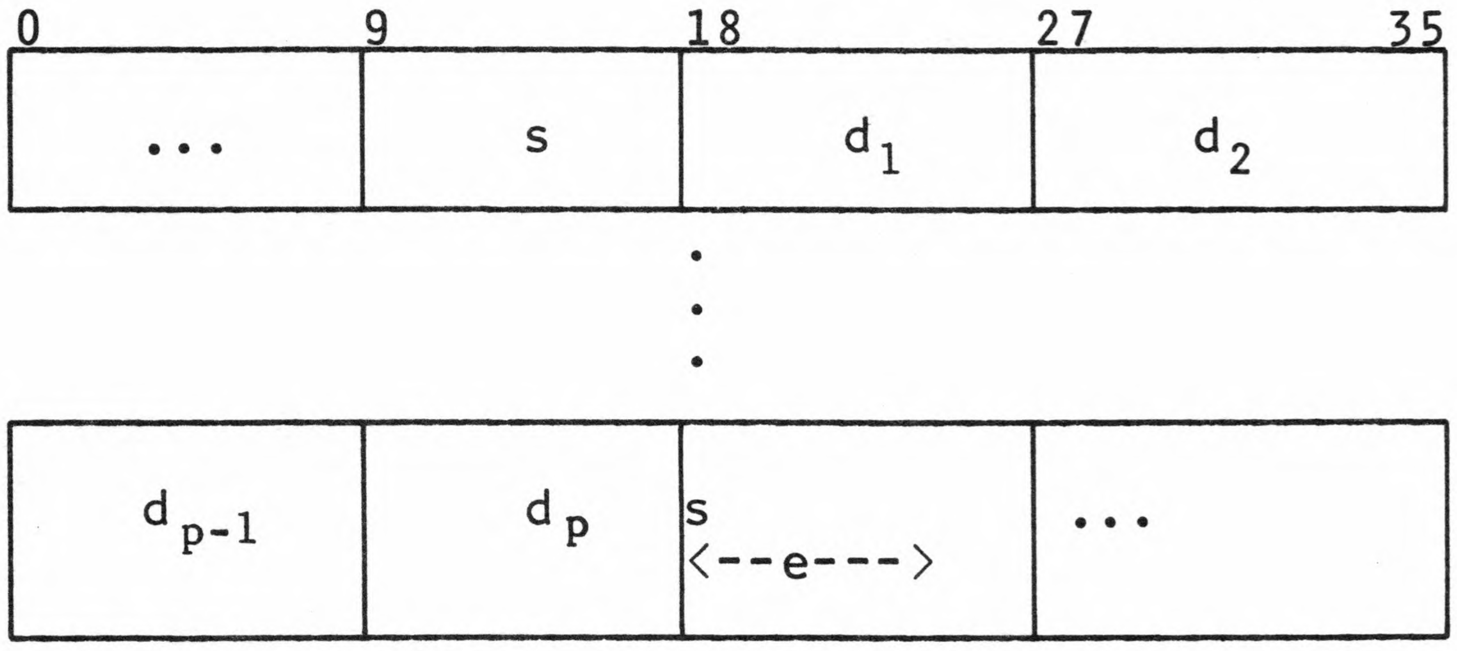
Real Floating-Point Decimal

A real floating-point decimal number of precision p is stored as a signed decimal integer, m, and a 9-bit, two's complement binary integer exponent, as follows:

If ALIGNED, the number is positioned at a word boundary and occupies an integral number of words, as follows:



If UNALIGNED within a structure, the number is positioned at a byte boundary and occupies p+2 bytes, as follows:

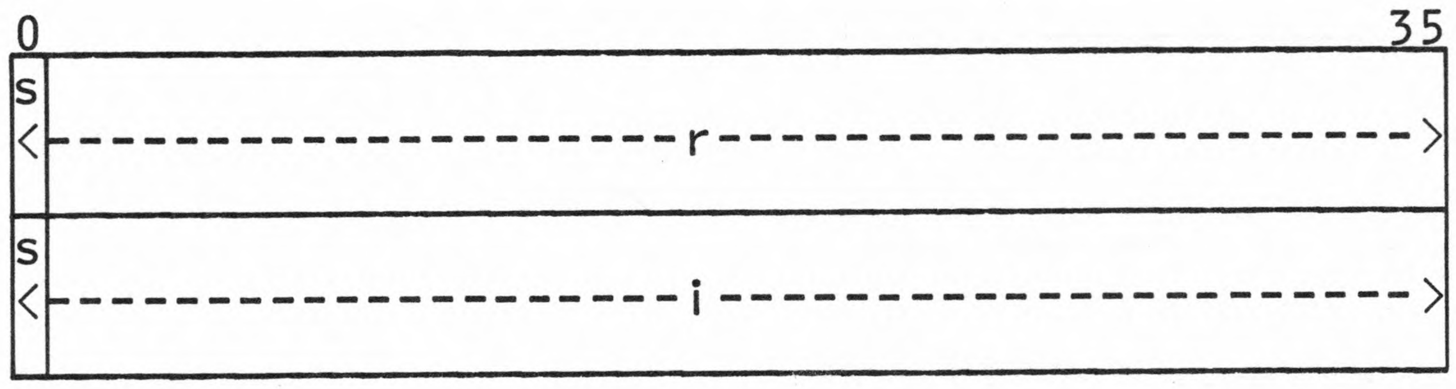


Complex Fixed-Point Binary

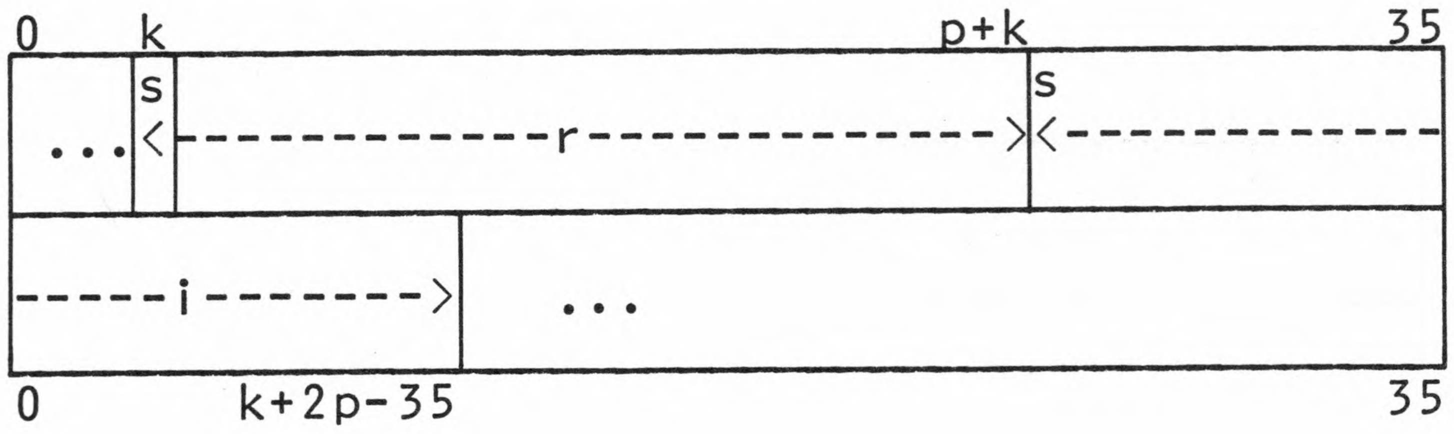
A complex fixed-point binary number is stored as a pair of two's complement binary integers. The first integer, r, is the real part of the complex value and the second integer, i, is the imaginary part of the complex value.

SINGLE PRECISION 0<p<36

If ALIGNED, the number is positioned on an even-word boundary and occupies two words, as follows:

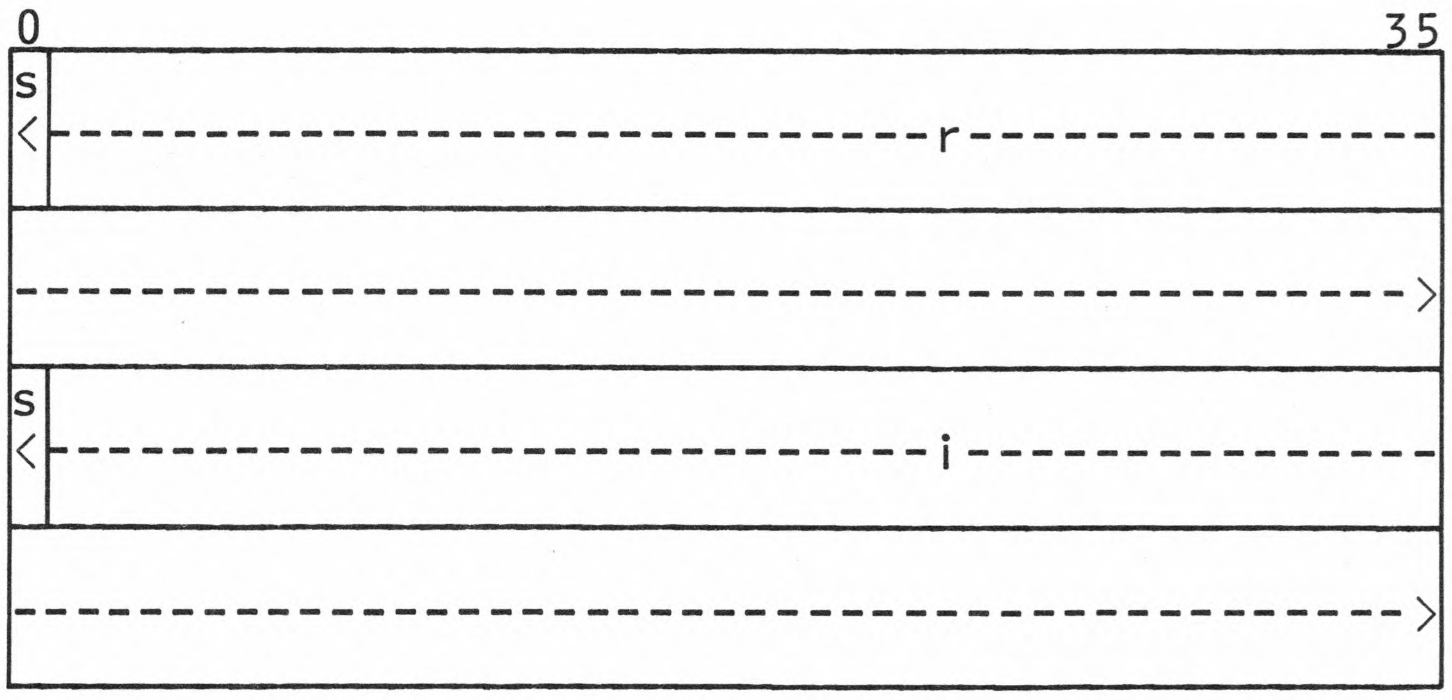


If UNALIGNED within a structure, the number is positioned at a bit boundary and occupies 2(p+1) bits, as follows:

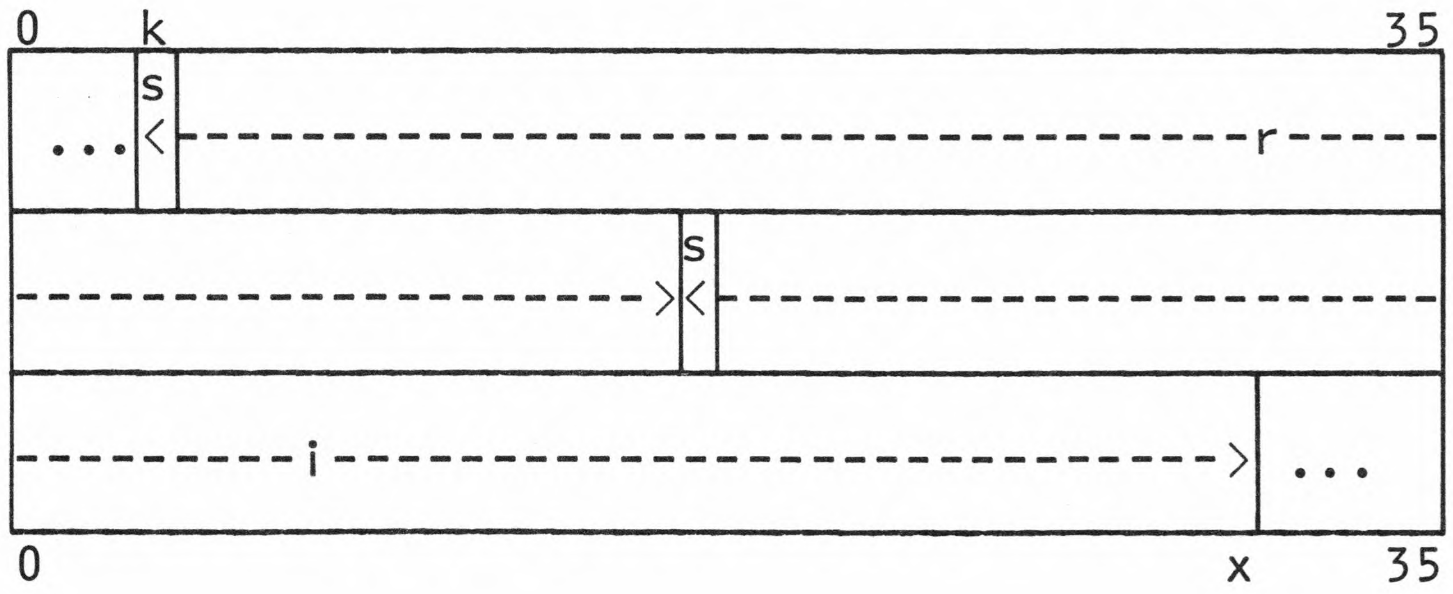


DOUBLE PRECISION 35<p<72

If ALIGNED, the number is positioned at an even-word boundary, and occupies four words, as follows:



If UNALIGNED within a structure, the number is positioned at a bit boundary and occupies 2(p+1) bits, as follows:

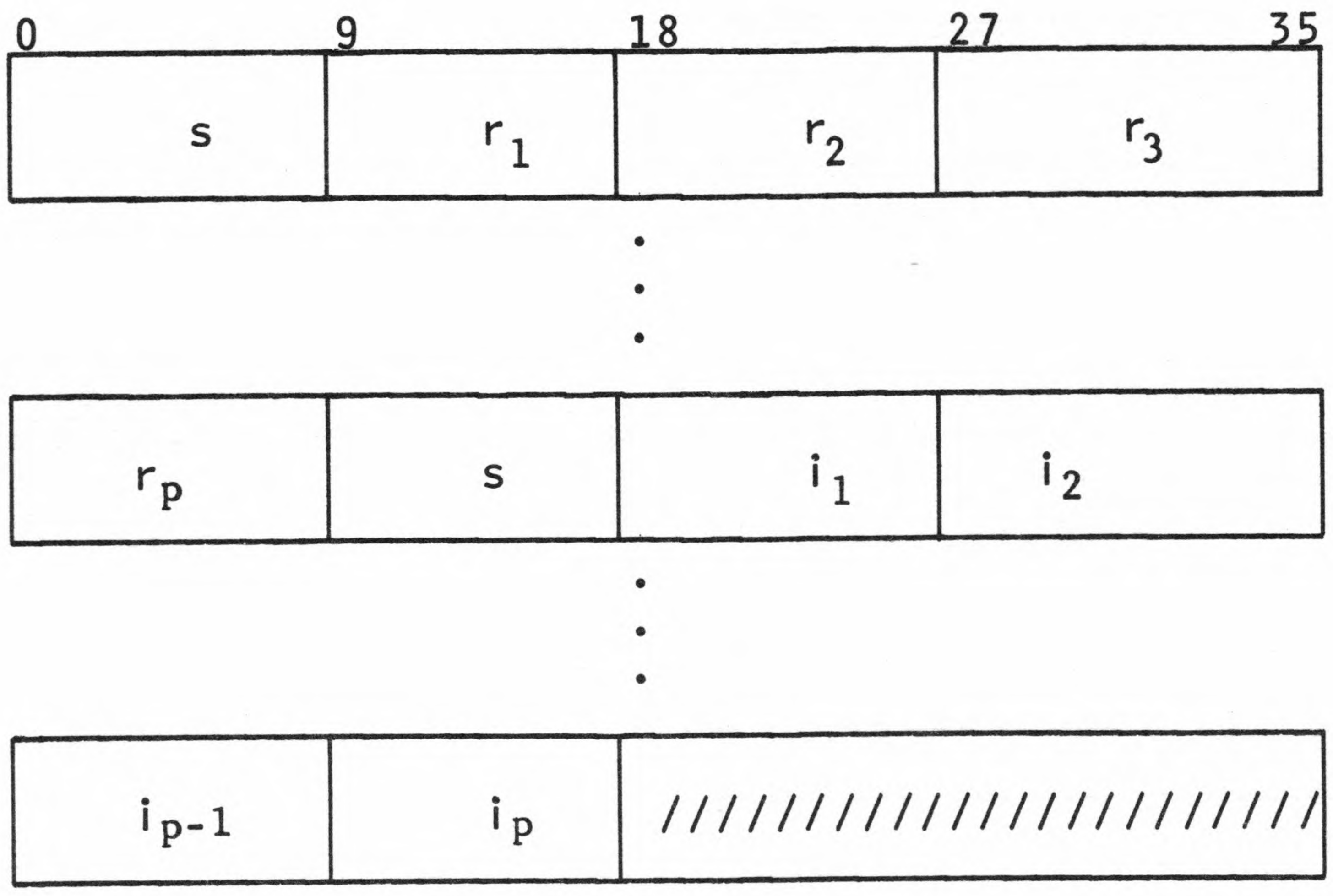


The ending bit position x has the value MOD(p+k+1, 36).

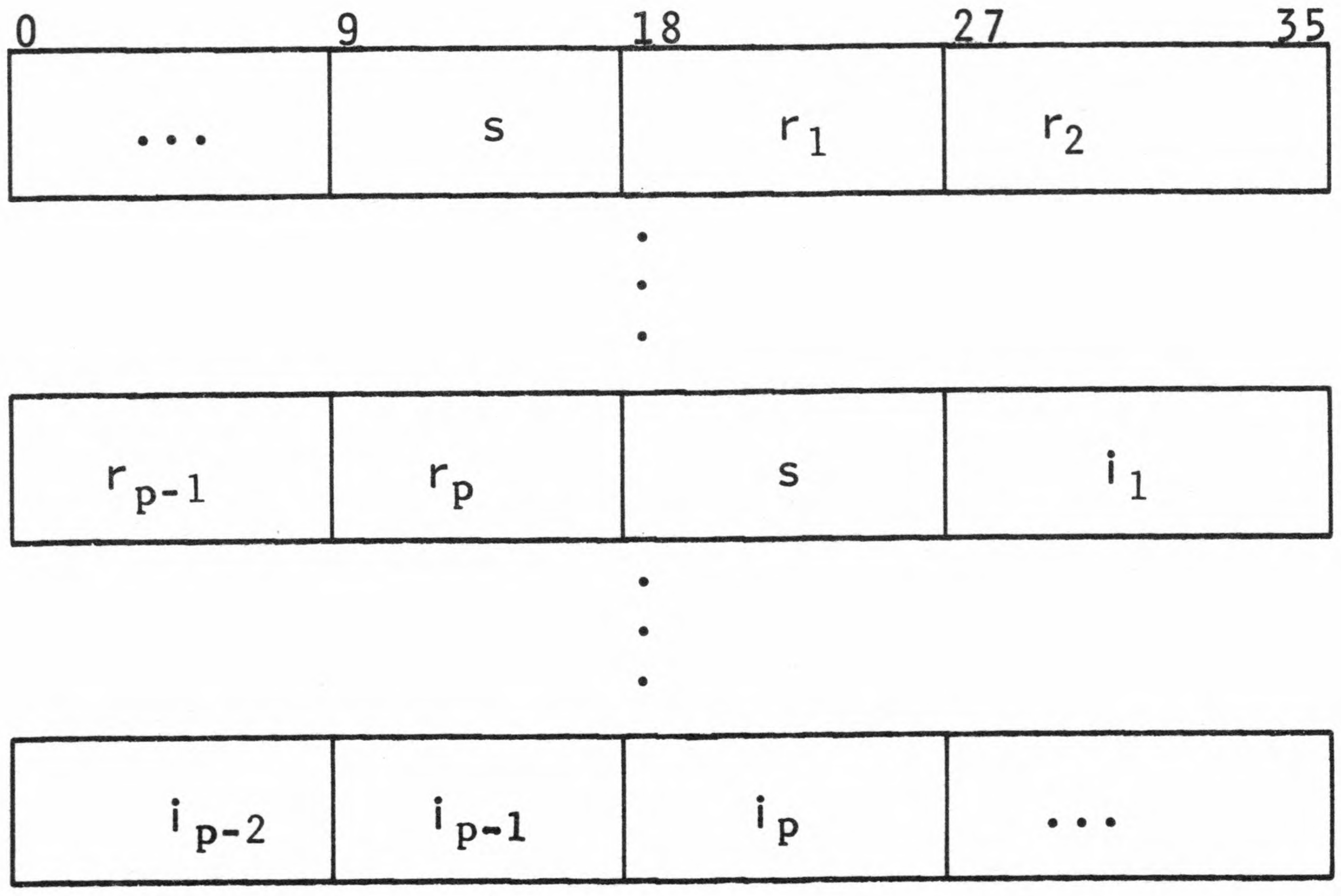
Complex Fixed-Point Decimal

A complex fixed-point decimal number of precision p is stored as a pair of real fixed-point decimal integers of precision p. The first integer, r, is the real part of the complex value and the second integer, i, is the imaginary part of the complex value.

If ALIGNED, the number is positioned at a word boundary and occupies an integral number of words, as follows:



If UNALIGNED within a structure, the number is positioned at a byte boundary and occupies 2(p+1) bytes, as follows:

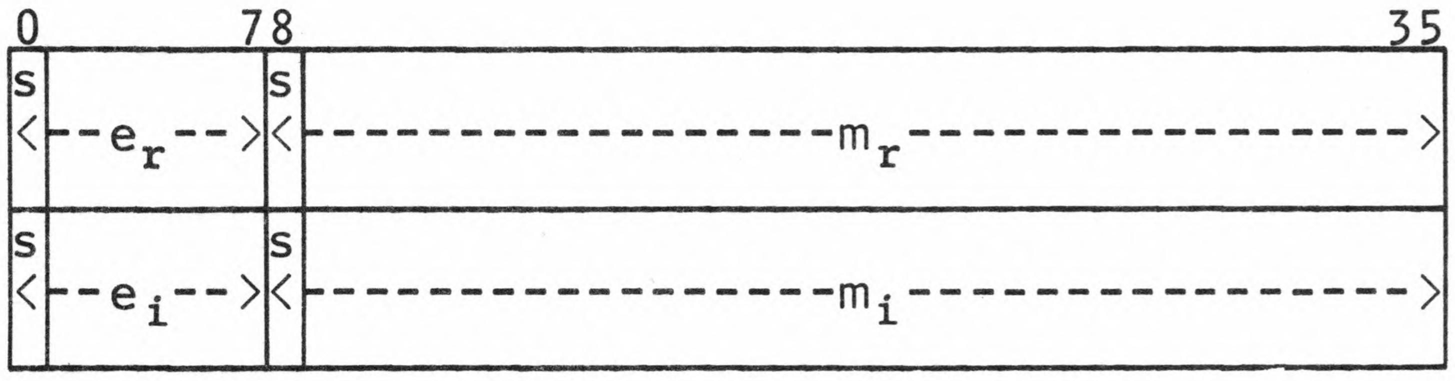


Complex Floating-Point Binary

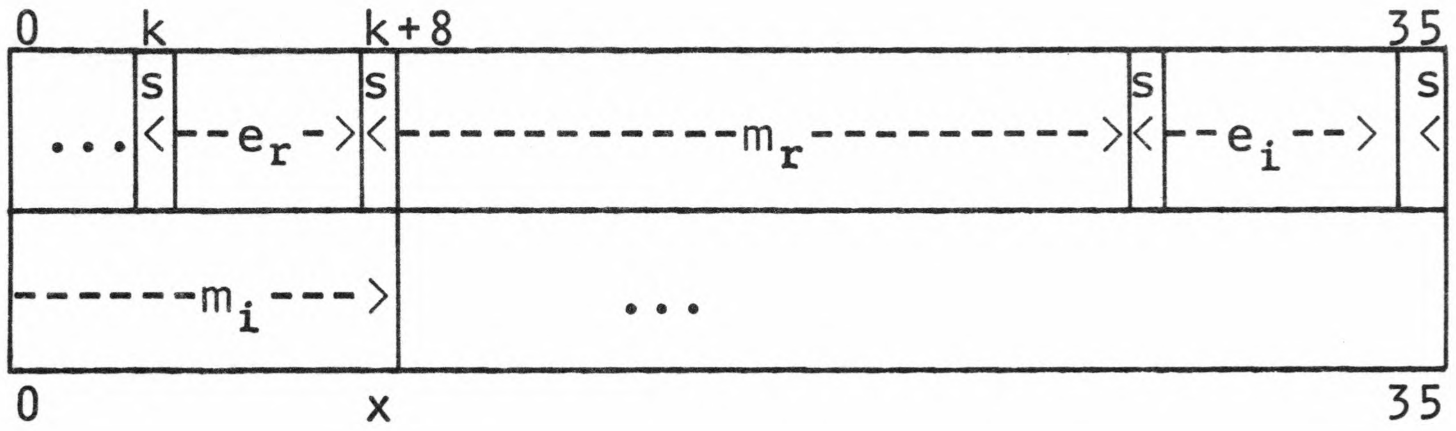
A complex floating-point binary number is stored as a pair of binary floating-point numbers. The first floating-point number, r, is the real part of the complex value and the second floating-point number, i, is the imaginary part of the complex value.

SINGLE PRECISION 0<p<28

If ALIGNED, the number is positioned at an even-word boundary and occupies two words, as follows:



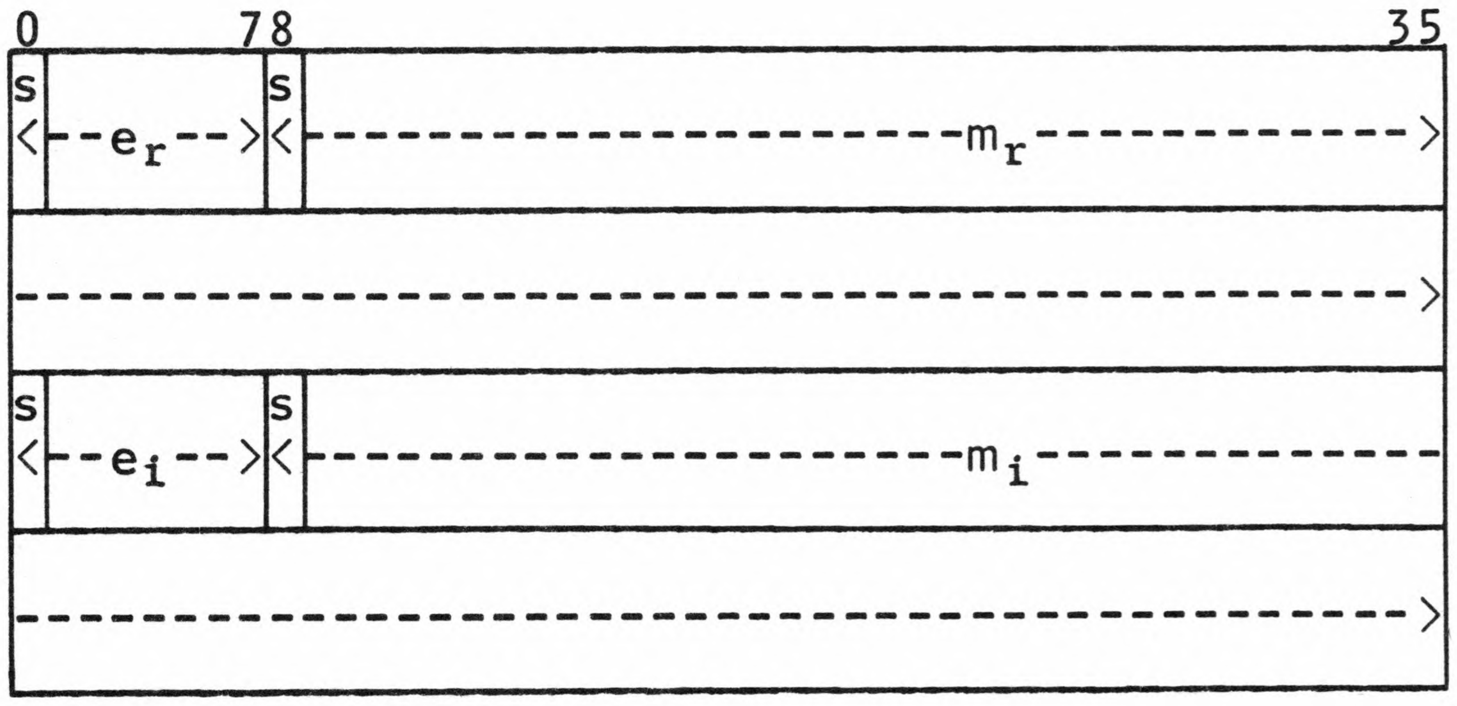
If UNALIGNED within a structure, the number is positioned at a bit boundary and occupies 2(p+9) bits, as follows:



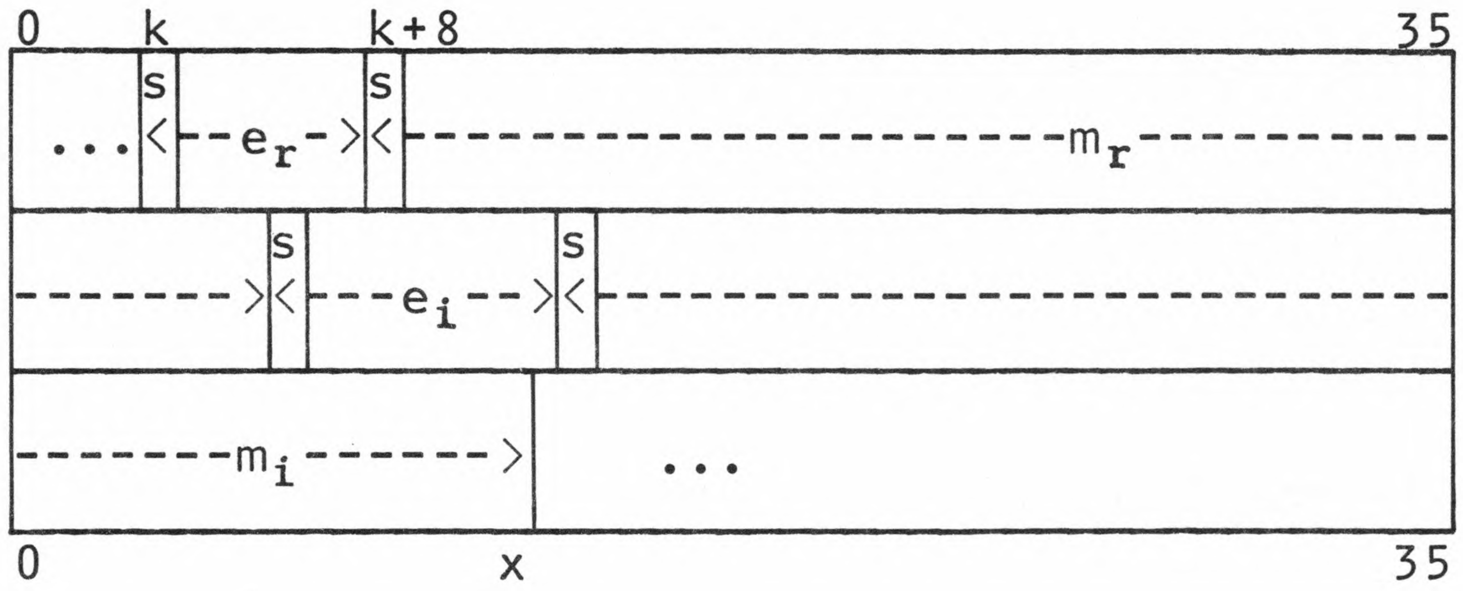
where x = MOD(k+2*p+17,36).

DOUBLE PRECISION 27<p<64

If ALIGNED, the number is positioned at an even-word boundary and occupies four words, as follows:



If UNALIGNED within a structure, the number is positioned at a bit boundary and occupies 2(p+9) bits, as follows:

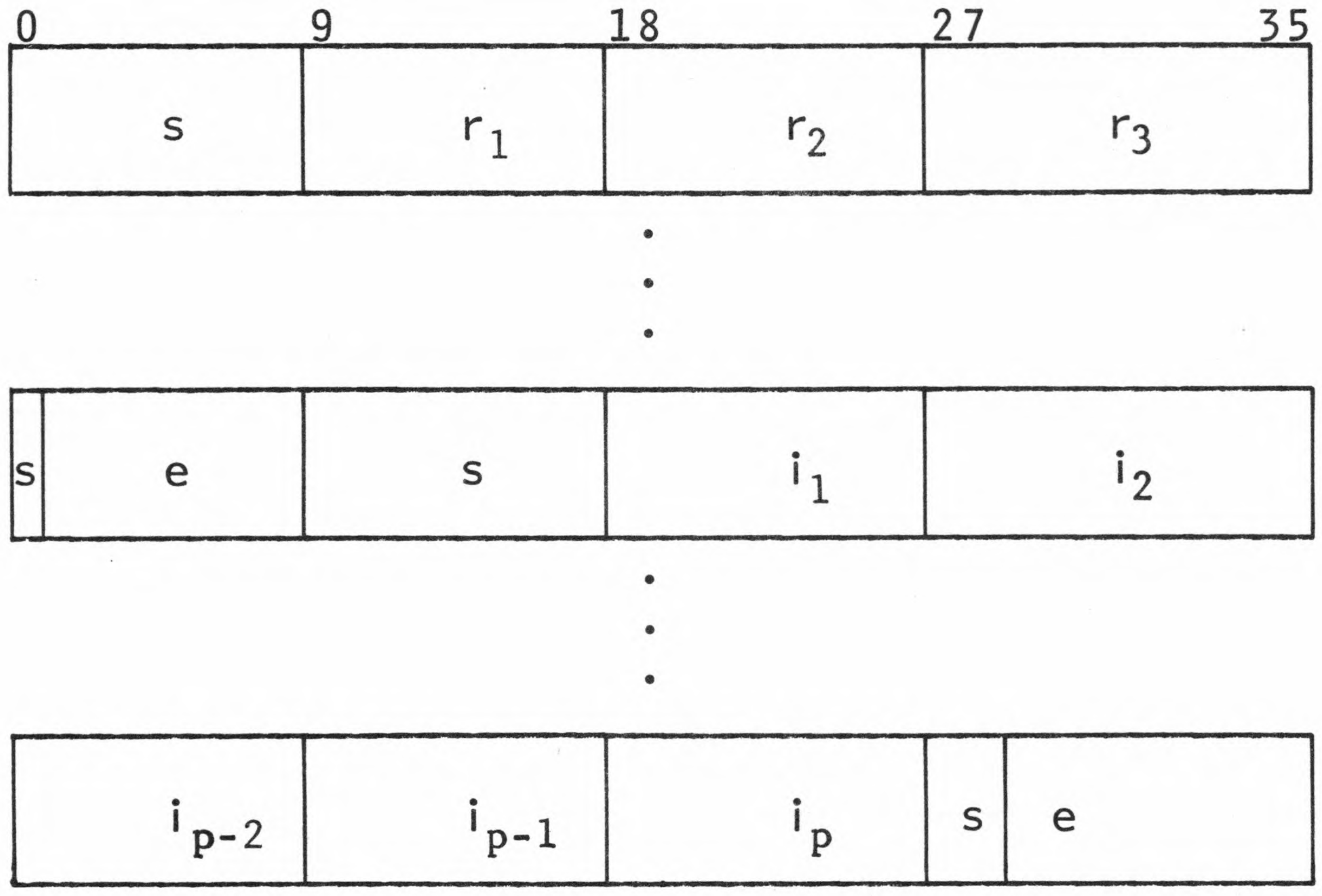


where x = MOD(k+2*p+17,36).

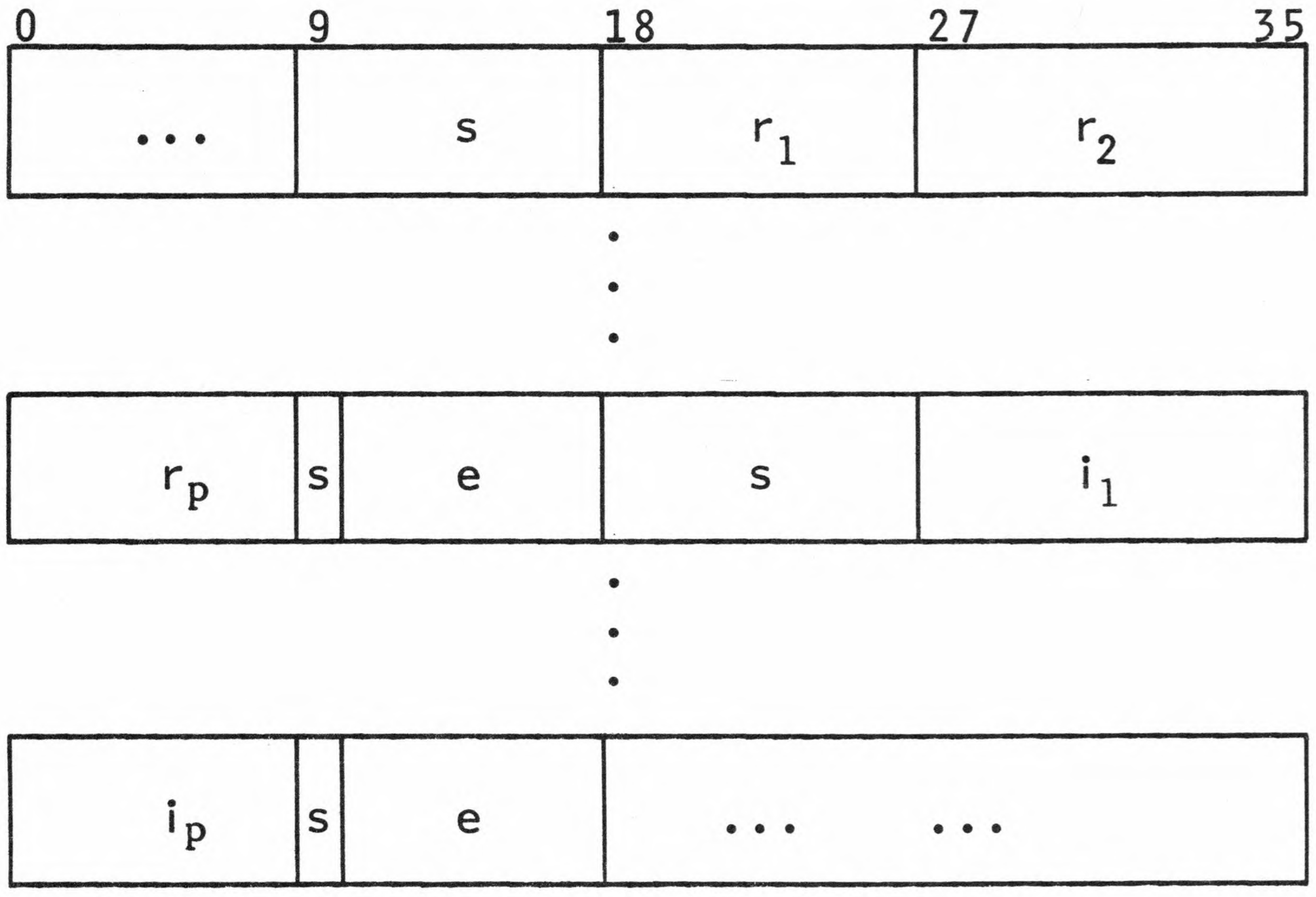
Complex Floating-Point Decimal

A complex floating-point decimal number of precision p is stored as a pair of real floating-point decimal numbers of precision p. The first number, r, is the real part of the complex value and the second number, i, is the imaginary part of the complex value.

If ALIGNED, the number is positioned at a word boundary and occupies an integral number of words, as follows:



If UNALIGNED within a structure, the number is positioned at a byte boundary and occupies 2(p+2) bytes, as follows:



STRING DATA TYPES

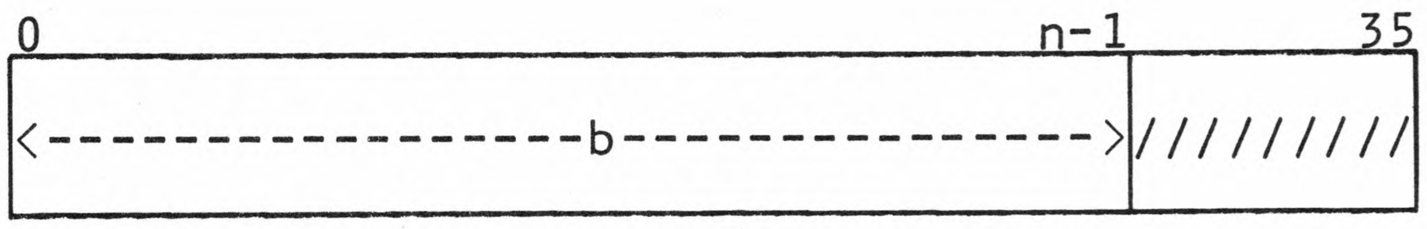
The string data types, their boundary requirements, and their default alignment are listed here:

Data Type	Boundary Required UNALIGNED	Boundary Required ALIGNED	Default Alignment
BIT	bit	word	UNALIGNED
CHARACTER	byte	word	UNALIGNED
PICTURE	byte	word	UNALIGNED
BIT VARYING	word	word	ALIGNED
CHARACTER VARYING	word	word	ALIGNED

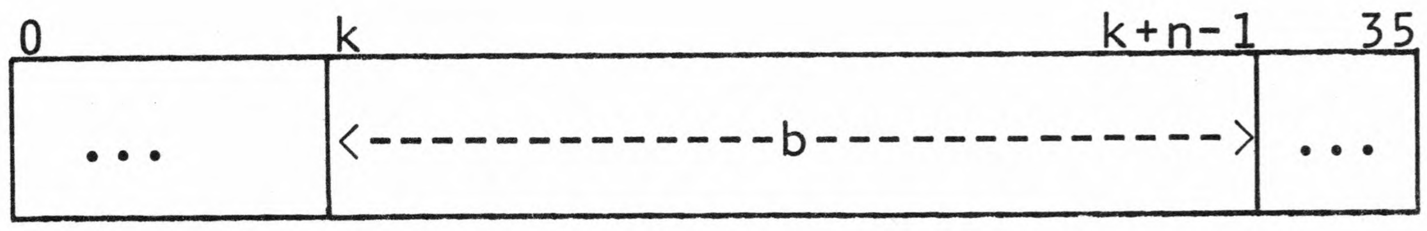
Bit-String

A bit-string of length n is stored as n consecutive bits.

If ALIGNED, the bit-string is positioned at a word boundary and occupies an integral number of words, as follows:



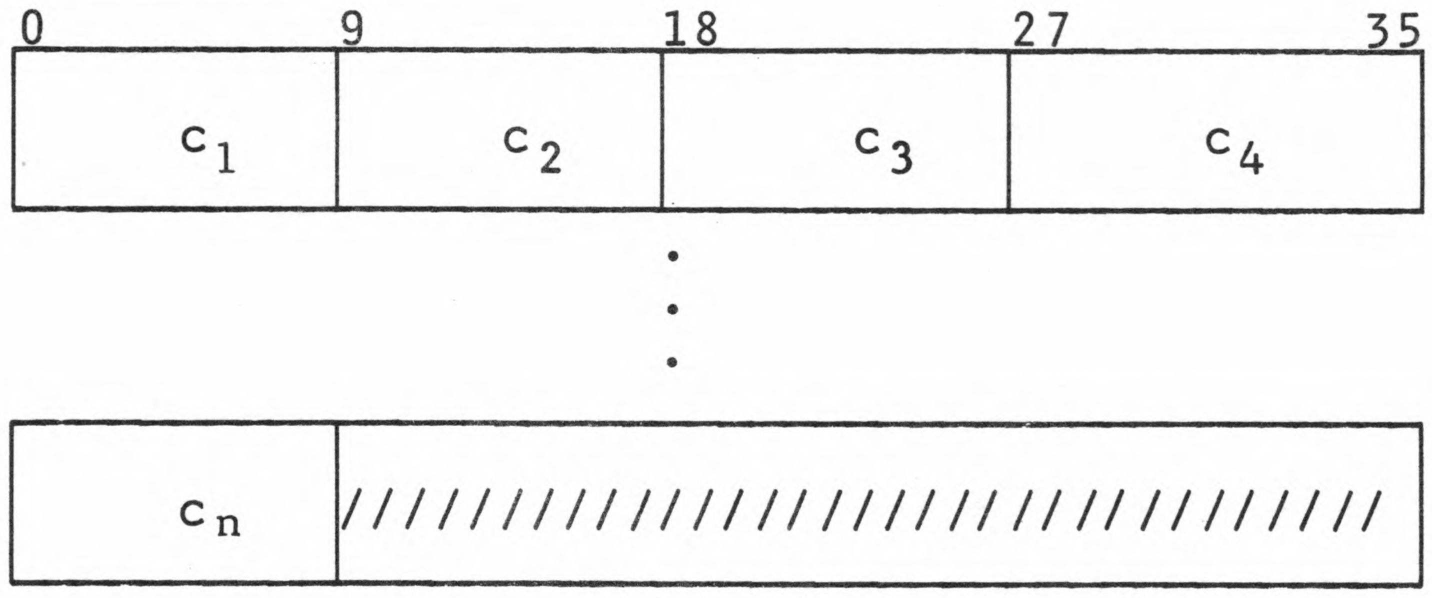
If UNALIGNED within a structure, the bit-string is positioned at a bit boundary and occupies n bits, as follows:



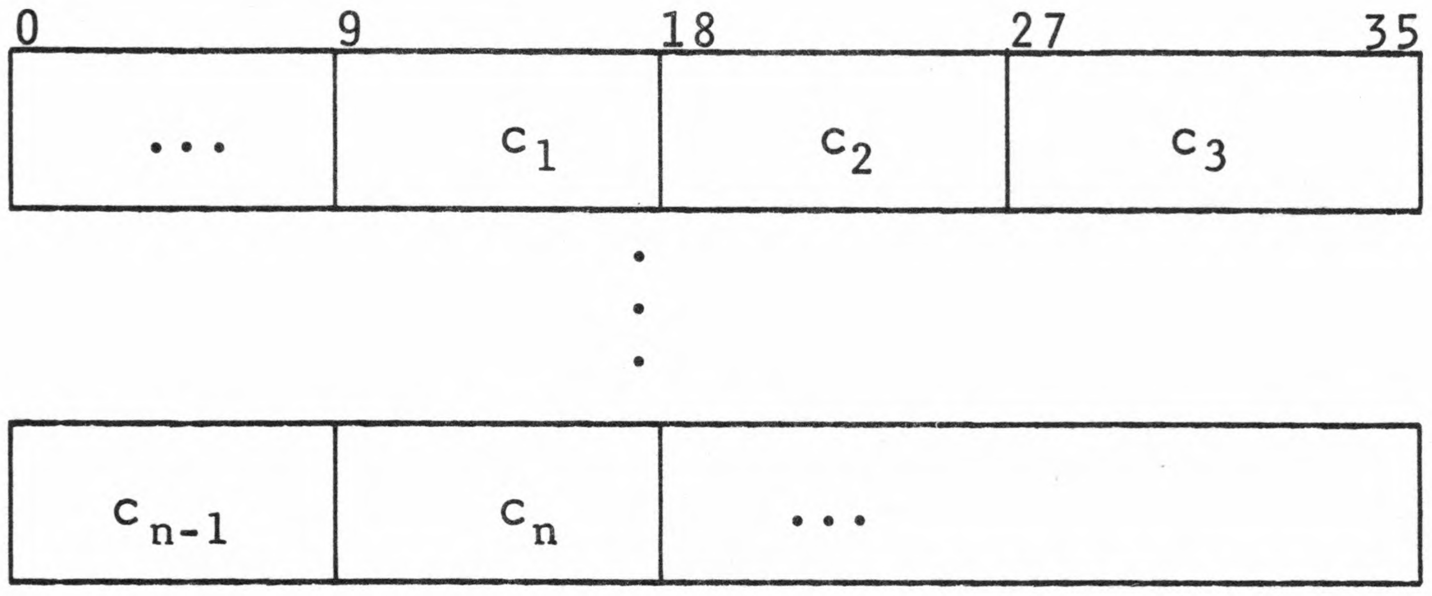
Character-String

A character-string of length n is stored as n consecutive bytes. Each byte contains a single 7-bit ASCII character right-justified within the byte. The two unused bits must be zero.

If ALIGNED, the character-string is positioned at a word boundary and occupies an integral number of words, as follows:



If UNALIGNED within a structure, the character-string is positioned at a byte boundary and occupies n bytes, as follows:



PICTURED CHARACTER-STRING

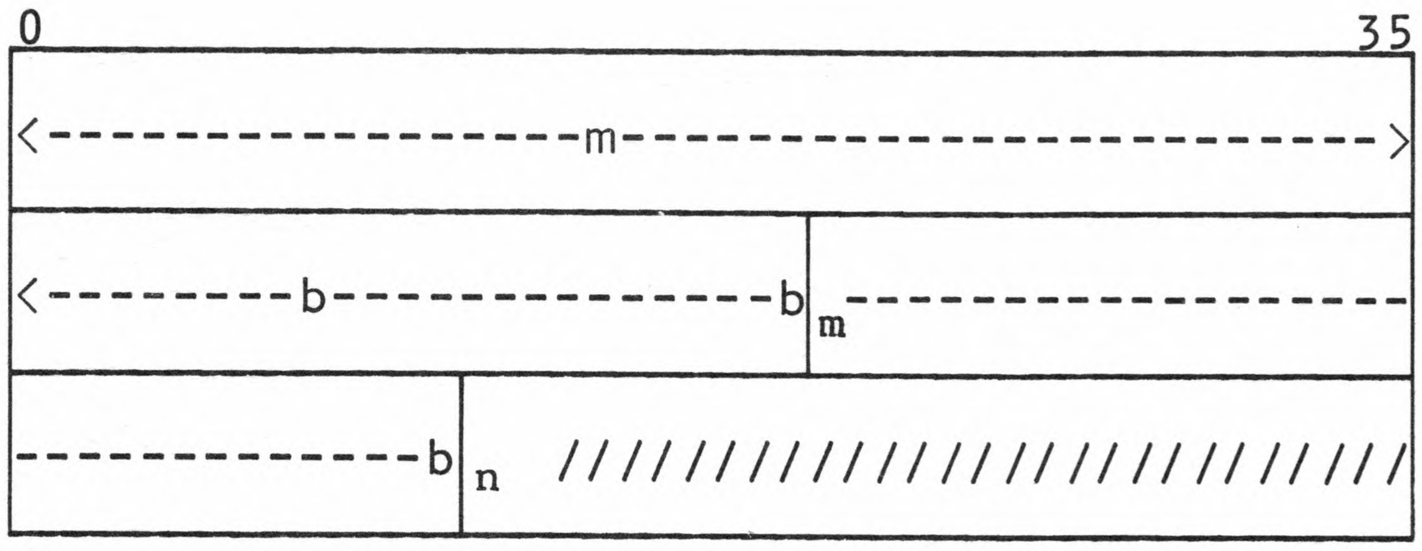
A pictured character-string is represented as a character string of length n, where n is the number of picture characters excluding the characters V and K and the scale factor indicator F(i). For example, the picture "999V9" is represented by a character string of length 4.

If ALIGNED, the pictured character-string is positioned at a word boundary and occupies an integral number of words.

If UNALIGNED within a structure, the pictured character-string is positioned at a byte boundary and occupies n bytes.

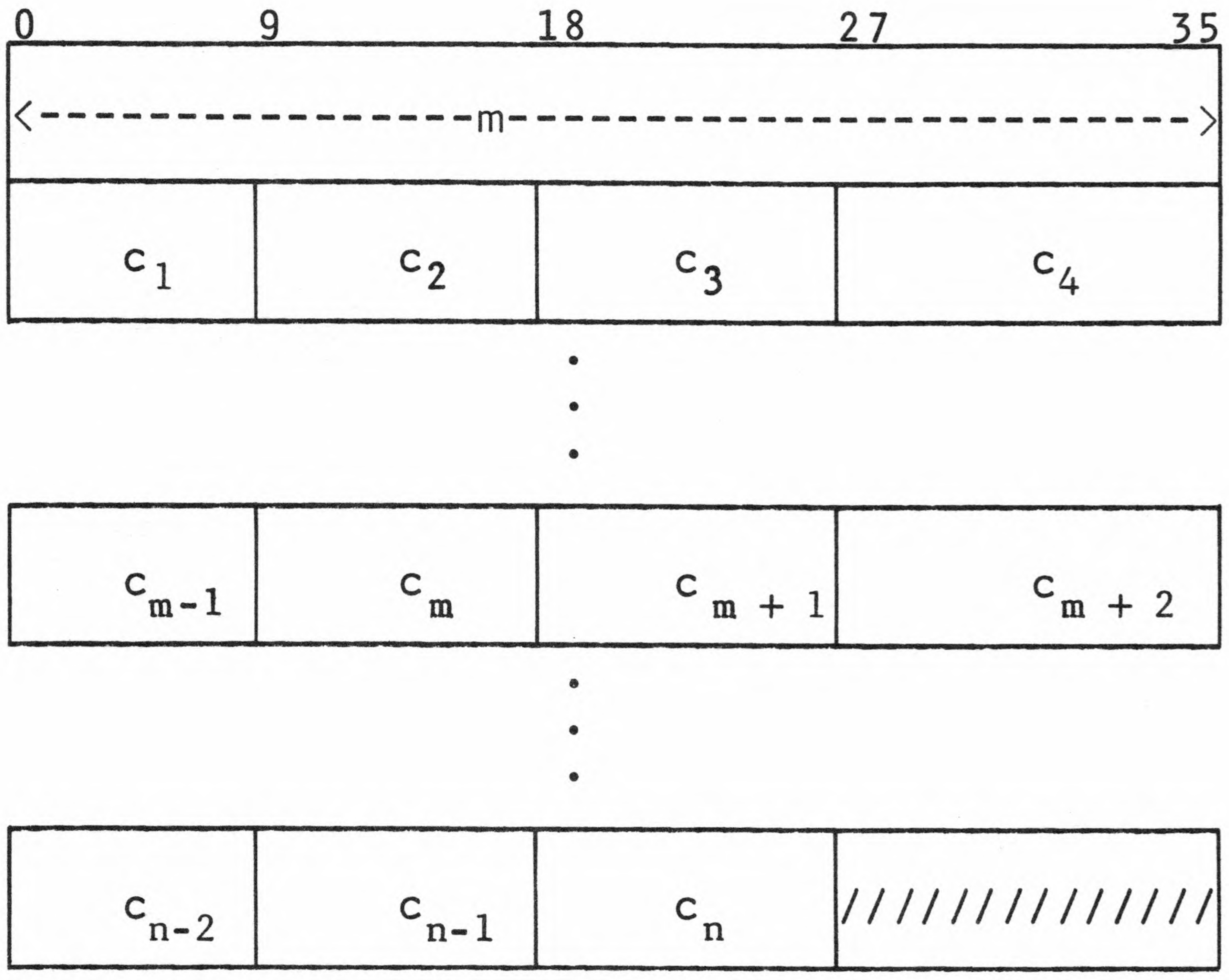
Varying Bit-String

The representation of a varying bit-string in storage is independent of its alignment. A varying bit-string of maximum length n is stored as an ALIGNED binary integer followed by an ALIGNED bit-string. The binary integer contains the current number of bits, m, as follows:



Varying Character-String

The representation of a varying character-string in storage is independent of its alignment. A varying character-string of maximum length n is represented by an ALIGNED binary integer followed by an ALIGNED non-varying character-string of length n. The binary integer contains the current number of characters in the string, as follows:



ADDRESS DATA TYPES

The address data types, their boundary requirements, and default alignments are given below:

Data Type	Boundary Required UNALIGNED	 ALIGNED	Default Alignment
LABEL	word	word	ALIGNED
ENTRY	word	word	ALIGNED
FORMAT	word	word	ALIGNED
POINTER	bit	word	ALIGNED
OFFSET	bit	word	ALIGNED
FILE	word	word	ALIGNED

Label, Entry, And Format

The label, entry, and format data type have the same internal representation. Each contains a pointer, p1, to a statement within a procedure and a pointer, p2, that identifies the stack frame for the most recent activation of the block immediately containing the statement located by p1. The UNALIGNED and ALIGNED representations are the same, namely:

```
0                  18                    35
 ________________________________________
|<-------p1-------->|<----------p2----------->|
|__________________|_____________________|
```

Pointer

A pointer consists of a word address and bit offset from the start of that word.

If ALIGNED, a pointer begins at a word boundary and occupies one word, as follows:

```
0                   18        24              35
|<----------w-------->|<---b--->|000000000000000|
```

If UNALIGNED within a structure, a pointer is positioned at a bit boundary and occupies 36 bits.

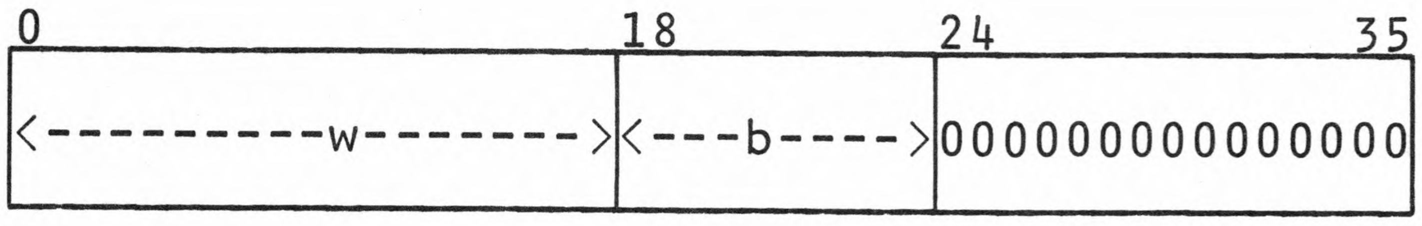
The null pattern for a pointer is:

```
0                  18        24              35
|1111111111111111111|0000000000|000000000000000|
```

Offset

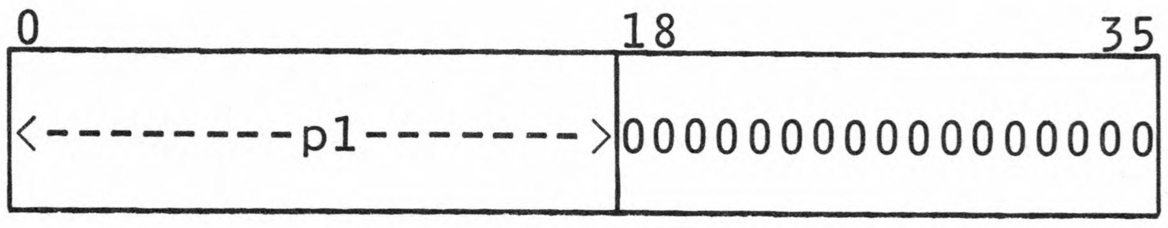
An offset contains a word offset, w, from the start of an area, and a bit offset, b, from the start of the word.

If UNALIGNED within a structure, an offset is positioned at a bit boundary and occupies 36 bits. If ALIGNED, an offset begins at a word boundary and occupies one word, as follows:



File

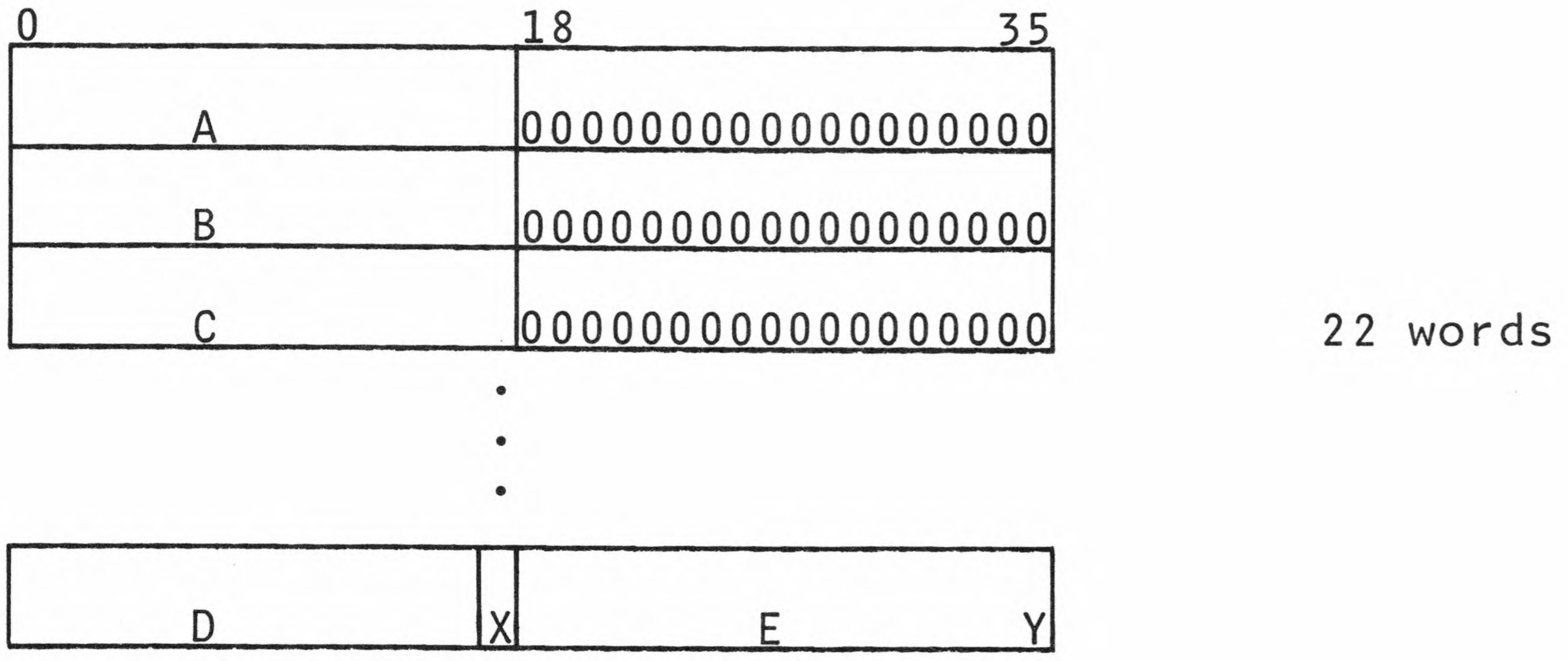
The UNALIGNED and ALIGNED representations for a file data type are the same. The representation consists of a full word whose upper half is a pointer, p1, to the file-state block, as follows:



AREA DATA TYPE

An area data type of size k is positioned at an even-word boundary and occupies k words.

An area is divided into a series of contiguous blocks by the allocation of based variables. The first block is the occupation record. The occupation record has the following format:



where A is the length of the occupation record.
B is the offset of the last word of the area.
C is the offset of the next available block space minus 1.
D is the size of the next contiguous block, if one has been allocated.
X is the busy indicator for the next contiguous block.
If X = 1, the next block is currently allocated.
E is the word offset from the beginning of the area of the next contiguous block
Y is the busy indicator for the current block.

The remaining words of the occupation record are used in the management of block space made available when based variables are freed in the area.

Before any based variables are allocated, the area is zero, except for C in the occupation record, which has the value k. When a based variable is allocated, a block is created in the area. In this way, blocks are allocated starting from the end of the occupation record and proceeding from low to high addresses until the end of the area is reached. Each block begins on an even word boundary and occupies an even number of words. If the based variable requires _n_ words, where _n_ is odd, the block consists of _n_+1 words. If _n_ is even, the block consists of _n_+2 words. The last word of the block is called the _block trailer_ word. The format of the block trailer word is illustrated in the diagram of the occupation record.

The number of words available for allocation in an area can be calculated, as follows:

- Before any based variables are allocated in an area of size k, the amount of block space _s_ is given, by the following formula:

  $$s = k - 22 - MOD(k,2)$$

  where MOD is the PL/I built-in function.

- After an initial sequence of n based variables has been allocated and none freed, the remaining available block space _rs_ is given by the following formula:

  $$rs = s - \sum_{i=1}^{n} (m_i + 1 + MOD(m_i + 1,2))$$

  where m_i is the number of words required by the _i_th based variable.

APPENDIX F

EXTERNAL NAMES

This appendix describes the conversion rule applied to external names and lists the external names reserved by the system.

CONVERSION RULE

Although in PL/I no restriction on the length of external names is imposed, the Loader restricts the length of external names to six or less characters. Also, the characters '$' and '_' cannot be part of the external name.

To convert a PL/I external name to an acceptable external name for the General Loader, the following rules are applied:

1. The characters '$' and '_' are converted to the character '.'. For example, 'AB_C' becomes '.A.B.C'.
2. If the external name contains more than six characters, it is converted to a six character name by concatenating three strings, as follows:

   s1!! s2!! s3

   where: s1 is the number of characters in the name mod 10. If the number of characters mod 10 is zero, then s1 = '.'.

   s2 is the first character of the name.

   s3 is composed of the last four characters of the name.

For example, consider the following external names and their converted names:

External Name	Converted External Name
ABCDEFG	7ADEFG
ABCDEFG__HI	.AG.HI
H25MODEL1	9HDEL1
H26MODEL1	9HDEL1

Note that the last two <u>different</u> external names are converted to the <u>same</u> six character name for the Loader. In order to be unique, names of the same length must differ in either the first character or one of the last four characters. Similarly, names that are distinguished from each other only by the characters '$' and '_' are converted to the same external name, as follows:

External Name	Converted External Name
$ABC	.ABC
_ABC	.ABC
H25MODEL$	9HDEL.
H26MODEL_	9HDEL.

In addition to being distinct from one another, external names must be unique with respect to reserved system names. A list of the external names reserved by the system is given in the next section of this appendix. To assure uniqueness, an external name must be converted and compared against the list of system reserved names. For example, the external name 'LONGLENGTH' cannot be used because it is converted to the name '.LNGTH', which appears on the list of reserved names.

RESERVED EXTERNAL NAMES

Table F-1 lists the external names that are reserved by the system. This list may change slightly depending on the update level of future software releases. It is primarily composed of entry names to execution-time support modules in the PL/I library file. A current list of that file, and other libraries, may be generated by use of the cross reference service program .SREF. For instructions on use of that program, see the Service Routines manual.

Table F-1. Reserved External Names

...F..	...L..	...N..	...S..	...W..
..ABOR	..ABT	..ABT1	..ALOC	..AND1
..CCLS	..CMB1	..CMC2	..CONV	..COPN
..CREA	..CRWT	..CWRT	..EXR1	..GBO
..GDB.	..IABT	..ICLS	..IDEL	..IOPN
..IREA	..IRVT	..IRWT	..IWRT	..LOOK
..MB1	..MC1	..MN1	..OR1	..RABT
..RALC	..RBAC	..RCLS	..REVT	..ROPN
..RREA	..RRVT	..RRWT	..RWRT	..SBA
..SCA	..SFCB	..SGET	..SPUT	..SS0
..ZRIT	.71B25	.A1	.A2	.A3
.ALLA	.ALLCA	.ALLOC	.BCS1	.BITOP
.BLANK	.BMUU	.BTOP	.CALSG	.CAT
.CMUU	.COMIO	.COUNT	.DBLTP	.DESC1
.DTTLY	.EMCP	.ERFC.	.ESW	.EXCG.
.EXTSZ	.FREE	.GETAQ	.GLGE	.INTET
.IOFLG	.L001.	.L002.	.L003.	.L005.
.L007.	.L033.	.L039.	.L040.	.L041.
.L042.	.L043.	.L045.	.L046.	.L047.
.L048.	.L049.	.L050.	.L051.	.L052.
.L053.	.L054.	.L055.	.L056.	.L058.
.L059.	.L061.	.L063.	.L065.	.L066.
.L067.	.L068.	.L069.	.L070.	.L072.
.L073.	.L074.	.L076.	.L078.	.L080.

Table F-1 (cont). Reserved External Names

.L082.	.L083.	.LG1	.LG2	.LNGTH
.LOG2.	.LONE.	.LV	.MCP	.MEIS.
.MVRET	.N	.NMORY	.NTEMP	.NUM
.OHAR.	.OIELD	.OILE.	.OLLG1	.ONCOD
.ONES	.P.DEL	.P.LOC	.P.MSG	.P.REA
.P.REW	.P.WRI	.P0000	.P0001	.P0002
.P0003	.P0004	.P0005	.P0006	.P0007
.P0008	.P0009	.P0010	.P0011	.P0013
.P0014	.P0015	.P0016	.P0017	.P0018
.P0019	.P0020	.P0021	.P0022	.P0023
.P0024	.P0030	.P0031	.P0032	.P0033
.P0034	.P0035	.P0036	.P0037	.P0038
.P0039	.P0040	.P0041	.P0042	.P0043
.P0044	.P0045	.P0046	.P0047	.P0048
.P0049	.P0050	.P0051	.P0052	.P0053
.P0054	.P0055	.P0058	.P0059	.P0060
.P0061	.P0062	.P0063	.P0064	.P0065
.P0066	.P0067	.P0068	.P0069	.P0070
.P0071	.P0072	.P0073	.P0074	.P0075
.P0076	.P0077	.P0078	.P0079	.P0080
.P0081	.P0082	.P0083	.P0084	.P0085
.P0086	.P0087	.P0088	.P0089	.P0090
.P0091	.P0093	.P0094	.P0095	.P0096
.P0097	.P0098	.P0099	.P0100	.P0101
.P0102	.P0103	.P0104	.P0105	.P0106
.P0107	.P0108	.P0109	.P0110	.P0111
.P0112	.P0113	.P0114	.P0115	.P0116
.P0117	.P0118	.P0119	.P0120	.P0121
.P0122	.P0123	.P0124	.P0125	.P0126
.P0127	.P0128	.P0129	.P0130	.P0131
.P0132	.P0133	.P0134	.P0135	.P0136
.P0137	.P0138	.P0139	.P0140	.P0141
.P0142	.P0143	.P0144	.P0145	.P0146
.P0147	.P0148	.P0149	.P0150	.P0151
.P0152	.P0153	.P0154	.P0155	.P0157
.P0158	.P0159	.P0160	.P0161	.P0162
.P0163	.P0164	.P0165	.P0166	.P0167
.P0168	.P0169	.P0170	.P0171	.P0172
.P0173	.P0174	.P0175	.P0176	.P0177
.P0178	.P0179	.P0180	.P0181	.P0182
.P0183	.P0184	.P0185	.P0186	.P0187
.P0188	.P0189	.P0190	.P0191	.P0192
.P0193	.P0194	.P0195	.P0196	.P0197
.P0198	.P0199	.P0200	.P0201	.P0202
.P0203	.P0204	.P0205	.P0206	.P0207
.P0208	.P0209	.P0210	.P0211	.P0212
.P0213	.P0214	.P0215	.P0216	.P0217
.P0218	.P0219	.P0220	.P0221	.P0222
.P0223	.P0224	.P0225	.P0226	.P0227
.P0228	.P0229	.P0230	.P0231	.P0232
.P0233	.P0234	.P0235	.P0236	.P0237
.P0238	.P0239	.P0240	.P0241	.P0242
.P0243	.P0244	.P0245	.P0246	.P0247
.P0248	.P0249	.P0250	.P0251	.P0252
.P0253	.P0254	.P0255	.P0257	.P0258
.P0259	.P0260	.P0261	.P0262	.P0263
.P0264	.P0265	.P0266	.P0267	.P0268
.P0269	.P0270	.P0271	.P0272	.P0273
.P0274	.P0275	.P0276	.P0278	.P0279
.P0280	.P0281	.P0282	.P0283	.P0284

.P0285	.P0286	.P0287	.P0288	.P0289
.P0290	.P0291	.P0292	.P0293	.P0294
.P0295	.P0296	.P0297	.P0298	.P0299
.P0300	.P0301	.P0302	.P0303	.P0304
.P0305	.P0306	.P0307	.P0308	.P0309
.P0310	.P0315	.P0316	.P0317	.P0318
.P0319	.P0320	.P0321	.P0322	.P0323
.P0324	.P0325	.P0326	.P0327	.P0328
.P0329	.P0330	.P0331	.P0332	.P0333
.P0334	.P0335	.P0336	.P0337	.P0338
.P0339	.P0340	.P0341	.P0342	.P0343
.P0344	.P0345	.P0346	.P0347	.P0348
.P0349	.P0350	.P0351	.P0352	.P0353
.P0354	.P0356	.P0357	.P0358	.P0359
.P0360	.P0361	.P0362	.P0363	.P0364
.P0365	.P0366	.P0367	.P0368	.P0369
.P0370	.P0371	.P0372	.P0373	.P0374
.P0375	.P0376	.P0377	.P0378	.P0379
.P0380	.P0381	.P0382	.P0383	.P0384
.P0385	.P0386	.P0387	.P0388	.P0389
.P0390	.P0391	.P0392	.P0393	.P0400
.P0401	.P0402	.P0403	.P0407	.P0408
.P0409	.P0410	.P0417	.P0418	.P0419
.P0423	.P0424	.P0425	.P0426	.P0428
.PADIT	.PAGES	.PASCB	.PBACK	.PBCDA
.PCTN.	.PDESC	.PEMP.	.PFLTG	.PGDL
.PILOP	.PL001	.PL004	.PL005	.PL006
.PL007	.PL008	.PL009	.PL010	.PL011
.PL012	.PL013	.PL014	.PL015	.PL016
.PL017	.PL018	.PL019	.PL020	.PL021
.PL022	.PL023	.PL024	.PL025	.PL026
.PL027	.PL028	.PL029	.PL030	.PL031
.PL032	.PL033	.PL034	.PL035	.PL036
.PL037	.PL038	.PL039	.PL040	.PL041
.PL042	.PL043	.PLABT	.PLMES	.PMEMF
.PNTB.	.PNTC.	.PNTN.	.PNTRM	.POVFL
.PR8EC	.PROPN	.PROR.	.PRPUT	.PSETU
.PUND.	.PWRPU	.PZDIV	.QMASK	.RABT
.RBUF1	.RBUF2	.RCLSE	.RDAND	.RDCP2
.RDCPY	.RDMOV	.RDOPR	.RDOR	.RDXOR
.RECIO	.REM1	.REM2	.RGET	.ROPEN
.RPDPT	.RPUT	.SBTTP	.SCTLY	.SETU.
.SGJIN	.SIND.	.SOSD.	.SREGS	.STKHD
.STKTL	.STORE	.STR1	.STR2	.SX01
.SX23	.SX45	.SX67	.TAND.	.TEMP
.TEMP2	.TMRAQ	.TMPBP	.TMPPT	.TMPSZ
.TSX45	.XR2	1ACOS.	1AEDEF	1ATAN.
1CCOS.	1CTAN.	1D.OP.	1DCOS.	1FREE.
1GCARD	1LANH.	1LG10.	10.TOP	1OURCE
1PEAD.	1PG2EE	1PLATE	1PNAL.	1PNTRY
1PP2EE	1SECK.	1SOSH.	2AAND.	2AIGN.
2AIND.	2AOSD.	2ATAN.	2BHAR.	2CANH.
2CBIT.	2CINH.	2COSH.	2DAND.	2DIND.
2DOG2.	2DONE.	2DOSD.	2DRFC.	2OINFO
2ORCE.	2PFSBP	2PITE.	2PLIO.	2PNPUT
2POSE.	2PR30S	2PREAD	3ABIT.	3AOSH.
3BITH.	3CCOS.	3DANH.	3DG10.	3DOSH.
3DORT.	3DTAN.	3FIDE.	3ONKEY	3ONLOC
3PATE.	3PETE.	3PP1DA	3PR30I	3PR4CS

Table F-1 (cont). Reserved External Names

3PR5RI	3PROR.	3SADDR	3STOR.	4AHAR.
4AND2.	4BLOAD	4CANH.	4CINH.	4CITH.
4COSH.	4DANH.	4DINH.	4DOSH.	4DTAN.
4OCODE	4DFILE	4OTTOM	4PG1DD	4PITE.
4PNTER	4PONV.	4PP1DD	4PP1DO	4PR3OR
4PR4CI	4PR5RR	4PR6WI	4PRACE	5AAGE.
5AITH.	5DCOS.	5DOSH.	5P..R.	5P..S.
5PBIT.	5PEXP.	5PLEX.	5PONV.	5PR4CR
5PR6WR	5PR7LI	5PR9DI	5PSET.	6DANH.
6DINGS	6DINH.	6DND2.	6DOSH.	6IURCE
6OURCE	6PALL.	6PDESC	6PDIT.	6PG1SS
6PHAR.	6PP1SS	6PR7LR	6PR8EI	6PR9DR
6PUND.	6RGGER	6SE.NO	6SMBOL	6SRESS
7DAN2.	7DINH.	7DORT.	7DSIN.	7DTAN.
7ERFC.	7CIELD	7PDLER	7PEDIT	7PGENO
7PHAR.	7PNENO	7PR3OC	7PR5RC	7PR8ER
7PS.L.	7PS.R.	7SALUE	7SGNAL	7SION.
7SLER.	7SNAL.	7UHECK	8C.OP.	8DRFC.
8EEXEC	8ONKEY	8ONLOC	8PENO.	8PEXT.
8PHASE	8PNAME	8PR4CC	8PR6WC	8PURE.
8SENCE	8SRINT	8SRMAT	8UCKER	9BABLE
9C.OP.	9GDATE	9GTIME	9MDESC	9OCHAR
9OCODE	9ONDEX	9PATE.	9PDUMP	9PGNAL
9PIME.	9PR.L.	9PR7LC	9PR9DC	9PSTAT
9SCOS.	9SINT#	9TS.2.	9TS.3.	ADEXP.
ALLOC.	AREA.	ASIN.	ASINH.	ATAN2.
CABS.	CASIN.	CATAN.	CEXP.	CFDP.
CFMP.	CLOG.	CSIN.	CSQRT.	CXP1.
CXP2.	DASTN.	DCABS.	DCEXP.	DCFDP.
DCFMP.	DCLOG.	DCSIN.	DCXP1.	DCXP2.
DERF.	DEXP.	DIEXP.	DLOG.	DMOD.
DND	DSIN.	DSINH.	DSQRT.	DTAN.
DTANH.	DXP1.	DXP2.	ERF.	EXP.
EXPON.	FREE.	FREEN.	IEXP.	LBSWRK
LOG.	LSTKBT	PACKER	PENTRY	PIDLNK
PLINK	PLLINK	POP.	PUSH.	RCLOSE
RGET	ROPEN	RPUT	SIN.	SINH.
SNAP.	SORT.	SYSIN	SYSIN#	TAN.
TANH.	XP2.	XP3.	ZAGPRT	ZASCEL
ZASCGE	ZGEASC	ZWATCH	ZWTEND	

APPENDIX G

STRUCTURE OF THE INCLUDE FILE

This appendix describes the structure of the INCLUDE file. Five figures are included. The first figure illustrates the general structure of the INCLUDE file; and the remaining figures illustrate the detailed structure of the components.

INCLUDE FILE

The INCLUDE file is structured as a random ASCII file divided into 320 word blocks. The file consists of catalog blocks and data blocks. The catalog blocks are linked together. Following each catalog block are the data blocks that contain the macro text for the macro names in the catalog block. The format of the catalog and data blocks are given later in this section. The general structure of the INCLUDE file is illustrated in Figure G-1.

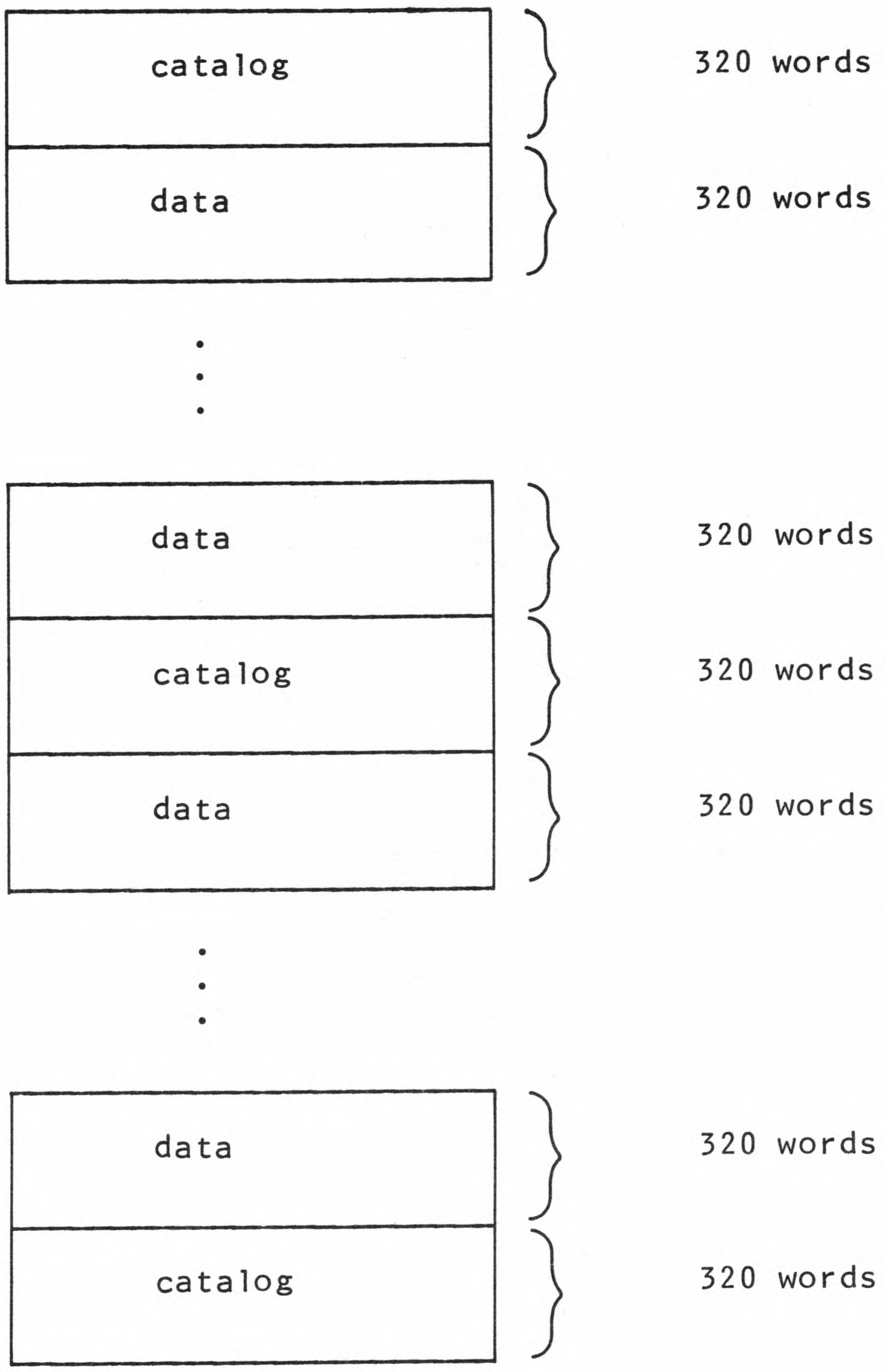


Figure G-1. Structure of the INCLUDE File.

Catalog Blocks

A catalog block of an INCLUDE file contains a header, control information, and up to 15 macro identifiers.

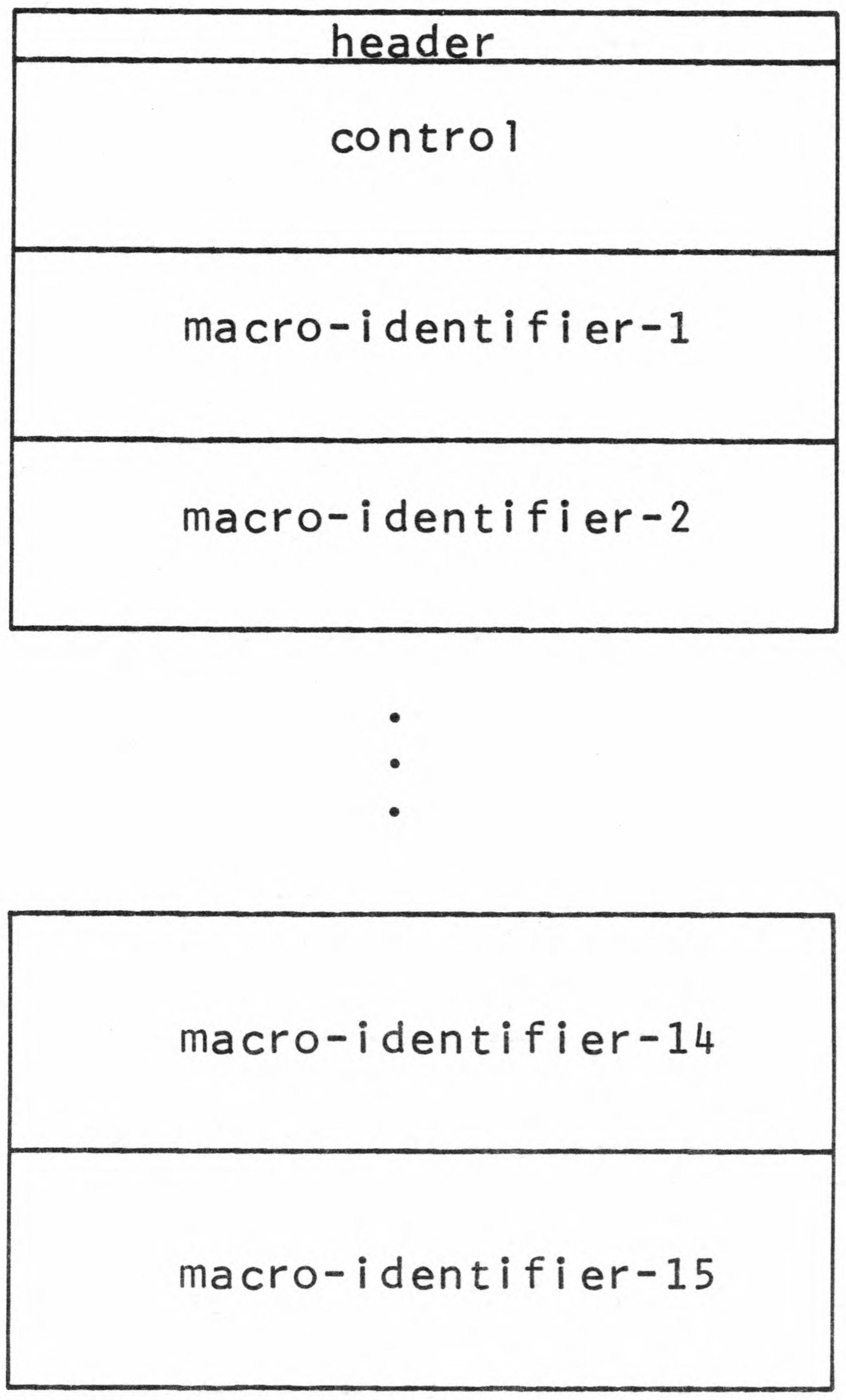


Figure G-2. Structure of a Catalog Block

The header contains the ASCII string "*SRCLIB*".

The control portion contains statistics about the macros in the file. The contents of the control portion are described in detail in the next section of this appendix.

The macro-identifier contains the text-name associated with the macro and a pointer to the text. A detailed description of the macro-identifier is given later in this appendix.

CONTROL PORTION OF THE CATALOG

The control portion of the catalog linkage and summary information is as follows.

0	NE	NDE
1	LGB	FGB
2	NAVB	NVB
3	PC	NC
4	▨	
5	▨	
6	▨	
7	▨	

where: NE is the number of macros registered.

NDE is the number of macros deleted.

LGB is the relative block address of the last macro deleted.

FGB is the relative block address of the first macro deleted.

NAVB is the number of entries available in the catalog.

NVB is the relative block address of the first unused entry within this catalog.

PC is the relative block address of the preceding catalog.

NC is the relative block address of the next catalog.

Each macro-identifier in the catalog portion contains the ASCII characters of the text-name associated with the macro and pointers to the text, as follows.

0	C_1	C_2	C_3	C_4	text-name
1	C_5	...			
2					
3					
4					
5					
6					
7					
10	LDB	FDB			
11	D	NB	NR		
12	not used at present				
.					
.					
19	reserved for expansion				

where: C_i are the ASCII characters of the text-name associated with the macro. The name can consist of up to 32 characters (left justified, space filled).

LDB is the relative block address of the last data block containing text associated with this text-name.

FDB is the relative block address of the first data block containing text associated with this text-name.

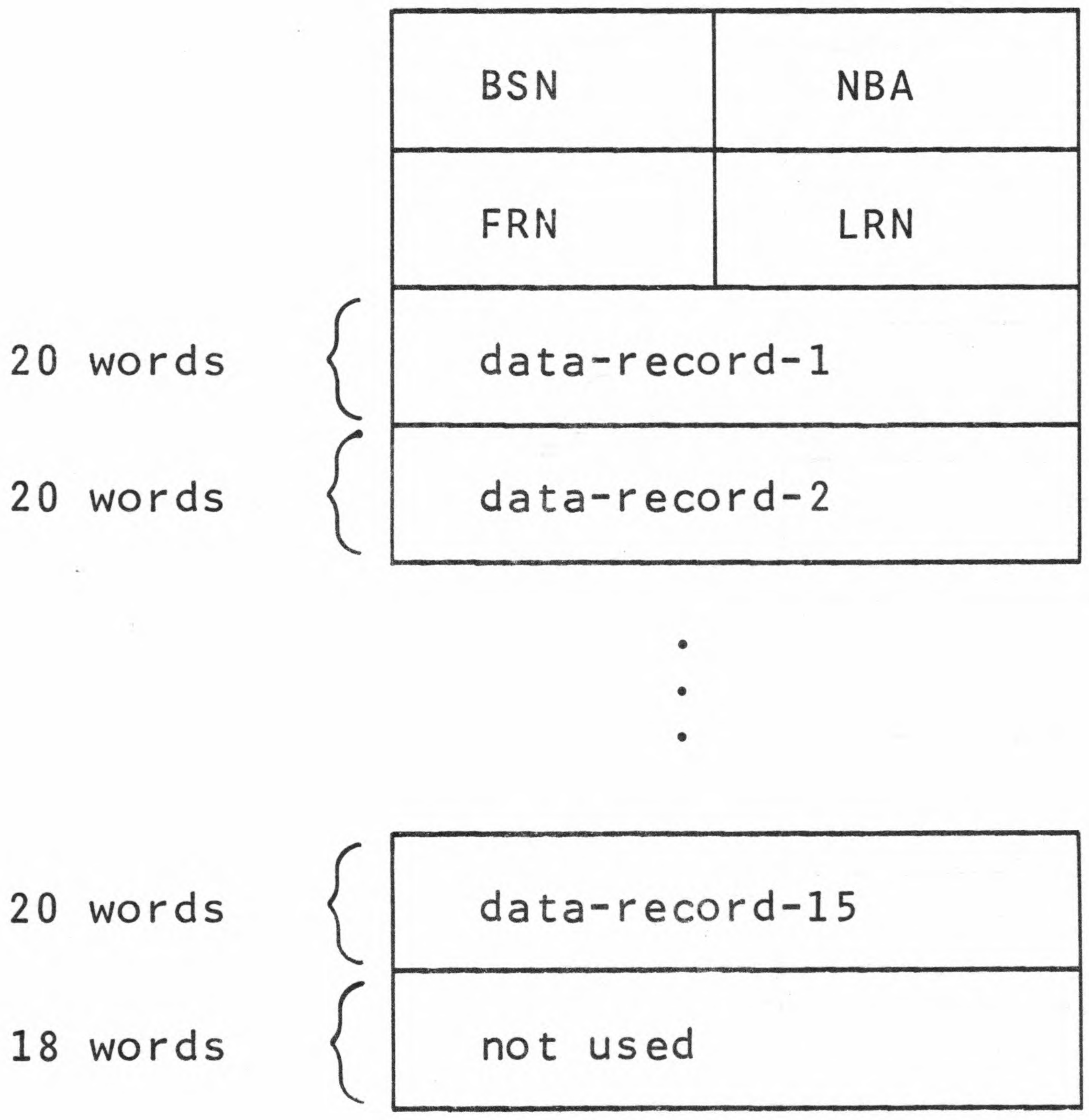
D is the deletion indicator.

NB is the number of blocks occupied by the text associated with this text-name.

NR is the number of records required for the representation of the text associated with this text-name.

Data Blocks

A data block contains two words of identification, followed by a series of fixed-length data-records. The text for a macro occupies one or more data-records and can occupy more than one data block. The structure of the data block is given here.



where: BSN is the logical serial number of the block in this macro text.

NBA is the relative block address of the next data block.

FRN is the serial number of the first data-record of the block relative to this macro.

LRN is the serial number of the last data-record of the block relative to this macro.

APPENDIX H

GCOS PL/I COMPILER ERROR MESSAGES

This appendix contains the error messages that can be produced by the GCOS PL/I compiler in the compilation of a source program. When the compiler detects an error, it writes the error message, as it appears in this appendix, on the error message listing. The action then taken by the compiler depends upon the severity of the error, as follows:

Severity	Action
WARNING	The program compilation continues.
SEVERITY 2	The error is corrected and the compilation continues.
SEVERITY 3	The compilation continues from the next logical starting point, but code generation is suspended.
FATAL	The compilation is terminated.

The symbol '$' in the error messages listed here is replaced by an identifying name when the PL/I compiler detects an error and prints a message.

```
ERROR 1, SEVERITY 3
SYNTAX ERROR HAS BEEN FOUND IN THIS STATEMENT. PROCESSING OF THE STATEMENT HAS BEEN TERMINATED.

ERROR 2, SEVERITY 3
EXCESS LEFT PARENTHESES HAVE BEEN FOUND IN THIS STATEMENT. PROCESSING OF THE STATEMENT HAS BEFN TERMINATED.

ERROR 3, SEVERITY 3
A SYNTAX ERROR HAS BEEN FOUND IN THIS DECLARE STATEMENT. PROCESSING WILL CONTINUE WITH WHAT APPEARS TO BE THE NEXT
DECLARATION.

ERROR 4, SEVERITY 3
EXCESS RIGHT PARENTHESES HAVE BEEN FOUND IN THIS STATEMENT.

ERROR 5, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE NUMBER OF REFERENCES ON THE LEFT SIDE OF AN ASSIGNMENT OPERATOR CANNOT EXCEED 128.

WARNING 6
THE PARENTHESIZED LIST FOLLOWING THE LABEL ATTRIBUTE IN THE DECLARATION OF $ IS NOT STANDARD PL/I AND HAS BEEN IGNORED.
USE THE "LOCAL" ATTRIBUTE TO INDICATE THAT A LABEL VARIABLE HAS VALUES LOCAL TO THE BLOCK IN WHICH IT IS DECLARED.

ERROR 7, SEVERITY 3
AN UNRECOGNIZABLE ATTRIBUTE HAS BEEN FOUND IN THE DECLARATION OF $.

ERROR 8, SEVERITY 3
TEMPORARY RESTRICTION: THIS VERSION OF THE COMPILER DOES NOT SUPPORT THE "PICTURE" ATTRIBUTE.

ERROR 9, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE ARRAY BOUND.

ERROR 10, SEVFRITY 3
THE DIMENSION ATTRIBUTE DECLARED FOR $ DOES NOT SEEM TO END WITH A RIGHT PARENTHESIS.

ERROR 11, SEVERITY 3
THERE IS A MISSING RIGHT PARENTHESIS IN THF DECLARATION OF $.

ERROR 12, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE STRING LENGTH OR AREA SIZE EXPRESSION.

ERROR 13, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "GENERIC" ATTRIBUTE.

ERROR 14, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "OPTIONS" ATTRIBUTE.

ERROR 15, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE PRECISION OR SCALE ATTRIBUTE.

ERROR 16, SEVERITY 3
A LEVEL NUMBER GREATFR THAN ONE WAS NOT PRECEDED BY A STRUCTURE.
```

```
ERROR 17, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "OFFSET" ATTRIBUTE.

ERROR 18, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "BASED" ATTRIBUTE.

ERROR 19, SEVERITY 3
MULTIPLE "INITIAL" ATTRIBUTES HAVE BEEN DECLARED FOR $.

ERROR 20, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "INITIAL" ATTRIBUTE.

ERROR 21, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "LABEL" ATTRIBUTE.

ERROR 22, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "RETURNS" ATTRIBUTE.

ERROR 23, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "PICTURE" ATTRIBUTE.

ERROR 24, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "DEFINED" ATTRIBUTE.

ERROR 25, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "LIKE" ATTRIBUTE.

ERROR 26, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "POSITION" ATTRIBUTE.

ERROR 27, SEVERITY 2
DUPLICATE ATTRIBUTES HAVE BEEN FOUND WHILE PROCESSING A FACTORED ATTRIBUTE LIST. THE FACTORED ATTRIBUTE HAS BEEN
IGNORED IN FAVOR OF THE UNFACTORED ONE.

ERROR 28, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE ARGUMENT DESCRIPTOR REQUIRED TO PASS $ AS AN ARGUMENT EXCEEDS THE COMPILER'S LIMIT OF
512 WORDS. REDUCE THE COMPLEXITY OF THE VARIABLE, OR PASS A POINTER TO THE VARIABLE INSTEAD OF PASSING THE VARIABLE.

ERROR 29, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE ARGUMENT DESCRIPTOR REQUIRED TO PASS $ AS AN ARGUMENT EXCEEDS THE COMPILER'S LIMIT OF
128 MEMBERS IN A CONSTANT SIZE STRUCTURE. MAKE SOME EXTENT OF THE STRUCTURE VARIABLE, OR PASS A POINTER TO THE
STRUCTURE INSTEAD OF PASSING THE STRUCTURE.

ERROR 30, SEVERITY 3
SYNTAX ERROR IN STATEMENT LABEL $. A STATEMENT LABEL MUST BE AN IDENTIFIER OR AN IDENTIFIER FOLLOWED BY A SINGLE
(OPTIONALLY SIGNED) DECIMAL INTEGER CONSTANT ENCLOSED IN PARENTHESES.

ERROR 31, SEVERITY 3
$ APPEARS AS A LABEL PREFIX WITH AND WITHOUT SUBSCRIPTS WITHIN THE SAME BLOCK. ALL LABEL PREFIX REFERENCES TO A LABEL
ARRAY MUST BE SUBSCRIPTED.
```

ERROR 32, SEVERITY 3
COMPILER ERROR: THE PROCEDURE "COPY-EXPRESSION" HAS RECEIVED BAD INPUT. CORRECT ALL SOURCE PROGRAM ERRORS AND RE-COMPILE. IF THIS ERROR MESSAGE PERSISTS CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 33, SEVERITY 3
THE ENTRY NAME $ OCCURS AS AN ENTRY NAME ON MORE THAN ONE ENTRY OR PROCEDURE STATEMENT WITHIN THE SAME BLOCK.

ERROR 34, SEVERITY 3
IMPLEMENTATION RESTRICTION: THIS ENTRY OR PROCEDURE STATEMENT CONTAINS MORE PARAMETERS THAN THE IMPLEMENTATION MAXIMUM OF 128.

ERROR 35, SEVERITY 3
SYNTAX ERROR IN PARAMETER LIST OF ENTRY OR PROCEDURE STATEMENT.

ERROR 36, SEVERITY 3
SYNTAX ERROR FOLLOWING THE PARAMETER LIST OF AN ENTRY OR PROCEDURE STATEMENT.

ERROR 37, SEVERITY 3
SYNTAX ERROR FOLLOWING THE KEYWORD "RETURNS" IN AN ENTRY OR PROCEDURE STATEMENT.

ERROR 38, SEVERITY 3
SYNTAX ERROR FOLLOWING THE KEYWORD "OPTIONS" IN AN ENTRY OR PROCEDURE STATEMENT.

ERROR 39, SEVERITY 3
MULTIPLE "VALIDATE" OPTIONS HAVE BEEN SPECIFIED.

ERROR 40, SEVERITY 3
SYNTAX ERROR FOLLOWING THF KEYWORD "VALIDATE" IN AN ENTRY OR PROCEDURE STATEMENT.

ERROR 41, SEVERITY 2
$ IS NOT A VALID OPTION ON AN ENTRY OR PROCEDURE STATEMENT. IT AND THE REST OF THE STATEMENT HAVE BEEN IGNORED.

ERROR 42, SEVERITY 3
$ APPEARS IN A REFER-OPTION BUT DOES NOT APPEAR IN THE SAME BASED STRUCTURE, OR HAS NOT BEEN PROPERLY QUALIFIED BY ITS CONTAINING STRUCTURE'S NAMES.

ERROR 43, SEVERITY 2
SYNTAX ERROR IN THE CONDITION PREFIX LIST. UNABLE TO LOCATE A RIGHT PARENTHESIS FOLLOWING THE IDENTIFIER $.

ERROR 44, SEVERITY 2
SYNTAX ERROR IN THE CONDITION PREFIX LIST. A COLON MUST IMMEDIATELY FOLLOW A PARENTHESIZED LIST OF CONDITION PREFIXES.

ERROR 45, SEVERITY 2
A CONDITION PREFIX CANNOT CONTAIN THE CONDITION NAME $.

ERROR 46, SEVERITY 2
"IMPLEMENTATION" OR "IMP" SHOULD BE CHANGED TO "OPTIONS" TO CONFORM TO STANDARD PL/I.

```
WARNING 47
$ HAS BEEN PASSED AS AN ARGUMENT BY VALUE RATHER THAN BY REFERENCE BECAUSE ITS ATTRIBUTES DID NOT MATCH THE PARAMFTER
TO WHICH IT WAS PASSED.

ERROR 48, SEVERITY 3
SYNTAX ERROR IN AN APPARENT DEFAULT STATEMENT. STATEMENT IGNORED.

ERROR 49, SEVERITY 3
THE RIGHT-HAND-SIDE OF THIS APPARENT ASSIGNMENT STATEMENT IS NOT AN EXPRESSION.

ERROR 50, SEVERITY 3
THE ARRAY OR STRUCTURE VALUE IN THIS RETURN STATEMENT CANNOT BE RETURNED BECAUSE ALL ENTRIES THAT RETURN VALUES RETURN
ONLY SCALAR VALUES.

ERROR 51, SEVERITY 3
AREA VALUES CANNOT BE COMPARED.

ERROR 52, SEVERITY 3
COMPILER ERROR: "OPERATOR-SEMANTICS" HAS RECEIVED AN OPERATOR IT CANNOT PROCESS. CORRECT ALL SOURCF PROGRAM ERRORS. IF
THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 53, SEVERITY 3
THIS RETURN STATEMENT IS INVALID BECAUSE NO ENTRY TO THIS PROCEDURE HAS BEEN DECLARED WITH A RETURNS ATTRIBUTE. IF AN
ENTRY IS TO BE INVOKED AS A FUNCTION IT MUST HAVE A RETURNS ATTRIBUTE.

ERROR 54, SEVERITY 3
THE IDENTIFIER $ APPEARS AS A LABEL PREFIX ON MORE THAN ONE STATEMENT IN THE SAME BLOCK.

ERROR 55, SEVERITY 3
MULTIPLE LIKE ATTRIBUTES HAVE BEEN SPECIFIED FOR $. PROCESSING OF THIS DECLARATION HAS BEEN TERMINATED.

WARNING 56
THIS STATEMENT CAN NEVER BE REACHED DURING EXECUTION DUE TO THE PRESENCE OF AN UNCONDITIONAL GOTO STATEMENT OR RETURN
STATEMENT IMMEDIATELY PRECEDING IT.

ERROR 57, SEVERITY 3
THE DECLARATION OF $ CONTAINS A "DIMENSION" ATTRIBUTE THAT IS NOT FOLLOWED BY A LIST OF ARRAY BOUNDS.

ERROR 58, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE REFER-OPTION.

ERROR 59, SEVERITY 3
$ HAS BEEN FOUND AS A LABEL ON MORE THAN ONE STATEMENT WITHIN THE SAME BLOCK.

ERROR 60, SEVERITY 3
THE FACTORED STRUCTURE LEVEL NUMBER CONFLICTS WITH A PREVIOUSLY SPECIFIED STRUCTURE LEVEL NUMBER. THE FACTORED LEVEL
NUMBER HAS BEFN IGNORED.

ERROR 61, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE NUMBER OF ARGUMENTS OR SUBSCRIPTS USED IN THIS REFERENCE EXCEEDS THE COMPILER LIMIT OF
128.
```

```
ERROR 62, SEVERITY 3
AN ARRAY OR STRUCTURE VALUED EXPRESSION HAS BEEN FOUND IN A CONTEXT WHICH REQUIRES A SCALAR VALUE.

ERROR 63, SEVERITY 3
THE NAME $ CANNOT BE DECLARED WITH THE BUILTIN ATTRIBUTE, BECAUSE IT IS NOT A BUILTIN FUNCTION.

ERROR 64, SEVERITY 2
THE UNDECLARED IDENTIFIER $ HAS BEEN USED AS AN ENTRY BUT IS NOT A BUILTIN FUNCTION KNOWN TO THIS IMPLEMENTATION. IT
HAS BEEN DECLARED AS AN EXTERNAL ENTRY CONSTANT.

ERROR 65, SEVERITY 3
A REFERENCE TO THE GENERIC ENTRY $ COULD NOT BE RESOLVED BECAUSE NO APPROPRIATE MATCH COULD BE FOUND.

ERROR 66, SEVERITY 3
THE BASED VARIABLE $ HAS BEEN REFERENCED WITHOUT A LOCATOR QUALIFIER.

ERROR 67, SEVERITY 3
THE NON-BASED VARIABLE $ HAS BEEN REFERENCED WITH A LOCATOR QUALIFIED REFERENCE.

ERROR 68, SEVERITY 3
THE LOCATOR USED TO QUALIFY A REFERENCE TO $ IS AN ARRAY OR A STRUCTURE VALUE. LOCATOR QUALIFIERS MUST BE SCALAR
POINTER OR OFFSET EXPRESSIONS.

WARNING 69
THE UNDECLARED IDENTIFIER $ HAS BEEN CONTEXTUALLY DECLARED AS A POINTER. IT WILL ACQUIRE DEFAULT ATTRIBUTES.

FATAL ERROR 70
IMPLEMENTATION RESTRICTION: A LOCATOR QUALIFIED REFERENCE TO $ HAS MORE THAN 128 LEVELS OF POINTER QUALIFICATION IN ITS
SIZE, ARRAY BOUNDS OR SUBSCRIPTS. THIS HAS CAUSED THE SEMANTIC TRANSLATOR'S LOCATOR STACK TO OVERFLOW. REDUCE THE
COMPLEXITY OF THE VARIABLE'S DECLARATION AND RECOMPILE.

FATAL ERROR 71
SYNTAX ERROR IN %INCLUDE STATEMENT: END-OF-FILE OCCURRED BEFORE ENCOUNTERING A SEMI-COLON. CHECK FOR UNCLOSED COMMENT
OR STRING, OR A MISSING SEMI-COLON.

ERROR 72, SEVERITY 3
A FUNCTION REFERENCE CONTAINING AN ARRAY VALUED ENTRY EXPRESSION HAS BEEN FOUND. ALL FUNCTION REFERENCES MUST INVOKE A
SCALAR ENTRY VALUE.

ERROR 73, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN ARRAY OR STRUCTURE VALUE FOR ITS AREA SIZE, STRING LENGTH, OR ARRAY BOUNDS.

WARNING 74
THE UNDECLARED IDENTIFIER $ HAS BEEN CONTEXTUALLY DECLARED AS AN AREA. IT WILL ACQUIRE DEFAULT ATTRIBUTES.

WARNING 75
THE UNDECLARED IDENTIFIER $ HAS BEEN CONTEXTUALLY DECLARED AS A FILE CONSTANT. IT WILL ACQUIRE DEFAULT ATTRIBJTES.
```

```
WARNING 76
THE IDENTIFIER $ HAS BEEN MULTIPLY DECLARED. REFERENCES TO IT MAY NOT HAVE BEEN PROPERLY RESOLVED.

WARNING 77
THE UNDECLARED IDENTIFIER $ HAS BEEN DECLARED BY IMPLICATION. IT WILL ACQUIRE DEFAULT ATTRIBUTES.

ERROR 78, SEVERITY 3
A LABEL VALUE HAS BEEN FOUND IN A CONTEXT WHICH CANNOT CONTAIN A LABEL VALUE.

ERROR 79, SEVERITY 3
THE OPERANDS OF AN INFIX OPERATOR OR AN ASSIGNMENT OPERATOR ARE EITHER ARRAY VALUES WITH UNEQUAL DIMENSIONALITY OR
STRUCTURES WITH DIFFERENT STRUCTURING.

ERROR 80, SEVERITY 3
A SUBSCRIPTED REFERENCE TO THE LABEL CONSTANT $ CONTAINS MORE THAN ONE SUBSCRIPT.

ERROR 81, SEVERITY 3
A SUBSCRIPTED REFERENCE TO $ HAS FEWER SUBSCRIPTS THAN THE ARRAY HAS DIMENSIONS. TO REFERENCE A CROSS-SECTION USE
ASTERISK SUBSCRIPTS.

ERROR 82, SEVERITY 3
A SUBSCRIPTED REFERENCE TO $ HAS MORE SUBSCRIPTS THAN THE ARRAY HAS DIMENSIONS.

ERROR 83, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE CONSTANT LABEL ARRAY $ HAS BEEN PASSED AS AN ARGUMENT. THIS IMPLEMENTATION DOES NOT
SUPPORT THE PASSING OF LABEL ARRAY CONSTANTS. ELEMENTS OF SUCH ARRAYS MAY BE PASSED BUT THE ENTIRE ARRAY CANNOT BE
PASSED.

ERROR 84, SEVERITY 3
ONE OF THE SUBSCRIPTS IN A SUBSCRIPTED REFERENCE TO $ IS AN AGGREGATE VALUE. SUBSCRIPTS MUST BE SCALAR VALUES.

ERROR 85, SEVERITY 3
$ HAS BEEN INVOKED WITHOUT ARGUMENTS, BUT ITS DECLARATION INDICATES THAT ARGUMENTS ARE REQUIRED.

ERROR 86, SEVERITY 2
$ HAS BEEN INVOKED WITH ARGUMENTS BUT NO PARAMETER DESCRIPTORS WERE SPECIFIED IN ITS DECLARATION. STANDARD PL/I
REQUIRES THAT ALL ENTRIES BE FULLY DECLARED. IF THIS IS A NON-STANDARD PL/I ENTRY, DECLARE IT WITH THE
"OPTIONS(VARIABLE)" ATTRIBUTE.

ERROR 87, SEVERITY 3
$ HAS BEEN INVOKED WITH THE WRONG NUMBER OF ARGUMENTS.

ERROR 88, SEVERITY 3
$ IS A FUNCTION INVOKED BY A CALL STATEMENT.

ERROR 89, SEVERITY 3
IMPLEMENTATION RESTRICTION: A CROSS-SECTION ARRAY REFERENCE CANNOT BE PASSED AS AN ARGUMENT UNLESS THE RECEIVING
PARAMETER IS DECLARED AS AN ARRAY WITH ASTERISK BOUNDS.
```

ERROR 90, SEVERITY 3
THE LEFT SIDE OF AN AGGREGATE ASSIGNMENT IS NOT A REFERENCE TO AN AGGREGATE VARIABLE.

ERROR 91, SEVERITY 3
THE LEFT SIDE OF AN AGGREGATE ASSIGNMENT IS A REFERENCE TO A CONSTANT.

ERROR 92, SEVERITY 3
COMPILER ERROR: THE RIGHT SIDE OF AN AGGREGATE ASSIGNMENT STATEMENT SEEMS TO BE INCORRECTLY REPRESENTED. CORRECT ALL SOURCE PROGRAM ERRORS AND RECOMPILE. IF THIS MESSAGE PERSISTS CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 93, SEVERITY 3
THE LEFT SIDE OF THIS ASSIGNMENT IS A SCALAR BUT THE RIGHT SIDE IS AN AGGREGATE. NO CONVERSION FROM AGGREGATE TO SCALAR IS DEFINED IN PL/I.

ERROR 94, SEVERITY 3
IMPLEMENTATION RESTRICTION: RENAME-OPTION TABLE OVERFLOW. A MAXIMUM OF 20 RENAMES MAY BE SPECIFIED IN ONE COMPILATION.

ERROR 95, SEVERITY 3
SYNTAX ERROR IN A CONDITION PREFIX LIST. $ SHOULD BE AN IDENTIFIER.

ERROR 96, SEVERITY 3
A STATEMENT MUST BEGIN WITH AN IDENTIFIER OR SEMI-COLON, NOT A $.

ERROR 97, SEVERITY 3
$ HAS BEEN GIVEN THE "PARAMETER" ATTRIBUTE BUT DOES NOT APPEAR AS A PARAMETER OF ANY ENTRY TO THE PROCEDURF IN WHICH IT WAS DECLARED.

ERROR 98, SEVERITY 3
$ HAS BEEN GIVEN THE "STRUCTURE" ATTRIBUTE BUT IS NOT FOLLOWED BY A DECLARATION WHOSE LEVEL NUMBER IS GREATER THAN ITS OWN.

ERROR 99, SEVERITY 2
TEXT FOLLOWS THE LOGICAL END OF THE PROGRAM. CHECK FOR A PREMATURE END STATEMENT.

ERROR 100, SEVERITY 2
IMPLEMENTATION RESTRICTION: THE IDENTIFIER OR CONSTANT $ EXCEEDS THE MAXIMUM LENGTH OF 256 CHARACTERS AND HAS BEEN TRUNCATED.

ERROR 101, SEVERITY 2
THE SOURCE PROGRAM HAD INSUFFICIENT END STATEMENTS. ONE HAS BEEN SUPPLIED.

ERROR 102, SEVERITY 3
A STRUCTURE QUALIFIED REFERENCE TO $ HAS BEEN USED, BUT NO STRUCTURE DECLARATION CORRESPONDING TO THIS REFERENCE FXISTS WITHIN THE SCOPE KNOWN TO THE REFERENCE.

ERROR 103, SEVERITY 2
ONLY THE %INCLUDE MACRO IS IMPLEMENTED BY THE COMPILER. THIS STATEMENT HAS BEEN IGNORED.

ERROR 104, SEVERITY 2
THE TEXT NAME $ FOLLOWING %INCLUDE MUST BE AN IDENTIFIER OR A CHARACTER-STRING. THE INCLUDE MACRO HAS BEEN IGNORED.

```
ERROR 105, SEVERITY 3
IMPLEMENTATION RESTRICTION: THIS STATEMENT EXCEEDS THE MAXIMUM OF 3000 TOKENS AND HAS BEEN TRUNCATED AT THAT NUMBER.

ERROR 106, SEVERITY 2
IMPLEMENTATION RESTRICTION: THE FILE NAME $ IN THIS %INCLUDE MACRO HAS BEEN TRUNCATED TO 2 CHARACTERS.

FATAL ERROR 107
INCLUDE FILE $ NOT FOUND.

FATAL ERROR 108
INFINITE RECURSION OF INCLUDE TEXTS.

ERROR 109, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE STRING $ IS TOO LONG WHEN REPLICATED. REPLICATION HAS BEEN IGNORED.

ERROR 110, SEVERITY 3
THE REPLICATION COEFFICIENT FOR $ IS NOT A DECIMAL INTEGER. REPLICATION HAS BEEN IGNORED.

ERROR 111, SEVERITY 2
THE LOGICAL END OF THE PROGRAM OCCURS IN INCLUDE TEXT $. CHECK FOR A PREMATURE END STATEMENT.

FATAL ERROR 112
IMPLEMENTATION RESTRICTION: INCLUDE TEXT $ HAS EXCEEDED THE NESTING LIMIT OF 10. REDUCE THE NUMBER OF NESTED INCLUDE
TEXTS AND RE-COMPILE.

ERROR 113, SEVERITY 3
$ HAS BEEN GIVEN THE "MEMBER" ATTRIBUTE BUT DOES NOT HAVE A LEVEL GREATER THAN ONE OR DOES NOT FOLLOW A STRUCTURE
DECLARATION.

ERROR 114, SEVERITY 3
THE CONTROLLED VARIABLE $ MAY NOT BE ALLOCATED OR FREED IN A USER-SPECIFIED AREA.

ERROR 115, SEVERITY 3
THE VARIABLE $ HAS NOT BEEN DECLARED WITH THE "BASED" OR "CONTROLLED" ATTRIBUTE.

ERROR 116, SEVERITY 3
THE LOCATOR VARIABLE WHICH IS TO BE USED FOR THE ALLOCATION OF $ IS AN OFFSET VARIABLE. THEREFORE, AN AREA VARIABLE
MUST BE SPECIFIED EITHER BY THE USE OF AN IN-OPTION OR AN AREA VARIABLE IN THE OFFSET ATTRIBUTE USED TO DECLARE THE
LOCATOR. NO SUCH AREA VARIABLE CAN BE FOUND FOR THIS ATTEMPTED ALLOCATION.

ERROR 117, SEVERITY 3
THE LOCATOR VARIABLE WHICH IS TO BE USED FOR THE ALLOCATION OF $ HAS NOT BEEN DECLARED.

ERROR 118, SEVERITY 3
THE VARIABLE WHICH IS TO BE USED FOR THE ALLOCATION OF $ HAS NOT BEEN DECLARED AS AN AREA.

ERROR 119, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN INVALID LIKE ATTRIBUTE.
```

ERROR 120, SEVERITY 3
THE DECLARATION OF $ CONTAINS A LIKE ATTRIBUTE THAT DOES NOT REFER TO A STRUCTURE.

ERROR 121, SEVERITY 3
THE NUMBER OF ARGUMENTS USED IN A REFERENCE TO THE BUILTIN FUNCTION $ IS NOT EQUAL TO THE NUMBER OF ARGUMENTS REQUIRED BY THAT FUNCTION.

ERROR 122, SEVERITY 3
THE NUMBER OF ARGUMENTS USED IN A REFERENCE TO THE BUILTIN FUNCTION $ IS LESS THAN THE MINIMUM NUMBER OF ARGUMENTS REQUIRED BY THAT FUNCTION.

ERROR 123, SEVERITY 3
THE NUMBER OF ARGUMENTS USED IN A REFERENCE TO THE BUILTIN FUNCTION $ IS EITHER LESS THAN THE MINIMUM NUMBER OF ARGUMENTS REQUIRED BY THAT FUNCTION OR IS MORE THAN THE MAXIMUM ALLOWED BY THAT FUNCTION.

ERROR 124, SEVERITY 3
THE BUILTIN FUNCTION $ HAS BEEN REFERENCED WITH ARGUMENTS NOT ACCEPTABLE TO THE FUNCTION.

FATAL ERROR 125
SYNTAX ERROR: ANOTHER PER-CENT SIGN WAS ENCOUNTERED WHILE PARSING A %INCLUDE STATEMENT. CHECK FOR A MISSING SEMI-COLON.

ERROR 126, SEVERITY 3
TEMPORARY RESTRICTION: THE THIRD ARGUMENT FOR THE BUILTIN FUNCTION $ MUST BE A BIT-STRING CONSTANT.

ERROR 127, SEVERITY 3
THE FIRST ARGUMENT TO THE BUILTIN FUNCTION $ MUST BE AN ARRAY.

ERROR 128, SEVERITY 3
THE SECOND ARGUMENT TO THE BUILTIN FUNCTION $ SPECIFIES A DIMENSION WHICH IS NOT DECLARED FOR THIS ARRAY.

FATAL ERROR 129
IMPLEMENTATION RESTRICTION: THE INCLUDE FILE $ EXCEEDS THE MAXIMUM NUMBER (32) OF INCLUDE FILES ALLOWED. REDUCE THE NUMBER OF %INCLUDE STATEMENTS AND RE-COMPILE.

ERROR 130, SEVERITY 3
COMPILER ERROR: A CONVERSION ERROR HAS BEEN DETECTED IN TRYING TO CONVERT THE IDENTIFIER $ INTO SOME OTHER DATA TYPE. CORRECT ALL SOURCE PROGRAM ERRORS AND RECOMPILE. IF THIS MESSAGE PERSISTS CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 131, SEVERITY 3
TEMPORARY RESTRICTION: THE BUILTIN FUNCTION $ IS NOT IMPLEMENTED BY THIS VERSION OF THE COMPILER.

ERROR 132, SEVERITY 3
THE ARGUMENT OF THE BUILTIN FUNCTION $ MUST BE A REFERENCE TO A VARIABLE. LABELS, CONSTANTS, AND EXPRESSIONS ARE NOT ALLOWED.

WARNING 133
THE UNDECLARED IDENTIFIER $ HAS BEEN CONTEXTUALLY DECLARED AS A CONDITION NAME. IT WILL ACQUIRE DEFAULT ATTRIBUTES.

ERROR 134, SEVERITY 3
A LABEL CONSTANT HAS BEEN USED ON THE LEFT SIDE OF THIS ASSIGNMENT STATEMENT.

ERROR 135, SEVERITY 3
A FILE, ENTRY, OR FORMAT CONSTANT HAS BEEN USED ON THE LEFT SIDE OF THIS ASSIGNMENT STATEMENT.

ERROR 136, SEVERITY 3
COMPILER ERROR: A CONVERSION ERROR HAS BEEN DETECTED BY THE PROCEDURE "CONVERT". CORRECT ALL SOURCE PROGRAM ERRORS AND RECOMPILE. IF THIS MESSAGE PERSISTS CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 137, SEVERITY 3
TEMPORARY RESTRICTION: CROSS-SECTIONS OF ARRAY $ CANNOT BE PASSED AS AN ARGUMENT.

ERROR 138, SEVERITY 3
THE OBJECT $ OF THE REFER-OPTION MUST NOT BE LOCATOR QUALIFIED, AND IT MUST NOT BE SUBSCRIPTED.

ERROR 139, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE ARGUMENT $ TO THE BUILTIN FUNCTION "STRING" MUST EITHER BE ALL CHARACTER STRINGS OR ALL BIT STRINGS. NO MIXED DATA-TYPES ARE ACCEPTABLE.

ERROR 140, SEVERITY 3
AGGREGATE EXPRESSIONS CANNOT BE USED IN A DO-SPECIFICATION LIST.

ERROR 141, SEVERITY 3
THE $ BUILTIN FUNCTION CANNOT BE USED AS A PSEUDO VARIABLE WHEN ITS ARGUMENT IS NOT AN UNALIGNED NON-VARYING STRING OR AN AGGREGATE CONSISTING ENTIRELY OF UNALIGNED NON-VARYING BIT STRINGS OR ENTIRELY OF UNALIGNED CHARACTER-STRINGS.

ERROR 142, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE $ BUILTIN FUNCTION CANNOT ACCEPT ARGUMENTS WHICH ARE NOT UNALIGNED NON-VARYING STRINGS OR AGGREGATES THAT CONSIST ENTIRELY OF UNALIGNED NON-VARYING BIT STRINGS OR ENTIRELY OF UNALIGNED NON-VARYING CHARACTER-STRINGS.

ERROR 143, SEVERITY 3
IMPLEMENTATION RESTRICTION: AREA REFERENCES CANNOT BE USED IN A DO-SPECIFICATION LIST.

ERROR 144, SEVERITY 3
IMPLEMENTATION RESTRICTION: ADJUSTABLE STRINGS OR STRINGS WITH STAR-EXTENTS CANNOT BE USED IN A DO-SPECIFICATION LIST.

ERROR 145, SEVERITY 3
THE LEFT HAND SIDE OF AN ASSIGNMENT OPERATOR CANNOT BE AN EXPRESSION OR A LABEL CONSTANT.

ERROR 146, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE BUILTIN FUNCTION $ CREATES A RESULT WHOSE PRECISION EXCEEDS THE IMPLEMENTATION LIMITS. THE MAXIMUM ALLOWABLE PRECISION HAS BEEN USED. THE LIMITS ARE: FIXED BIN(71), FIXED DEC(63), FLOAT BIN(63) AND FLOAT DEC(63).

ERROR 147, SEVERITY 3
IN THE USE OF THE $ BUILTIN FUNCTION OR PSEUDO-VARIABLE THE SUBSTRING DOES NOT LIE COMPLETELY WITHIN THE STRING.

ERROR 148, SEVERITY 3
EXPRESSIONS OR PSEUDO-VARIABLES CANNOT BE USED AS ARGUMENTS TO THE $ PSEUDO-VARIABLE. THE ARGUMENT MUST BE A VARIABLE.

```
ERROR 149, SEVERITY 2
THE DECLARATION OF $ IS FOLLOWED BY DECLARATIONS WITH LEVEL NUMBERS GREATER THAN ONE, BUT IT IS NOT A STRUCTURE OR DOES
NOT HAVE A LEVEL NUMBER. IT HAS BEEN GIVEN A LEVEL OF ONE.

ERROR 150, SEVERITY 2
THIS ELSE CLAUSE HAS NO IF STATEMENT PRECEDING IT AND HAS BEEN IGNORED. CHECK FOR A MISSING END STATEMENT.

ERROR 151, SEVERITY 2
SYNTAX ERROR: AN UNDERSCORE ILLEGALLY BEGINS AN IDENTIFIER.

ERROR 152, SEVERITY 2
NON-BINARY DIGIT IN APPARENT BIT STRING $. A CHARACTER-STRING IS ASSUMED.

ERROR 153, SEVERITY 2
NON-BINARY DIGIT IN APPARENT BINARY CONSTANT $. CONSTANT ASSUMED TO BE DECIMAL.

ERROR 154, SEVERITY 2
A DECIMAL POINT OCCURRED IN THE EXPONENT OF THE CONSTANT $. THE EXPONENT FIELD IS TRUNCATED AT THE DECIMAL POINT.

ERROR 155, SEVERITY 2
NO EXPONENT IN THE FLOATING-POINT CONSTANT $. THE EXPONENT IS ASSUMED TO BE ZERO AND THE CONSTANT IS ASSUMED TO BE A
REAL FLOATING-POINT DECIMAL CONSTANT.

ERROR 156, SEVERITY 2
THE CONSTANT $ IS EITHER AN ISUB WHICH IS IN ERROR OR A SPACE IS MISSING BETWEEN THE CONSTANT AND AN IDENTIFIER WHICH
FOLLOWS. A DECIMAL INTEGER IS ASSUMED.

ERROR 157, SEVERITY 2
A LETTER IMMEDIATELY FOLLOWS THE CONSTANT $.

ERROR 158, SEVERITY 2
A CONSTANT IMMEDIATELY FOLLOWS THE IDENTIFIER $.

ERROR 159, SEVERITY 2
THIS LINE CONTAINS A CHARACTER WHICH IS NOT A MEMBER OF THE PL/I CHARACTER SET.

ERROR 160, SEVERITY 3
THE $ BUILTIN FUNCTION REQUIRES A SCALE FACTOR WHEN ITS ARGUMENTS ARE FIXED-POINT.

ERROR 161, SEVERITY 3
IN A RENAME-OPTION $ WAS ENCOUNTERED WHEN A LEFT PARENTHESIS WAS EXPECTED. THE RENAMING HAS BEEN DELETED.

ERROR 162, SEVERITY 3
IN A RENAME-OPTION $ WAS ENCOUNTERED WHEN EXPECTING THE FIRST NAME OF THE OPTION. THE RENAMING HAS BEEN DELETED.

ERROR 163, SEVERITY 3
IN A RENAME-OPTION $ COULD NOT BE FOUND AMONG THE LIST OF RENAMEABLE NAMES. THE RENAMING HAS BEEN DELETED.

ERROR 164, SEVERITY 3
IN A RENAME-OPTION $ WAS ENCOUNTERED WHEN A COMMA WAS EXPECTED. THE RENAMING HAS BEEN DELETED.
```

ERROR 165, SEVERITY 3
IN A RENAME-OPTION $ WAS ENCOUNTERED WHEN EXPECTING THE SECOND NAME OF THE OPTION. THE RENAMING HAS BEEN DELETED.

ERROR 166, SEVERITY 2
IN A RENAME-OPTION $ WAS ENCOUNTERED WHEN A RIGHT PARENTHESIS WAS EXPECTED.

ERROR 167, SEVERITY 3
THE $ BUILTIN FUNCTION CANNOT HAVE A SCALE FACTOR WHEN ITS ARGUMENTS ARE FLOATING-POINT.

ERROR 168, SEVERITY 3
ILLEGAL DECLARATION OF THE ARRAY $. THE LOWER BOUND IS GREATER THAN THE UPPER BOUND.

ERROR 169, SEVERITY 3
THE LABEL PREFIX OF A FORMAT STATEMENT CANNOT BE SUBSCRIPTED.

ERROR 170, SEVERITY 3
A COMPLEX FORMAT ITEM MUST CONTAIN EITHER "E", "F" OR "P" FORMAT ITEMS.

ERROR 171, SEVERITY 3
A REMOTE FORMAT ITEM MUST CONTAIN A FORMAT VALUE.

ERROR 172, SEVERITY 2
TEMPORARY RESTRICTION: AREA $ IS DIMENSIONED OR IS A MEMBER OF A DIMENSIONED STRUCTURE. THIS VERSION OF THF COMPILER CAN NOT PROPERLY SET THE AREA TO THE EMPTY STATE. THE VALUE OF THE "EMPTY" BUILTIN FUNCTION MAY BE ASSIGNED TO EACH ELEMENT OF THE ARRAY OF AREAS TO SET THEM TO THE EMPTY STATE.

ERROR 173, SEVERITY 2
TEMPORARY RESTRICTION: AREA $ IS AN EXTERNAL STATIC, DIMENSIONED AREA. THIS VERSION OF THE COMPILER CAN NOT PROPERLY SET THE AREA TO THE EMPTY STATE. THE VALUE OF THE "EMPTY" BUILTIN FUNCTION MAY BE ASSIGNED TO THE AREA TO SET IT TO THE EMPTY STATE.

ERROR 174, SEVERITY 2
TEMPORARY RESTRICTION: AREA $ IS A MEMBER OF AN EXTERNAL STATIC STRUCTURE. THIS VERSION OF THE COMPILER CAN NOT PROPERLY SET THE AREA TO THE EMPTY STATE. THE VALJE OF THE "EMPTY" BUILTIN FUNCTION MAY BE ASSIGNED TO THE AREA TO SET IT TO THE EMPTY STATE.

ERROR 175, SEVERITY 3
THE BASE REFERENCE OF THE DEFINED ATTRIBUTE DOES NOT REFERENCE A VARIABLE DECLARED IN ONE OF THE BLOCKS CONTAINING THE DECLARATION OF $.

ERROR 176, SEVERITY 3
THE BASE REFERENCE OF THE DEFINED ATTRIBUTE IN THE DECLARATION OF $ IDENTIFIES ANOTHER DEFINED VARIABLE OR A NAMED CONSTANT.

ERROR 177, SEVERITY 3
THE DECLARATION OF $ IS IN ERROR BECAUSE IT SPECIFIES ISUB OR SIMPLE DEFINING AND INCLUDES A "POSITION" ATTRIBUTE.

ERROR 178, SEVERITY 3
$ CANNOT BE STRING OVERLAY DEFINED ONTO THE VARIABLE IDENTIFIED BY THE BASE REFERENCE OF ITS DEFINED ATTRIBUTE.

ERROR 179, SEVERITY 3
$ CANNOT BE ISUB, SIMPLE, OR STRING OVERLAY DEFINED ONTO THE VARIABLE IDENTIFIED BY THE BASE REFERENCE OF ITS DEFINED ATTRIBUTE DUE TO DIFFERENCES IN THE ATTRIBUTES OR EXTENTS OF THE DEFINED VARIABLE AND ITS BASE.

FATAL ERROR 180
THE FIRST STATEMENT OF A PROGRAM MUST BE A PROCEDURE STATEMENT.

ERROR 181, SEVERITY 3
THE BASE REFERENCE OF THE DEFINED ATTRIBUTE IN THE DECLARATION OF $ CONTAINS TOO MANY ASTERISKS OR AN INVALID ISUB.

ERROR 182, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE ISUB DEFINED ARRAY $ CANNOT BE PASSED AS AN ARGUMENT.

ERROR 183, SEVERITY 3
THE BASE REFERENCE OF THE DEFINED ATTRIBUTE OF $ CONTAINS ISUBS OR ASTERISKS, BUT THE DEFINED VARIABLE IS NOT AN ARRAY.

ERROR 184, SEVERITY 3
A SUBSCRIPTED REFERENCE CONTAINS ONE OR MORE SUBSCRIPTS THAT ARE OUTSIDE THE BOUNDS OF THE ARRAY $.

ERROR 185, SEVERITY 3
THE POSITION ATTRIBUTE IN THE DECLARATION OF $ CONTAINS AN AGGREGATE VALUED EXPRESSION.

ERROR 186, SEVERITY 3
THE DATA TYPES OF THE OPERANDS OF A RELATIONAL OPERATOR CANNOT BE CONVERTED TO A COMMON TYPE SUITABLE FOR COMPARISON.

ERROR 187, SEVERITY 3
THE $ BUILTIN FUNCTION CAN ONLY USED AS THE RIGHT SIDE OF THE ASSIGNMENT STATEMENT, AS AN ARGUMENT, OR AS AN INITIAL VALUE IN AN "INITIAL" ATTRIBUTE.

ERROR 188, SEVERITY 3
THE IDENTIFIER $ IS NOT AN AREA VARIABLE, AND THEREFORE CANNOT BE ASSIGNED THE AREA RETURNED BY THE BUILTIN FUNCTION "EMPTY".

ERROR 189, SEVERITY 3
THE DECLARATION OF THE STRUCTURE $ CONTAINS BOTH A LIKE ATTRIBUTE AND STRUCTURE MEMBERS.

ERROR 190, SEVERITY 3
THE BUILTIN FUNCTION $ EXPECTS A ONE-DIMENSIONAL ARRAY AS ITS ARGUMENT.

ERROR 191, SEVERITY 2
STANDARD PL/I DOES NOT ALLOW THE "IMPLEMENTATION" ATTRIBUTE. USE "OPTIONS(VARIABLE)" OR "ENVIRONMENT(...)" TO AVOID THIS MESSAGE.

ERROR 192, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "OPTIONS" ATTRIBUTE.

ERROR 193, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN UNRECOGNIZABLE "ENVIRONMENT" ATTRIBUTE.

```
ERROR 194, SEVERITY 3
COMPILER ERROR: "DECLARE" WAS UNABLE TO FIND A DECLARATION OF $ WHEN PROCESSING A DECLARATION OF AN ENTRY CONSTANT.
CORRECT ALL SOURCE PROGRAM ERRORS AND RECOMPILE. IF THIS MESSAGE PERSISTS CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 195, SEVERITY 3
COMPILER ERROR: THE INTERNAL PROCEDURE "SIZE" IN "EXPAND-ASSIGN" HAS BEEN CALLED WITH A NULL POINTER OR A POINTER TO A
NODE OTHER THAN AN OPERATOR OR REFERENCE NODE. FIX ALL SOURCE PROGRAM ERRORS. IF THIS MESSAGE PERSISTS CONTACT THE
COMPILER MAINTENANCE PERSONNEL.

ERROR 196, SEVERITY 3
$ HAS BEEN USED AS A LABEL CONSTANT AND AS A PARAMETER IN THE SAME BLOCK.

ERROR 197, SEVERITY 3
AN OPEN STATEMENT CONTAINS AN UNRECOGNIZABLE "ENVIRONMENT" ATTRIBUTE.

ERROR 198, SEVERITY 3
ONLY STRING AND REAL ARITHMETIC VALUES CAN BE COMPARED WITH RELATIONAL OPERATORS OTHER THAN = AND ¬=.

ERROR 199, SEVERITY 3
A DESCRIPTOR WITH NO ATTRIBUTES WAS FOUND IN AN ENTRY(...) ATTRIBUTE, A RETURNS(...) ATTRIBUTE OR OPTION, OR A
WHEN(...) CLAUSE. AT LEAST ONE ATTRIBUTE IS REQUIRED IN A DESCRIPTOR.

ERROR 200, SEVERITY 3
THE DECLARATION OF $ CONTAINS INCOMPATIBLE ATTRIBUTES.

ERROR 201, SEVERITY 2
IMPLEMENTATION RESTRICTION: THE PRECISION OF $ EXCEEDS THE IMPLEMENTATION LIMIT. THE MAXIMUM ALLOWABLE PRECISION HAS
BEEN USED. THE LIMITS ARE: FIXED BIN(71), FIXED DEC(63), FLOAT BIN(63), FLOAT DEC(63).

ERROR 202, SEVERITY 3
TEMPORARY RESTRICTION: THIS VERSION OF THE COMPILER DOES NOT IMPLEMENT THE PICTURE FORMAT ITEM. REMOVE THE PICTURF
FORMAT ITEM AND RECOMPILE.

ERROR 203, SEVERITY 2
THE STRING VARIABLE $ HAS NO DECLARED LENGTH. A LENGTH OF ONE HAS BEEN SUPPLIED.

ERROR 204, SEVERITY 2
IMPLEMENTATION RESTRICTION: THE STRING OR AREA VARIABLE $ HAS A LENGTH OR SIZE LESS THAN THE MINIMUM ALLOWED BY THE
IMPLEMENTATION. STRING LENGTHS MUST BE NON-NEGATIVE, AREA SIZES MUST BE GREATER THAN 29.

ERROR 205, SEVERITY 2
IMPLEMENTATION RESTRICTION: THE STRING OR AREA VARIABLE $ HAS A LENGTH OR SIZE GREATER THAN THE MAXIMUM ALLOWED BY THE
IMPLEMENTATION. THE LIMITS ARE: CHAR(262144), BIT(2359296), AREA(65536).

WARNING 206
THIS DEFAULT STATEMENT WOULD CREATE AN ILLEGAL DECLARATION IF IT WERE APPLIED TO THE DECLARATION OF $. IT HAS NOT BEEN
APPLIED.

ERROR 207, SEVERITY 3
SYNTAX ERROR IN THE RANGE OPERAND OF THIS DEFAULT STATEMENT. THE STATEMENT HAS BEEN REMOVED FROM THE PROGRAM.
```

```
ERROR 208, SEVERITY 3
THE PREDICATE OF THIS DEFAULT STATEMENT CONTAINS NON-BOOLEAN OPERATORS. THE STATEMENT HAS BEEN REMOVED FROM THE
PROGRAM.

ERROR 209, SEVERITY 3
THE PREDICATE OF THIS DEFAULT STATEMENT CONTAINS AN ILLEGAL OPERAND. THE STATEMENT HAS BEEN REMOVED FROM THE PROGRAM.

ERROR 210, SEVERITY 3
$ IS A MEMBER OF A STRUCTURE BUT HAS BEEN DECLARED WITH THE "BASED", "AUTO", "STATIC", "CONTROLLED", "DEFINED",
"CONSTANT", OR "PARAMETER" ATTRIBUTES. MEMBERS OF STRUCTURES INHERIT THESE ATTRIBUTES FROM THEIR LEVEL-ONE CONTAINING
STRUCTURE.

ERROR 211, SEVERITY 3
THE DECLARATION OF $ HAS BEEN INVALIDATED BY THIS DEFAULT STATEMENT.

ERROR 212, SEVERITY 3
$ HAS THE "MEMBER" ATTRIBUTE OR HAS A LEVEL NUMBER GREATER THAN ONE BUT IT WAS NOT PRECEDED BY A STRUCTURE. THE LEVEL
NUMBER HAS BEEN SET TO ZERO AND THE "MEMBER" ATTRIBUTE REMOVED.

ERROR 213, SEVERITY 2
THE ENTRY CONSTANT $ HAS BEEN DECLARED IN A DECLARE STATEMENT. IT IS AN ERROR TO DECLARE SUCH ENTRIES IN A DECLARE
STATEMENT BECAUSE THEY ARE EXPLICITLY DECLARED BY THE ENTRY OR PROCEDURE STATEMENT. THE DECLARATION PRODUCED BY THE
DECLARE STATEMENT HAS BEEN REMOVED.

WARNING 214
THE UNDECLARED IDENTIFIER $ HAS BEEN CONTEXTUALLY DECLARED AS A PARAMETER. IT WILL ACQUIRE A DEFAULT DATA TYPE.

ERROR 215, SEVERITY 3
THE DECLARATION OF $ CONTAINS ASTERISK ARRAY BOUNDS, STRING LENGTHS, OR AREA SIZES. ONLY PARAMETERS MAY HAVE ASTERISKS
IN THESE CONTEXTS.

ERROR 216, SEVERITY 3
THE DECLARATION OF $ CONTAINS EXPRESSIONS IN ITS ARRAY BOUNDS, STRING LENGTHS, OR AREA SIZES. ONLY AUTOMATIC, BASED,
CONTROLLED OR DEFINED VARIABLES MAY HAVE EXPRESSIONS IN THESE CONTEXTS.

ERROR 217, SEVERITY 3
THE DECLARATION OF $ CONTAINS REFER-OPTIONS IN ITS ARRAY BOUNDS, STRING LENGTHS, OR AREA SIZES. ONLY MEMBERS OF BASED
STRUCTURE VARIABLES MAY HAVE REFER-OPTIONS.

ERROR 218, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN "EXTERNAL" ATTRIBUTE WHICH IS INCONSISTENT WITH THE VARIABLE'S STORAGE CLASS. ONLY
STATIC OR CONTROLLED VARIABLES MAY BE EXTERNAL.

ERROR 219, SEVERITY 3
THE DECLARATION OF $ CONTAINS A "VARYING" ATTRIBUTE WHICH IS INCONSISTENT WITH THE VARIABLE'S DATA TYPE. ONLY
BIT-STRING OR CHARACTER-STRING VARIABLES MAY BE VARYING.

ERROR 220, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN "INITIAL" ATTRIBUTE WHICH IS INCONSISTENT WITH THE VARIABLE'S STORAGE CLASS. PARAMETER
OR DEFINED VARIABLES CANNOT BE GIVEN INITIAL VALUES.
```

```
ERROR 221, SEVERITY 3
AN AMBIGUOUS REFERENCE TO $ HAS BEEN FOUND. ADDITIONAL STRUCTURE QUALIFICATION IS NECESSARY TO MAKE THE REFERENCE
UNIQUE.

ERROR 222, SEVERITY 2
IMPLEMENTATION RESTRICTION: THE SCALE FACTOR DECLARED FOR $ IS OUTSIDE THE RANGE -128<=Q<=127. IT HAS BEEN SET TO THE
MINIMUM OR MAXIMUM ALLOWED.

ERROR 223, SEVERITY 3
A NON-ENTRY VALUE HAS BEEN USED IN A CONTEXT WHICH REQUIRES AN ENTRY VALUE.

ERROR 224, SEVERITY 3
$ HAS BEEN USED IN A CONTEXT WHICH REQUIRES AN ENTRY VALUE.

ERROR 225, SEVERITY 3
A NON-STRING OR NON-ARITHMETIC VALUE HAS BEEN USED IN A CONTEXT WHICH REQUIRES A STRING VALUE.

ERROR 226, SEVERITY 3
$ HAS BEEN USED IN A CONTEXT WHICH REQUIRES A STRING VALUE.

ERROR 227, SEVERITY 3
A NON-ARITHMETIC OR NON-STRING VALUE HAS BEEN USED IN A CONTEXT WHICH REQUIRES AN ARITHMETIC VALUE.

ERROR 228, SEVERITY 3
$ HAS BEEN USED IN A CONTEXT WHICH REQUIRES AN ARITHMETIC VALUE.

ERROR 229, SEVERITY 3
A NON-LABEL VALUE HAS BEEN USED IN A CONTEXT WHICH REQUIRES A LABEL VALUE.

ERROR 230, SEVERITY 3
$ HAS BEEN USED IN A CONTEXT WHICH REQUIRES A LABEL VALUE.

ERROR 231, SEVERITY 3
A NON-LOCATOR VALUE HAS BEEN USED IN A CONTEXT WHICH REQUIRES A LOCATOR VALUE.

ERROR 232, SEVERITY 3
$ HAS BEEN USED IN A CONTEXT WHICH REQUIRES A LOCATOR VALUE.

WARNING 233
$ HAS BEEN CONVERTED FROM AN ARITHMETIC VALUE TO A STRING VALUE.

WARNING 234
AN ARITHMETIC VALUE HAS BEEN CONVERTED TO A STRING VALUE.

WARNING 235
$ HAS BEEN CONVERTED FROM A STRING VALUE TO AN ARITHMETIC VALUE.

WARNING 236
A STRING VALUE HAS BEEN CONVERTED TO AN ARITHMETIC VALUE.
```

```
ERROR 237, SEVERITY 3
A LEFT PARENTHESIS IS REQUIRED IN PLACE OF $.

ERROR 238, SEVERITY 3
A RIGHT PARENTHESIS IS REQUIRED IN PLACE OF $.

ERROR 239, SEVERITY 3
ILLEGAL OPTION OR COMBINATION OF OPTIONS USED IN INPUT/OUTPUT STATEMENT.

ERROR 240, SEVERITY 3
AN IDENTIFIER IS REQUIRED IN PLACE OF $.

ERROR 241, SEVERITY 3
THE FROM-OPTION MUST BE USED IN A WRITE STATEMENT.

ERROR 242, SEVERITY 2
TEMPORARY RESTRICTION: THIS VERSION OF THE COMPILER DOES NOT IMPLEMENT DEFAULTING OF CONSTANTS. THE PREDICATE OF THIS
DEFAULT STATEMENT SHOULD CONTAIN "&↑CONSTANT" UNTIL THIS RESTRICTION IS REMOVED.

WARNING 243
THE VARIABLE $ HAS BEEN DECLARED WITH THE "DEFINED" ATTRIBUTE. THE SYNTAX OF THE DEFINED ATTRIBUTE HAS BEEN CHANGED TO
"DEFINED [(<BASE REFERENCE>)]." THIS VERSION OF THE COMPILER TEMPORARILY USES THE OLD SYNTAX IF NO PARENTHESES ARE
PRESENT.

ERROR 244, SEVERITY 3
THE BUILTIN FUNCTION $ CANNOT BE USED AS A PSEUDO-VARIABLE. HENCE IT CANNOT APPEAR ON THE LEFT-HAND-SIDE OF A
STATEMENT.

ERROR 245, SEVERITY 3
THE FILE-OPTION MUST BE SUPPLIED IN THIS KIND OF INPUT/OUTPUT STATEMENT.

ERROR 246, SEVERITY 3
THE CONSTANT $ CANNOT BE CONVERTED TO A DATA-TYPE OTHER THAN ARITHMETIC-TYPE AND STRING-TYPE.

ERROR 247, SEVERITY 3
IMPROPER OPTION FOR THIS INPUT/OUTPUT STATEMENT, OR INVALID REDUNDANT USE OF THE SAME OPTION.

ERROR 248, SEVERITY 3
COMPILER ERROR: BAD INPUT $ TO THE CONVERSION ROUTINE "CONVERT". CORRECT ALL PROGRAM ERRORS AND RECOMPILE. IF THIS
MESSAGE PERSISTS CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 249, SEVERITY 3
A NON-AREA HAS BEEN USED IN A CONTEXT WHICH REQUIRES AN AREA.

ERROR 250, SEVERITY 3
$ HAS BEEN USED IN A CONTEXT WHICH REQUIRES AN AREA.

ERROR 251, SEVERITY 3
A NON-FILE VALUE HAS BEEN USED IN A CONTEXT WHICH REQUIRES A FILE VALUE.
```

```
ERROR 252, SEVERITY 3
$ HAS BEEN USED IN A CONTEXT WHICH REQUIRES A FILE VALUE.

ERROR 253, SEVERITY 3
A CONVERSION ERROR OCCURRED WHEN $ WAS CONVERTED TO ITS INTERNAL REPRESENTATION.

ERROR 254, SEVERITY 3
A DATA LIST IS REQUIRED IN THIS GET OR PUT STATEMENT. ONLY THE GET DATA OR PUT DATA STATEMENT CAN BE WRITTEN WITHOUT A
DATA LIST.

ERROR 255, SEVERITY 3
THE "EDIT" OR "LIST" KEYWORDS OF A GET OR PUT STATEMENT MUST BE FOLLOWED BY A PARENTHESIZED DATA LIST. $ WAS FOUND IN
PLACE OF THE LEFT PARENTHESIS.

ERROR 256, SEVERITY 3
SYNTAX ERROR IN A DATA-SPECIFICATION LIST. THE STATEMENT APPEARS TO BE IN ERROR SOMEWHERE AFTER THE TOKEN $.

ERROR 257, SEVERITY 3
SYNTAX ERROR IN LOCATE STATEMENT. AN IDENTIFIER SHOULD IMMEDIATELY FOLLOW THE WORD "LOCATE".

ERROR 258, SEVERITY 3
A DATA LIST DO-GROUP APPEARS TO BE TERMINATED BY $ INSTEAD OF A RIGHT PARENTHESIS.

ERROR 259, SEVERITY 2
IMPLEMENTATION RESTRICTION: MORE THAN 16384 WORDS OF INTERNAL STATIC STORAGE HAVE BEEN DECLARED. MOVE SOME VARIABLES TO
EXTERNAL STATIC STORAGE AND RE-COMPILE.

ERROR 260, SEVERITY 3
A CONVERSION ERROR OCCURRED WHEN THE INITIAL VALUE OF $ WAS CONVERTED TO ITS INTERNAL REPRESENTATION.

ERROR 261, SEVERITY 3
COMPILER ERROR: TARGET PRECISION MISSING IN CONVERSION FROM FLOAT TO FIXED. CORRECT ALL SOURCE PROGRAM ERRORS AND
RE-COMPILE. IF THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 262, SEVERITY 3
COMPILER ERROR: TARGET PRECISION MISSING IN CONVERSION OF $ FROM FLOAT TO FIXED. CORRECT ALL SOURCE PROGRAM ERRORS AND
RE-COMPILE. IF THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 263, SEVERITY 3
THE SUBROUTINE $ IS INVOKED BY A FUNCTION REFERENCE.

ERROR 264, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE MAXIMUM NUMBER OF VALUES IN AN INITIAL LIST IS 256. THE DECLARATION FOR $ EXCEEDS THIS
MAXIMUM.

ERROR 265, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE EMPTY BUILTIN FUNCTION CANNOT BE ASSIGNED TO A STRUCTURE $.
```

```
ERROR 266, SEVERITY 3
A PROCEDURE STATEMENT OR ENTRY STATEMENT MUST HAVE AT LEAST ONE LABEL.

WARNING 267
PROCEDURE $ HAS NEVER BEEN REFERENCED.

ERROR 268, SEVERITY 3
A NON-CONDITION NAME HAS BEEN USED IN A CONTEXT WHICH REQUIRES A CONDITION NAME.

ERROR 269, SEVERITY 3
THE 'DIMENSION' ATTRIBUTE OF THE PARAMETER OR THE PARAMETER DESCRIPTOR MUST HAVE THE SAME DIMENSIONALITY AS THE ARRAY
ARGUMENT $.

ERROR 270, SEVERITY 3
THE LABEL OF AN ENTRY STATEMENT OR A PROCEDURE STATEMENT CANNOT BE SUBSCRIPTED.

ERROR 271, SEVERITY 3
THE SECOND ARGUMENT TO THE BUILTIN FUNCTION $ MUST BE NON-NEGATIVE IF THE ARGUMENT TO BE ROUNDED HAS THE FLOAT
ATTRIBUTE.

ERROR 272, SEVERITY 3
EVALUATION OF THE OFFSET EXPRESSION TO THE IDENTIFIER $ IS NOT POSSIBLE DUE TO THE PRESENCE OF AN ASTERISK AS ONE OF
ITS SUBSCRIPTS.

ERROR 273, SEVERITY 3
$ IS NOT A LEVEL ONE ITEM. THEREFORE, IT CANNOT BE ALLOCATED.

ERROR 274, SEVERITY 3
SYNTAX ERROR IN A FORMAT-LIST. THE ITERATION FACTOR SEEMS MISSED OR ERRONEOUS.

ERROR 275, SEVERITY 3
A DATA LIST HAS A $ WHERE "DO", ")", OR "," SHOULD APPEAR.

ERROR 276, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE BUILTIN FUNCTION $ CANNOT BE USED AS A PSEUDO-VARIABLE AS YET.

ERROR 277, SEVERITY 3
IMPLEMENTATION RESTRICTION: MATHEMATICAL BUILTIN FUNCTIONS WITH AGGREGATE ARGUMENTS ARE NOT IMPLEMENTED YET.

ERROR 278, SEVERITY 3
SYNTAX ERROR IN A FORMAT-LIST. THE SYMBOL $ SEEMS OUT OF PLACE.

ERROR 279, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN INCOMPLETE "RETURNS" ATTRIBUTE.

ERROR 280, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN INCOMPLETE "DIMENSION" ATTRIBUTE.

ERROR 281, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN INCOMPLETE "PICTURE" ATTRIBUTE.
```

```
ERROR 282, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN INCOMPLETE "POSITION" ATTRIBUTE.

ERROR 283, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN INCOMPLETE "INITIAL" ATTRIBUTE.

ERROR 284, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN INCOMPLETE "GENERIC" ATTRIBUTE.

ERROR 285, SEVERITY 3
THE DECLARATION OF $ CONTAINS AN INCOMPLETE "ENVIRONMENT" ATTRIBUTE.

ERROR 286, SEVERITY 2
*UNUSED*

ERROR 287, SEVERITY 2
*UNUSED*

ERROR 288, SEVERITY 2
UNRECOGNIZABLE OPTION OR ATTRIBUTE $ IN AN INPUT/OUTPUT STATEMENT. IT HAS BEEN IGNORED.

ERROR 289, SEVERITY 3
A "DATA", "EDIT", OR "LIST" OPTION MAY ONLY BE USED IN A GET STATEMENT OR A PUT STATEMENT.

ERROR 290, SEVERITY 3
AN EXPRESSION IS REQUIRED IN PLACE OF $.

ERROR 291, SEVERITY 3
COMPILER ERROR: THERE IS NO POINTER IN THE STACK TO BE USED BY THE BIT-POINTER OPERATOR. CORRECT ALL SOURCF PROGRAM
ERRORS AND RE-COMPILE. IF THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 292, SEVERITY 2
*UNUSED*

ERROR 293, SEVERITY 3
SYNTAX ERROR IN A DATA-LIST SPECIFICATION: POSSIBLY MISSING A PAIR OF PARENTHESES AROUND A DO-GROUP.

ERROR 294, SEVERITY 2
*UNUSED*

ERROR 295, SEVERITY 2
*UNUSED*

ERROR 296, SEVERITY 2
*UNUSED*

ERROR 297, SEVERITY 2
*UNUSED*
```

ERROR 298, SEVERITY 2
UNUSED

ERROR 299, SEVERITY 2
UNUSED

ERROR 300, SEVERITY 3
COMPILER ERROR: OPCODE $ FOUND BY CODE GENERATOR IN WRONG CONTEXT. RETAIN OUTPUT AND CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 301, SEVERITY 3
COMPILER ERROR: OPCODE $ NOT YET HANDLED BY CODE GENERATOR. RETAIN OUTPUT AND CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 302, SEVERITY 3
COMPILER ERROR: MACRO $ NOT FOUND IN MACRO TABLE. RETAIN OUTPUT AND CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 303, SEVERITY 3
COMPILER ERROR: MACRO $ USED WITH TOO FEW ARGUMENTS. RETAIN OUTPUT AND CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 304, SEVERITY 3
COMPILER ERROR: MACRO $ HAS NO BODY. RETAIN OUTPUT AND CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 305, SEVERITY 3
COMPILER ERROR: STORAGE CLASS $ NOT HANDLED. RETAIN OUTPUT AND CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 306, SEVERITY 2
IMPLEMENTATION RESTRICTION: THE DEFINED VARIABLE $ CANNOT BE REPRESENTED IN THE RUN-TIME SYMBOL TABLE REQUIRED BY THE "TABLE" OPTION OR DATA-DIRECTED I/O. THIS IS DUE TO THE USE OF A "*" OR "ISUB" IN THE DEFINING DECLARATION.

WARNING 307
THE VARIABLE $ HAS BEEN REFERENCED BUT HAS NEVER BEEN SET.

ERROR 308, SEVERITY 3
COMPILER ERROR: CONDITION $ OCCURRED IN WRONG CONTEXT. RETAIN OUTPUT AND CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 309, SEVERITY 3
COMPILER ERROR: COMPILER PROCEDURE $ HAS BEEN CALLED BUT IS NOT YET IMPLEMENTED. CORRECT ALL SOURCE PROGRAM ERRORS AND RECOMPILE. IF THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

FATAL ERROR 310
COMPILER ERROR: $ WHILE IN THE CODE GENERATOR. CORRECT ALL SOURCE PROGRAM ERRORS AND RE-COMPILE. IF THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

FATAL ERROR 311
IMPLEMENTATION RESTRICTION: THE MAXIMUM PROGRAM SIZE OF $ WORDS HAS BEEN EXCEEDED. REDUCE THE SIZE OF THE SOURCE PROGRAM AND RE-COMPILE.

ERROR 312, SEVERITY 2
UNUSED

```
ERROR 313, SEVERITY 2
*UNUSED*

ERROR 314, SEVERITY 2
COMPILER ERROR: REFERENCE COUNT < 0 WHEN FREEING TEMPORARY NODE $. CORRECT ALL SOURCE PROGRAM ERRORS AND RE-COMPILE. IF
THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 315, SEVERITY 3
COMPILER ERROR: TEMPORARY NODE $ DOES NOT HAVE VALUE IN STORAGE. CORRECT ALL SOURCE PROGRAM ERRORS AND RE-COMPILE. IF
THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 316, SEVERITY 3
COMPILER ERROR: ATTEMPT TO ACCESS TEMPORARY $ WITH REFERENCE COUNT <= 0. CORRECT ALL SOURCE PROGRAM ERRORS AND
RE-COMPILE. IF THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 317, SEVERITY 3
COMPILER ERROR: LENGTH OR OFFSET EXPRESSION FOUND WITH OPERAND(1) = NULL. CORRECT ALL SOURCE PROGRAM ERRORS AND
RE-COMPILE. IF THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 318, SEVERITY 2
COMPILER ERROR: ONE OR MORE FIELDS IN THE RUNTIME SYMBOL TABLE ENTRY OF $ CANNOT BE ENCODED. CORRECT ALL SOURCE PROGRAM
ERRORS AND RECOMPILE. IF THIS MESSAGE PERSISTS, CONTACT THE COMPILER MAINTENANCE PERSONNEL.

WARNING 319
THE "STRINGSIZE" CONDITION WILL BE RAISED WHEN THIS STATEMANT IS EXECUTED.

ERROR 320, SEVERITY 3
THE STATIC LABEL VARIABLE $ HAS BEEN DECLARED WITH AN INITIAL ATTRIBUTE. STATIC LABEL VARIABLES CANNOT BE INITIALIZED
BECAUSE A LABEL VALUE CONSISTS OF AN ENVIRONMENT OR BLOCK ACTIVATION POINTER IN ADDITION TO THE ADDRESS OF A STATEMENT.
SINCE STATIC VARIABLES ARE INITIALIZED PRIOR TO BLOCK ACTIVATION THEY CANNOT POSSIBLY BE INITIALIZED.

ERROR 321, SEVERITY 3
THE STATIC AREA VARIABLE $ HAS BEEN DECLARED WITH AN INITIAL ATTRIBUTE. STATIC AREA VARIABLES CANNOT BE INITIALIZED BY
THE PROGRAMMER BECAUSE EXPRESSIONS OR CALLS ARE NOT PERMITTED AS STATIC INITIAL VALUES.

ERROR 322, SEVERITY 3
THE STATIC ENTRY VARIABLE $ HAS BEEN DECLARED WITH AN INITIAL ATTRIBUTE. STATIC ENTRY VARIBLES CANNOT BE INITIALIZED
BECAUSE AN ENTRY VALUE CONSISTS OF AN ENVIRONMENT OR BLOCK ACTIVATION POINTER IN ADDITION TO THE ADDRESS OF AN ENTRY
POINT. SINCE STATIC VARIABLES ARE INITIALIZED PRIOR TO BLOCK ACTIVATION $ CANNOT POSSIBLY BE INITIALIZED.

ERROR 323, SEVERITY 3
THE STATIC VARIABLE $ HAS BEEN DECLARED WITH AN INITIAL-CALL AS ITS INITIAL VALUE. BECAUSE STATIC VARIABLES ARE
INITIALIZED PRIOR TO BLOCK ENTRY, ENTRY EXPRESSIONS AND CALLS ARE NOT PERMITTED AS INITIAL VALUES OF STATIC DATA.

ERROR 324, SEVERITY 3
THE STATIC VARIABLE $ HAS AN INITIAL ATTRIBUTE WHICH CONTAINS NON-CONSTANT REPETITION FACTORS OR INITIAL VALUES.
BECAUSE STATIC VARIABLES ARE INITIALIZED PRIOR TO BLOCK ENTRY, EXPRESSIONS ARE NOT PERMITTED IN THE INITIAL ATTRIBUTE
OF STATIC DATA.
```

```
ERROR 325, SEVERITY 3
THE PROGRAM CONTAINS AN INFINITE LOOP.

ERROR 326, SEVERITY 2
AN ELEMENT OF THE CONSTANT LABEL ARRAY $ HAS BEEN USED AS A LABEL ON MORE THAN ONE STATEMENT.

ERROR 327, SEVERITY 2
IMPLEMENTATION RESTRICTION: ONLY $ DECLARATIONS OF A TYPE MAY BE PRINTED.

ERROR 328, SEVERITY 2
*UNUSED*

ERROR 329, SEVERITY 2
*UNUSED*

ERROR 330, SEVERITY 2
*UNUSED*

ERROR 331, SEVERITY 2
*UNUSED*

ERROR 332, SEVERITY 2
*UNUSED*

ERROR 333, SEVERITY 2
*UNUSED*

ERROR 334, SEVERITY 2
*UNUSED*

ERROR 335, SEVERITY 2
*UNUSED*

ERROR 336, SEVERITY 2
*UNUSED*

ERROR 337, SEVERITY 2
*UNUSED*

ERROR 338, SEVERITY 2
*UNUSED*

ERROR 339, SEVERITY 2
*UNUSED*

ERROR 340, SEVERITY 2
*UNUSED*

ERROR 341, SEVERITY 2
*UNUSED*
```

```
ERROR 342, SEVERITY 2
*UNUSED*

ERROR 343, SEVERITY 2
*UNUSED*

ERROR 344, SEVERITY 2
*UNUSED*

ERROR 345, SEVERITY 2
*UNUSED*

ERROR 346, SEVERITY 2
*UNUSED*

ERROR 347, SEVERITY 2
*UNUSED*

ERROR 348, SEVERITY 2
*UNUSED*

ERROR 349, SEVERITY 2
*UNUSED*

ERROR 350, SEVERITY 2
*UNUSED*

ERROR 351, SEVERITY 2
*UNUSED*

ERROR 352, SEVERITY 2
*UNUSED*

ERROR 353, SEVERITY 2
*UNUSED*

ERROR 354, SEVERITY 2
*UNUSED*

ERROR 355, SEVERITY 2
*UNUSED*

ERROR 356, SEVERITY 2
*UNUSED*

ERROR 357, SEVERITY 2
*UNUSED*

ERROR 358, SEVERITY 2
*UNUSED*
```

```
ERROR 359, SEVERITY 2
*UNUSED*

ERROR 360, SEVERITY 2
*UNUSED*

ERROR 361, SEVERITY 2
*UNUSED*

ERROR 362, SEVERITY 2
*UNUSED*

ERROR 363, SEVERITY 2
*UNUSED*

ERROR 364, SEVERITY 2
*UNUSED*

ERROR 365, SEVERITY 2
*UNUSED*

ERROR 366, SEVERITY 2
*UNUSED*

ERROR 367, SEVERITY 2
*UNUSED*

ERROR 368, SEVERITY 2
*UNUSED*

ERROR 369, SEVERITY 2
*UNUSED*

ERROR 370, SEVERITY 2
*UNUSED*

ERROR 371, SEVERITY 2
*UNUSED*

ERROR 372, SEVERITY 2
*UNUSED*

ERROR 373, SEVERITY 2
*UNUSED*

ERROR 374, SEVERITY 2
*UNUSED*
```

```
ERROR 375, SEVERITY 2
*UNUSED*

ERROR 376, SEVERITY 2
*UNUSED*

ERROR 377, SEVERITY 2
*UNUSED*

ERROR 378, SEVERITY 2
*UNUSED*

ERROR 379, SEVERITY 2
*UNUSED*

ERROR 380, SEVERITY 2
*UNUSED*

ERROR 381, SEVERITY 2
*UNUSED*

ERROR 382, SEVERITY 2
*UNUSED*

ERROR 383, SEVERITY 2
*UNUSED*

ERROR 384, SEVERITY 2
*UNUSED*

ERROR 385, SEVERITY 2
*UNUSED*

ERROR 386, SEVERITY 2
*UNUSED*

ERROR 387, SEVERITY 2
*UNUSED*

ERROR 388, SEVERITY 2
*UNUSED*

ERROR 389, SEVERITY 2
*UNUSED*

ERROR 390, SEVERITY 2
*UNUSED*

ERROR 391, SEVERITY 2
*UNUSED*
```

```
ERROR 392, SEVERITY 2
*UNUSED*

ERROR 393, SEVERITY 2
*UNUSED*

ERRCR 394, SEVERITY 2
*UNUSED*

ERROR 395, SEVERITY 2
*UNUSED*

ERROR 396, SEVERITY 2
*UNUSED*

ERROR 397, SEVERITY 2
*UNUSED*

ERROR 398, SEVERITY 2
*UNUSED*

ERROR 399, SEVERITY 2
*UNUSED*

ERROR 400, SEVERITY 3
THIS STATEMENT COULD NOT BE RECOGNIZED.

ERROR 401, SEVERITY 3
AN ILLEGAL OCCURRENCE OF AN ELSE CLAUSE.

ERROR 402, SEVERITY 2
SYNTAX ERROR IN A DO WHILE(...) STATEMENT.

ERROR 403, SEVERITY 3
AN ILLEGAL OCCURRENCE OF THE SYSTEM ON-UNIT.

ERROR 404, SEVERITY 2
THE EXPRESSION FOLLOWING THE KEYWORD "WHILE" MUST BE ENCLOSED IN PARENTHESES.

ERROR 405, SEVERITY 2
SYNTAX ERROR IN WHILE EXPRESSION OR MISSING RIGHT PARENTHESIS FOLLOWING THE WHILE EXPRESSION.

ERROR 406, SEVERITY 3
SYNTAX ERROR IN THE CONTROL VARIABLE OF A DO STATEMENT. ITERATION HAS BEEN OMITTED.

ERROR 407, SEVERITY 3
THE CONTROL VARIABLE OF A DO STATEMENT WAS NOT FOLLOWED BY AN EQUAL SIGN. ITERATION HAS BEEN OMITTED.

ERROR 408, SEVERITY 3
SYNTAX ERROR IN THE FIRST EXPRESSION OF A DO-STATEMENT SPECIFICATION. ITERATION HAS BEEN OMITTED.
```

ERROR 409, SEVERITY 3
THE "TO" CLAUSE WAS ENCOUNTERED TWICE IN A DO-STATEMENT SPECIFICATION. ITERATION HAS BEEN OMITTED.

ERROR 410, SEVERITY 3
SYNTAX ERROR IN A BEGIN STATEMENT.

ERROR 411, SEVERITY 3
AN ENTRY STATEMENT IS NOT ALLOWED WITHIN A BEGIN BLOCK.

ERROR 412, SEVERITY 3
A RETURN STATEMENT IS NOT ALLOWED WITHIN AN ON-UNIT BEGIN BLOCK.

ERROR 413, SEVERITY 3
AN ENTRY STATEMENT MAY NOT BE CONTAINED WITHIN AN ITERATED DO GROUP.

ERROR 414, SEVERITY 3
IMPLEMENTATION RESTRICTION; A NORMALIZED PICTURE CANNOT CONTAIN MORE THAN 64 CHARACTERS.

ERROR 415, SEVERITY 2
SYNTAX ERROR IN AN END STATEMENT. THE SYMBOL $ WHICH FOLLOWS A LABEL AFTER THE KEYWORD "END" IS NOT A SEMI-COLON.

ERROR 416, SEVERITY 2
SYNTAX ERROR IN AN END STATEMENT. THE SYMBOL $ WHICH FOLLOWS THE KEYWORD "END" IS NEITHER AN IDENTIFIER NOR A SEMI-COLON.

ERROR 417, SEVERITY 2
THE IDENTIFIER $ FOLLOWING THE KEYWORD "END" ON AN END STATEMENT COULD NOT BE MATCHED WITH A PREVIOUS LABEL ON A DO, BEGIN, OR PROCEDURE STATEMENT.

ERROR 418, SEVERITY 3
SYNTAX ERROR IN THE EXPRESSION OF A "TO" CLAUSE IN A DO-STATEMENT CONTROL. ITERATION HAS BEEN OMITTED.

ERROR 419, SEVERITY 3
THE "BY" CLAUSE WAS ENCOUNTERED TWICE IN A DO-STATEMENT CONTROL. ITERATION HAS BEEN OMITTED.

ERROR 420, SEVERITY 3
SYNTAX ERROR IN THE CONDITION NAME OF AN ON, REVERT, OR SIGNAL STATEMENT.

ERROR 421, SEVERITY 2
ON-UNITS MAY NOT BE LABELED.

ERROR 422, SEVERITY 2
THE TOKEN $ AFTER THE IDENTIFIER "SYSTEM" IN AN ON-STATEMENT HAS BEEN IGNORED.

ERROR 423, SEVERITY 3
AN ILLEGAL STATEMENT OCCURRED WITHIN AN ON STATEMENT.

ERROR 424, SEVERITY 3
SYNTAX ERROR IN THE EXPRESSION OF A "BY" CLAUSE IN A DO-STATEMENT CONTROL. ITERATION HAS BEEN OMITTED.

ERROR 425, SEVERITY 3
A DO-STATEMENT CONTROL IS NOT TERMINATED BY A COMMA OR SEMI-COLON. ITERATION HAS BEEN OMITTED.

ERROR 426, SEVERITY 3
SYNTAX ERROR IN THE EXPRESSION OF A "WHILE" CLAUSE IN A DO-STATEMENT CONTROL. ITERATION HAS BEEN OMITTED.

ERROR 427, SEVERITY 3
SYNTAX ERROR IN A COMPLEX FORMAT-LIST.

ERROR 428, SEVERITY 2
SYNTAX ERROR IN A FORMAT STATEMENT. A SEMI-COLON MUST IMMEDIATELY FOLLOW THE FORMAT-LIST. THE CHARACTERS AFTER THE APPARENT END OF THE FORMAT LIST HAVE BEEN IGNORED.

ERROR 429, SEVERITY 3
SYNTAX ERROR IN THE EXPRESSION OF A "REPEAT" CLAUSE IN A DO-STATEMENT CONTROL. ITERATION HAS BEEN OMITTED.

ERROR 430, SEVERITY 3
AN ILLEGAL STATEMENT OCCURRED WITHIN AN IF STATEMENT.

ERROR 431, SEVERITY 2
THE KEYWORD "THEN" IS MISSING FROM AN IF STATEMENT.

ERROR 432, SEVERITY 3
SYNTAX ERROR IN THE EXPRESSION OF AN IF STATEMENT.

ERROR 433, SEVERITY 3
SYNTAX ERROR IN A DO-STATEMENT CONTROL. THE "REPEAT" CLAUSE MAY NOT BE USED WITH A "TO" OR "BY" CLAUSE.

ERROR 434, SEVERITY 3
IMPLEMENTATION RESTRICTION: THE SCALE FACTOR OF A PICTURE MUST LIE BETWEEN -128 AND +127.

ERROR 435, SEVERITY 3
ILLEGAL CONVERSION BETWEEN POINTER AND OFFSET VARIABLES BECAUSE NO AREA HAS BEEN DECLARED TO ASSOCIATE WITH THE OFFSET VARIABLE $.

ERROR 436, SEVERITY 3
THE SECOND ARGUMENT $ USED IN THE POINTER BUILTIN FUNCTION MUST BE A BIT-STRING OR AN INTEGER.

ERROR 437, SEVERITY 3
THE SECOND ARGUMENT $ USED IN THE POINTER BUILTIN FUNCTION MUST BE AN AREA VARIABLE.

ERROR 438, SEVERITY 3
THE FIRST ARGUMENT $ USED IN THE POINTER BUILTIN FUNCTION MUST BE EITHER A POINTER OR AN OFFSET VARIABLE.

ERROR 439, SEVERITY 3
SYNTAX ERROR IN A REMOTE FORMAT.

ERROR 440, SEVERITY 3
SYNTAX ERROR IN A PICTURE DECLARED FOR $.

```
ERROR 441, SEVERITY 3
SYNTAX ERROR IN %INCLUDE STATEMENT.

ERROR 442, SEVERITY 2
THE SCALAR VARIABLE $ IS DECLARED WITH AN 'INITIAL' ATTRIBUTE THAT SPECIFIES MORE THAN ONE INITIAL VALUE.

ERROR 443, SEVERITY 2
*UNUSED*

ERROR 444, SEVERITY 3
SYNTAX ERROR IN CALL STATEMENT.

ERROR 445, SEVERITY 3
THE KEYWORD "CALL" IS NOT FOLLOWED BY AN IDENTIFIER IN A CALL STATEMENT.

ERROR 446, SEVERITY 3
SYNTAX ERROR IN GOTO STATEMENT.

ERROR 447, SEVERITY 3
SYNTAX ERROR IN RETURN STATEMENT.

ERROR 448, SEVERITY 2
*UNUSED*

ERROR 449, SEVERITY 2
*UNUSED*

ERROR 450, SEVERITY 3
A SET-OPTION APPEARS MORE THAN ONCE IN AN ALLOCATE STATEMENT.

ERROR 451, SEVERITY 3
A LEFT PARENTHESIS IS MISSING IN A SET-OPTION OF AN ALLOCATE STATEMENT.

ERROR 452, SEVERITY 3
AN IN-OPTION APPEARS MORE THAN ONCE IN AN ALLOCATE STATEMENT.

ERROR 453, SEVERITY 3
A LEFT PARENTHESIS IS MISSING IN AN IN-OPTION OF AN ALLOCATE STATEMENT.

ERROR 454, SEVERITY 3
SYNTAX ERROR IN AN ALLOCATE STATEMENT.

ERROR 455, SEVERITY 3
A LEFT PARENTHESIS IS MISSING IN A FREE STATEMENT.

ERROR 456, SEVERITY 3
SYNTAX ERROR IN FREE STATEMENT.
```

```
ERROR 457, SEVERITY 3
SYNTAX ERROR IN THE CHARACTER PICTURE DECLARED FOR $.

ERROR 458, SEVERITY 3
SYNTAX ERROR IN THE FLOATING-POINT PICTURE DECLARED FOR $.

ERROR 459, SEVERITY 3
SYNTAX ERROR IN THE FIXED-POINT PICTURE DECLARED FOR $.

ERROR 460, SEVERITY 2
IMPLEMENTATION RESTRICTION: THIS STATEMENT IS NOT ALLOWED BY THIS VERSION OF THE COMPILER.

ERROR 461, SEVERITY 3
THE VARIABLE $ TO BE ALLOCATED BY A LOCATE STATEMENT MUST BE UNSUBSCRIPTED AND UNQUALIFIED.

ERROR 462, SEVERITY 3
THE REFERENCE $ APPEARING IN A FILE-OPTION OR A COPY-OPTION MUST BE OF TYPE FILE.

ERROR 463, SEVERITY 3
THE VARIABLE $ APPEARING IN A SET-OPTION MUST BE A POINTER VARIABLE.

ERROR 464, SEVERITY 3
THE VARIABLE $ IN A KEYTO-OPTION MUST BE A CHARACTER-STRING VARIABLE.

ERROR 465, SEVERITY 3
THE VARIABLE $ APPEARING IN A LOCATE STATEMENT MUST BE OF LEVEL-ONE.

ERROR 466, SEVERITY 3
THE STRING-OPTION OF A PUT STATEMENT REQUIRES A CHARACTER-STRING VARIABLE.

ERROR 467, SEVERITY 3
COMPILER ERROR: IO-SEMANTICS TABLE SIZE EXCEEDED. RETAIN OUTPUT AND CONTACT THE COMPILER MAINTENANCE PERSONNEL.

ERROR 468, SEVERITY 3
THE LOCATE STATEMENT REQUIRES THAT A POINTER VARIABLE BE ASSOCIATED WITH THE VARIABLE TO BE ALLOCATED EITHER BY A
SET-OPTION OR IMPLICITLY, VIA THE DECLARATION OF THE VARIABLE.

ERROR 469, SEVERITY 3
THE ELEMENT $ OF A GET DATA STATEMENT'S LIST MUST BE AN UNQUALIFIED, UNSUBSCRIPTED VARIABLE.

ERROR 470, SEVERITY 3
THE ELEMENT $ OF A GET DATA STATEMENT'S LIST MUST BE A STRING OR A NUMERIC SCALAR VARIABLE OR AN AGGREGATE CONSISTING
ONLY OF SUCH ELEMENTS.

ERROR 471, SEVERITY 3
THE ELEMENT $ APPEARING IN THE DATA LIST OF A PUT DATA OR GET STATEMENT MUST BE A VARIABLE.

ERROR 472, SEVERITY 3
THE ELEMENT $ APPEARING IN THE DATA LIST OF A STREAM I/O STATEMENT MUST BE OF ARITHMETIC OR STRING TYPE.
```

```
ERROR 473, SEVERITY 3
THE DATA LIST OF A GET DATA STATEMENT MAY NOT CONTAIN AN EXPRESSION.

ERROR 474, SEVERITY 3
THE DATA LIST OF A PUT DATA OR OF A GET STATEMENT MAY NOT CONTAIN AN EXPRESSION.

ERROR 475, SEVERITY 3
THE DATA LIST OF A STREAM I/O STATEMENT CONTAINS AN EXPRESSION OF OTHER THAN STRING OR ARITHMETIC TYPE.

ERROR 476, SEVERITY 2
*UNUSED*

ERROR 477, SEVERITY 2
*UNUSED*

ERROR 478, SEVERITY 2
*UNUSED*

ERROR 479, SEVERITY 2
*UNUSED*

ERROR 480, SEVERITY 2
*UNUSED*

ERROR 481, SEVERITY 2
*UNUSED*

ERROR 482, SEVERITY 2
*UNUSED*

ERROR 483, SEVERITY 2
*UNUSED*

ERROR 484, SEVERITY 2
*UNUSED*

ERROR 485, SEVERITY 2
*UNUSED*

ERROR 486, SEVERITY 2
*UNUSED*

ERROR 487, SEVERITY 2
*UNUSED*

ERROR 488, SEVERITY 2
*UNUSED*

ERROR 489, SEVERITY 2
*UNUSED*
```

```
ERROR 490, SEVERITY 2
*UNUSED*

ERROR 491, SEVERITY 2
*UNUSED*

ERROR 492, SEVERITY 3
A NON-CONDITION NAME HAS BEEN USED IN A CONTEXT WHICH REQUIRES A CONDITION NAME.

ERROR 493, SEVERITY 2
THE DECLARATION OF $ CONFLICTS WITH DIFFERENT DECLARATIONS OF SAME EXTERNAL NAME.

ERROR 494, SEVERITY 3
MULTIPLE "MAIN" OPTIONS HAVE BEEN SPECIFIED.

WARNING 495
IMPLEMENTATION RESTRICTION: LONG EXTERNAL NAME $ HAS BEEN CONVERTED TO A 6 CHARACTER NAME. RESTRICTIONS ARE: EXTERNAL
FILE NAME SIZE <= 5 AND OTHER EXTERNAL NAME SIZE <= 6.

ERROR 496, SEVERITY 2
*UNUSED*

ERROR 497, SEVERITY 2
*UNUSED*

ERROR 498, SEVERITY 2
*UNUSED*

ERROR 499, SEVERITY 2
*UNUSED*

WARNING 500
$ IS AN ATTRIBUTE WHICH IS NOT CURRENTLY DEFINED IN THE NUCLEUS OF THPL.

WARNING 501
THIS STATEMENT IS NOT CURRENTLY DEFINED IN THE NUCLEUS OF THPL.

ERROR 502, SEVERITY 2
*UNUSED*
```

APPENDIX I

ON-CODES

This section contains the meaning of the ON-code numbers printed as a result of the detection of an error at execution time.

The ON-code meanings are listed in the order of the associated number. For each ON-code number, the condition name and the meaning are given.

ONCODE	CONDITION NAME	MEANING

1	ERROR	IEXP←: EXPONENTIATIƆN ERROR 0**0.
		SET RESULT = 0.

2	ERROR	IEXP←: EXPONENTIATIƆN ERROR 0**(-J).
		SET RESULT = 0.

3	ERROR	DXP1←: EXPONENTIATIƆN ERROR 0**0.
		SET RESULT = 0.

4	ERROR	DXP1←: EXPONENTIATIƆN ERROR 0**(-J).
		SET RESULT = 0.

5	ERROR	EXP←: EXP(X), X<-88.028E0 NOT ALLOWED,
		SET RESULT = 0.

8	ERROR	EXP←: EXP(X), X>88.028 NOT ALLOWED.
		SET RESULT = OMEGA.

9	ERROR	LOG←: LOG(0),LOG2(0), OR LOG10(0) NOT ALLOWED.
		SET RESULT = - OMEGA.

10	ERROR	ALOG←: ALOG(-X),ALOG2(-X), OR ALOG10(-X) NOT ALLOWED. EVALUATE FOR +X.

11	ERROR	ATAN2←: ATAN2(0,0) NOT ALLOWED.
		SET RESULT = 0.

12	ERROR	SIN←: SIN(X), COS(X), SIND(X), OR COSD(X),!X!/ >= 2**27 NOT ALLOWED.
		SET RESULT = 0.

13	ERROR	SQRT←: SQRT(-X) NOT ALLOWED. EVALUATE FOR +X.

14	ERROR	DDCXP1←: EXPONENTIATION ERROR (0+0I)**0.
		SET RESULT = 0.15(5)CXP1←: EXPONENTIATION ERROR (0+0I)**(-J).
		SET RESULT = 0.

16	ERROR	DXP2←: EXPONENTIATIƆN ERROR (-A)**B. EVALUATE FOR +A.

17	ERROR	DXP2←: EXPONENTIATIƆN ERROR 0**0.
		SET RESULT = 0.

18	ERROR	DXP2←: EXPONENTIATIƆN ERROR 0**(-B).
		SET RESULT = 0.

19	ERROR	DEXP←: EXP(X), X>88.028 NOT ALLOWED.
		SET RESULT = OMEGA.

20	ERROR	DLOG←: LOG(0), LOG2(0), OR LOG10(0) NOT ALLOWED.
		SET RESULT = - OMEGA.

21	ERROR	DLOG←: LOG(-X), LOG2(-X), OR LOG10(-X) NOT ALLOWED. EVALUATE FOR +X.
22	ERROR	DSQRT←: SQRT(-X) NOT ALLOWED. EVALUATE FOR +X.

23	ERROR	DSIN←: SIN(X), COS(X), SIND(X), OR COSD(X), !X! >= 2**54 NOT ALLOWED.
		SET RESULT = 0.

24	ERROR	DATAN2←: ATAN2(0,0) NOT ALLOWED.
		SET RESULT = 0.

```
25   ERROR            DCFDP←: DIVISION ERROR (X+IY)/(0+0I) NOT ALLOWED.
                      SET RESULT = OMEGA * (1+I).
26   ERROR            CEXP←: EXP(X+IY), X > 88.028 NOT ALLOWED.
                      PROCEED WITH E**X = OMEGA.
27   ERROR            CEXP←: EXP(X+IY), !y! >= 2**27 NOT ALLOWED.
                      SET RESULT = ZERO.
28   ERROR            CLOG←: LOG(0+0I) NOT ALLOWED.
                      SET RESULT = - OMEGA.
29   ERROR            CSIN←: N(X+IY) OR NH(Y+IX), !X! >= 2**27 NOT ALLOWED. N = SIN, COS, TAN.
                      SET RESULT = 0.
30   ERROR            CSIN←: N(X+IY) OR NH(Y+IX), !Y! > 88.028 NOT ALLOWED. N = SIN, COS, TAN.
                      SET RESULT = 0.
31   ERROR            DCLOG←: LOG(0+0I) NOT ALLOWED.
                      SET RESULT = - OMEGA.
32   ERROR            DCATAN←: ATAN(0+IX) OR ATANH(X+0I), !X! = 1 NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
33   ERROR            TAN←: TAN(X), !X! >= 2**27 NOT ALLOWED.
                      SET RESULT = 0.
34   ERROR            ASINH←: ACOSH(X), !X! < 1 NOT ALLOWED.
                      SET RESULT = 0.
35   ERROR            DASINH←: ACOSH(X), !X! < 1 NOT ALLOWED.
                      SET RESULT = 0.
36   ERROR            CSIN←: TAN(X+IY) OR TANH(X+IY), X+IY TOO CLOSE TO SINGULARITY NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
39   ERROR            SINH←: SINH(X) OR COSH(X), !X! > 88.028 NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
41   ERROR            EXERFC←: EXERFC(X), X < -9.30630096 NOT ALLOWED.
                      SET RESULT = OMEGA.
42   ERROR            LOG←: ATANH(X), !X! = 1 NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
43   ERROR            LOG←: ATANH(X), !X! > 1 NOT ALLOWED.
                      SET RESULT = 0.
44   ERROR            DLOG←: ATANH(X), !X! = 1 NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
45   ERROR            DLOG←: ATANH(X), !X! > 1 NOT ALLOWED.
                      SET RESULT = 0.
46   ERROR            TAN←: TAN(X), X TOO CLOSE TO SINGULARITY NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
47   ERROR            DTAN←: TAN(X), X TOO CLOSE TO SINGULARITY NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
50   ERROR            DSINH←: SINH(X) OR COSH(X), !X! > 88.028 NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
57   ERROR            DCXP2←: EXPONENTIATION ERROR (0+0I)**(X+IY).
                      SET RESULT = 0.
58   ERROR            ASIN←: ASIN(X), ACOS(X), ASIND(X), OR ACOSD(X), !X! > 1 NOT ALLOWED.
                      SET RESULT = 0.
59   ERROR            CATAN←: ATAN(0+IX) OR ATANH(X+0I), !X! = 1 NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
60   ERROR            CXP2←: EXPONENTIATION ERROR (0+0I)**(X+IY).
                      SET RESULT = 0.
61   ERROR            DCSIN←: N(X+IY) OR NH(Y+IX), !X! >= 2**27 NOT ALLOWED. N = SIN, COS, TAN.
                      SET RESULT = 0.
```

```
 62  ERROR            DCSIN+: N(X+IY) OR NH(Y+IX), !X! >= 2**54 NOT ALLOWED. N = SIN, COS, TAN.
                      SET RESULT = 0.
 63  ERROR            DTAN+: TAN(X), !X! >= 2**54 NOT ALLOWED.
                      SET RESULT = 0.
 64  ERROR            DCSIN+: TAN(X+IY) OR TANH(X+IY), X+IY TOO CLOSE TO SINGULARITY NOT ALLOWED.
                      SET RESULT = + OR - OMEGA.
 65  ERROR            DASIN+: ASIN(X), ACOS(X), ASIND(X), OR ACOSD(X), !X! > 1 NOT ALLOWED.
                      SET RESULT = 0.
 66  ERROR            DEXERFC+: EXERFC(X), X < -9.30630096 NOT ALLOWED.
                      SET RESULT = OMEGA.
 68  ERROR            DCEXP+: EXP(X+IY), X > 88.028 NOT ALLOWED.
                      PROCEED WITH E**X = OMEGA.
 69  ERROR            DCEXP+: EXP(X+IY), !Y! >= 2**54 NOT ALLOWED.
                      SET RESULT = 0.
105  KEY              DELETE,READ,REWRITE: THE RECORD SPECIFIED BY THE KEY VALUE OF AN INDEXED
                      FILE CANNOT BE FOUND.
106  ENDFILE          READ: ENDFILE
107  UNDEFINEDFILE    OPEN: $DATA RECORD DESCRIBING RECORD:
                         1.  CONTAINS DBIT ARGUMENT BUT ORGANIZATION IS
                             REGINONAL
                         2.  IMPROPER DBIT ARGUMENT
108  UNDEFINEDFILE    OPEN: BACKWARDS TAPE FILES NOT CURRENTLY IMPLEMENTED
109  KEY              LOCATE,WRITE: THE KEY VALUE IS NOT GREATER THAN THE PREVIOUS KEY
                      VALUE FOR A FILE WITH THE SEQUENTIAL AND OUTPUT
                      ATTRIBUTES.
110  KEY              WRITE: THE KEY VALUE IDENTIFIES A RECORD WHITCH ALREADY EXISTS
                      ON A FILE WITH THE INDEXED, DIRECT AND UPDATE ATTRIBUTES.
112  ERROR            PLIO: SOFTWARE ERROR IN PL/I RECORD RUNTIME PROGRAM.
113  TRANSMIT         GFRC: BLANK TAPE RATHER THAN INPUT HEADER LABEL.
114  TRANSMIT         GFRC: ERROR IN INPUT TAPE HEADER LABEL ON TAPE.
115  TRANSMIT         GFRC: ERROR IN OUTPUT TAPE HEADER LABEL.
116  TRANSMIT         GFRC: ERROR IN OLD LABEL ON OUTPUT TAPE.
117  TRANSMIT         GFRC: END+OF+TAPE WHILE WRITING HEADER LABEL.
118  TRANSMIT         GFRC: BLANK TAPE RATHER THAN INPUT TRAILER LABEL.
119  TRANSMIT         GFRC: BLOCK COUNT ERROR IN INPUT TAPE TRAILER LABEL.
120  TRANSMIT         GFRC: END+OF+TAPE DETECTED ON MULTIFILE REEL.
                      (NOTE: A MULTIFILE REEL CANNOT BE CONTAINED ON A SECONDARY TAPE).
121  TRANSMIT         GFRC: ATTEMPT TO ACCESS TELETYPE THROUGH GFRC  WITHOUT
                      HAVING LOADED PROPER ROUTINE,; PTYP OPTION ENTERED AS
                      SYMREF ON THE $ USE CARD.
122  UNDEFINEDFILE    BSREC: ILLEGAL REQUEST FOR THIS ROUTINE.
123  UNDEFINEDFILE    BSREC: STATUS NOT TAPE ON LOAD POINT OR OK.
124  UNDEFINEDFILE    BSTFM: ILLEGAL REQUEST FOR THIS ROUTINE.
125  UNDEFINEDFILE    BSTFM: REACHED LOAD POINT BUT MORE FILES TO SKIP.
126  UNDEFINEDFILE    CLOSE: EOF STATUS ON OUTPUT FILE.
127  UNDEFINEDFILE    CLOSE: UNRECOVERABLE I/O ERROR, NO USER ROUTINE.
128  UNDEFINEDFILE    CLOSE; FILE TO BE CLOSED IS NOT IN CHAIN.
129  UNDEFINEDFILE    CLOSE; ILLEGAL STATUS FOR DISC OR DRUM.
131  UNDEFINEDFILE    FORCT; ILLEGAL REQUEST FOR THIS ROUTINE.
132  UNDEFINEDFILE    FSREC; FILE IS NOT PRESENT.
133  UNDEFINEDFILE    FSREC: EOF ON DEVICE FROM PRIOR COMMAND.
134  UNDEFINEDFILE    FSREC: IMPOSSIBLE RETURN FROM SYSTEM ROUTINE.
```

```
135  UNDEFINEDFILE   FSREC: ILLEGAL FILE DEFINITION IN FCB.
136  UNDEFINEDFILE   FSREC: UNRECOVERABLE I/O ERROR, NO USER ROUTINE.
137  UNDEFINEDFILE   FSREC: ILLEGAL REQUEST FOR THIS ROUTINE.
138  UNDEFINEDFILE   FSREC: I/O STATUS OTHER THAN BLANK TAPE ON READ.
139  ERROR           FSTFM: ILLEGAL REQUEST FOR THIS ROUTINE.
140  UNDEFINEDFILE   GET: FILE DESIGNATED AS OUTPUT FILE.
141  UNDEFINEDFILE   GET: ILLEGAL FILE DEFINITION IN FCB.
142  TRANSMIT        GET: UNRECOVERABLE I/O ERROR, NO USER ROUTINE.
143  TRANSMIT        GET: BLOCK SERIAL NUMBER DO NOT AGREE.
144  TRANSMIT        GET: FIXED OR MIXED REC. SIZE FOR A FILE IS ZERO,
                     OR VAR. REC. SIZE IS ZERO FOR FOR TAPE FILE.
145  TRANSMIT        GF200: EOF STATUS ON OUTPUT FILE.
146  TRANSMIT        GF200: UNRECOVERABLE I/O ERROR, NO USER ROUTINE.
148  ERROR           GF980: TRIED TO CREATE AN ILLEGAL I/O REQUEST.
149  UNDEFINEDFILE   OPEN: ILLEGAL DEVICE CODE FOR GFRC.
150  UNDEFINEDFILE   OPEN: FILE IS LOCKED.
151  UNDEFINEDFILE   OPEN: DEVICE IS PRINTER OR PUNCH, NOT OUTPUT FILE.
152  UNDEFINEDFILE   OPEN: ILLEGAL DISC OR DRUM FORMAT.
153  UNDEFINEDFILE   OPEN: LINKED FILE BUT NOT VARIABLE RECORD TYPE.
154  UNDEFINEDFILE   OPEN: ILLEGAL FORMAT FOR SYSOUT FILE.
155  UNDEFINEDFILE   OPEN: FILE DESIGNATED AS REQUIRED IS NOT PRESENT.
156  UNDEFINEDFILE   OPEN: TWO FILE DESIGNATORS POINTING TO SAME FILE.
157  UNDEFINEDFILE   PUT: EOF STATUS ON OUTPUT FILE.
158  TRANSMIT        PUT: UNRECOVERABLE I/O ERROR.
159  UNDEFINEDFILE   PUT: ILLEGAL FILE DEFINITION IN FCB.
160  TRANSMIT        PUT: CURRENT LOGICAL RECORD LARGER THAN BUFFER.
161  ERROR           PUTSZ: ILLEGAL REQUEST FOR THIS ROUTINE.
162  TRANSMIT        PUTSZ: NEW SIZE LARGER THAN OLD RECORD SIZE.
163  TRANSMIT        RDREC: BINARY CARD IS NOT A COMDK CARD.
164  TRANSMIT        RDREC: THE COMDK CARD ARE OUT OF SEQUENCE.
165  TRANSMIT        RDREC: DATA ERROR IN DECOMPRESSING COMDK CARD.
166  UNDEFINEDFILE   READ: ILLEGAL FILE DEFINITION IN FCB.
167  UNDEFINEDFILE   REMOT: REMOTE TERMINAL OUTPUT FILE HAS NO BUFFER.
168  UNDEFINEDFILE   REWND: ILLEGAL REQUEST FOR THIS ROUTINE.
169  UNDEFINEDFILE   USERR: ROUTINE CALLING ERROR WAS USED BY SAME.
170  TRANSMIT        WAIT: UNRECOVERABLE I/O ERROR, NO USER ROUTINE.
171  TRANSMIT        WEF: EOF STATUS ON OUTPUT FILE.
172  TRANSMIT        WEF: UNRECOVERABLE I/O ERROR, NO USER ROUTINE.
173  TRANSMIT        WEF: ILLEGAL REQUEST FOR THIS ROUTINE.
174  TRANSMIT        WEF: ILLEGAL STATUS FOR DISC, DRUM OR TAPE.
175  UNDEFINEDFILE   WRITE: FILE IS NOT PRESENT.
176  TRANSMIT        WRITE: EOF STATUS ON OUTPUT FILE.
177  UNDEFINEDFILE   WRITE: ILLEGAL FILE DEFINITION IN FCB.
178  UNDEFINEDFILE   ISP: FILE OPENED PREVIOUSLY OR AN ACCESS WAS MADE TO A FILE THAT IS NOT
                     OPEN.
179  UNDEFINEDFILE   ISP: FILE NOT OPENED PROPERLY. ATTEMPTING TO CREATE A FILE WITHOUT FIRST
                     OPENING WITH NOPEN.
180  TRANSMIT        ISP: PHYSICAL READ OR WRITE ERROR.
181  UNDEFINEDFILE   ISP: INSUFFICIENT BUFFER STORAGE, CANNOT CREATE THE REQUIRED MINIMUM OF
                     3 BUFFERS. INCREASE THE AMOUNT OF CORE ALLOCATED ON $ LIMITS CARD,
                     OR THE AMOUNT OF LABELED COMMON STORAGE.
182  UNDEFINEDFILE   ISP: INSUFFICIENT DEVICE STORAGE FOR INDEX FILE. INCREASE THE INDEX FILE
                     SIZE AND RERUN JOB.
```

```
183  UNDEFINEDFILE    ISP: INSUFFICIENT DEVICE STORAGE FOR DATA FILE. COULD BE CAUSED AT FILE
                      INITIALIZATION TIME, OR BY ADDING OVERFLOW RECORDS TO THE FILE. THE
                      FILE IS FILLED TO CAPACITY AND USER ACTION IS REQUIRED.
184  UNDEFINEDFILE    ISP: RECORD AND/OR KEY FIELD DESCRIBED BY IOPEN DO NOT AGREE WITH RECORD
                      AND/OR KEY FIELD DESCRIPTION PROVIDED BY NOPEN WHEN THE FILE WAS
                      CREATED.
185  UNDEFINEDFILE    ISP: FILE ACCESSED IS NOT THE DATA FILE.
189  UNDEFINEDFILE    ISP: NO COARSE OR NO FINE INDEX PAGE FOUND. FILE HAS MOST LIKELY BEEN
                      DESTROYED.
190  UNDEFINEDFILE    ISP: RECORD SIZE GREATER THAN 255 WORDS.
191  UNDEFINEDFILE    ISP: PAGE-SIZE SPECIFIED IS LARGER THEN ALLOWED. THE FILE USING THE
                      LARGEST PAGE-SIZE MUST BE OPENED FIRST.
192  UNDEFINEDFILE    ISP: PAGE-SIZE DESCRIBED BY IOPEN DOES NOT AGREE WITH PAGE-SIZE PROVIDED
                      WHEN THE FILE WAS CREATED.
193  UNDEFINEDFILE    ISP: NO DISC SPACE ALLOCATED TO EITHER THE INDEX FILE OR DATA FILE.
                      ALSO OCCURS IF THE FILE CODE IS NOT PROPERLY DEFINED.
194  UNDEFINEDFILE    OPEN: A PREVIOUS OPEN STATEMENT, EITHER EXPLICIT OR IMPLICIT,
                      HAS CONNECTED A FILE TO THIS TITLE VALUE.
195  UNDEFINEDFILE    OPEN: NO FILE CONTROL BLOCK CREATED BY THE LOADER.
196  UNDEFINEDFILE    OPEN: THE TITLE VALUE SPECIFIED FOR AN INDEXED OR REGIONAL
                      DOES NOT REFER TO A VALID $ DATA FILE.
197  UNDEFINEDFILE    OPEN: NO WORKING AREA OR BUFFERS ESTABLISHED FOR INDEXED
                      OR REGIONAL FILE. $ USE CARD MISSING OR IMPROPER.
198  UNDEFINEDFILE    OPEN: MIXED LENGTH RECORDS(MIXLNG) SPECIFIED ON $ FFILE.
199  UNDEFINEDFILE    OPEN: THE RECORD LENGTH SPECIFIED BY THE FIXLNG OPTION OF
                      THE $ FFILE DOES NOT EQUAL THE LENGTH SPECIFIED BY THE
                      RECSZ OPTION OF THE $ DATA FILE.
200  UNDEFINEDFILE    OPEN: THE BUFFER SIZE SPECIFIED BY THE BUFSIZ OPTION OF THE
                      $ FFILE IS NOT LARGE ENOUGH TO CONTAIN THE LARGEST
                      RECORD.
201  UNDEFINEDFILE    OPEN: ERROR ANALYSIS ROUTINE SPECIFIED BY THE ERRXIT OPTION
                      OF THE $ FFILE CARD. USE ONCODE FUNCTION INSTEAD.
202  UNDEFINEDFILE    OPEN: MIXED LENGTH RECORDS MAY NOT BE SPECIFIED BY THE
                      MIXLNG OPTION OF THE $ FFILE CARD.
203  UNDEFINEDFILE    OPEN: PARTITIONED RECORDS MAY NOT BE SPECIFIED BY THE PRTREC
                      OPTION OF THE $ FFILE CARD.
204  UNDEFINEDFILE    OPEN: AT LEAST ONE BUFFER MUST BE SPECIFIED FOR A CONSECUTIVE
                      FILE. ONLY ONE MAY BE SPECIFIED FOR AN UPDATE OR
                      BACKWARDS FILE.
205  UNDEFINEDFILE    OPEN: THE TYPE OF TRANSMISSION(RECORD/STREAM) OPENING ATTRIBUTE
                      CONFLICTS WITH THE TYPE OF TRANSMISSION ATTRIBUTE OF THE DECLARE
                      STATEMENT FOR THE FILE.
206  UNDEFINEDFILE    OPEN: THE ENVIRONMENT OPENING ATTRIBUTE CONFLICTS WITH THE
                      ENVIRONMENT ATTRIBUTE OF THE DECLARE STATEMENT FOR THE
                      FILE.
207  UNDEFINEDFILE    OPEN: A CONFLICT EXISTS BETWEEN THE FUNCTION (INPUT/OUTPUT/
                      UPDATE) OPENING ATTRIBUTE AND THE FUNCTION ATTRIBUTE
                      OF THE DECLARE STATEMENT FOR THE FILE.
208  UNDEFINEDFILE    OPEN: A CONFLICT EXISTS BETWEEN THE ACCESS(DIRECT, SEQUENTIAL)
                      OPENING ATTRIBUTE AND THE ACCESS ATTRIBUTE OF THE DECLARE
                      STATEMENT FOR THE FILE.
```

```
209  UNDEFINEDFILE   OPEN: A CONFLICT EXISTS BETWEEN THE ORDER(FORWARDS, BACKWARDS)
                     OPENING ATTRIBUTE AND THE ORDER ATTRIBUTE OF THE DECLARE
                     STATEMENT FOR THE FILE.
210  UNDEFINEDFILE   OPEN: INCONSISTENT FILE DESCRIPTION ATTRIBUTES FOR A RECORD
                     SEQUENTIAL FILE.
211  UNDEFINEDFILE   OPEN: INCONSISTENT FILE DESCRIPTION ATTRIBUTE FOR A RECORD
                     DIRECT FILE.
212  UNDEFINEDFILE   OPEN: INCONSISTENT FILE DESCRIPTION ATTRIBUTES FOR A STREAM
                     FILE.
213  UNDEFINEDFILE   OPEN: THE PAGESIZE OPENING OPTION IS SPECIFIED FOR A FILE
                     WHICH DOES NOT CONTAIN THE PRINT ATTRIBUTE.
214  UNDEFINEDFILE   OPEN: THE LINESIZE OPENING OPTION IS SPECIFIED FOR A FILE
                     WHICH DOES NOT CONTAIN THE ATTRIBUTES STREAM AND OUTPUT.
215  UNDEFINEDFILE   OPEN: THE TITLE VALUE SPECIFIED FOR CONSECUTIVE FILE DOES NOT
                     REFER TO A VALID $ DATA FILE AND A FILE CONTROL BLOCK
                     FOR THE TITLE VALUE HAS NOT BEEN CREATED BY THE LOADER.
216  UNDEFINEDFILE   OPEN: THE MAXIMUM OF 59 DATA SETS PER ACTIVITY WITH THE
                     PRINT ATTRIBUTE HAS BEEN EXCEEDED.
217  UNDEFINEDFILE   OPEN: THE VALUE OF EXPRESSION OF THE PAGESIZE OPENING
                     OPTION IS LESS THAN 1.
218  UNDEFINEDFILE   OPEN: THE VALUE OF EXPRESSION OF THE LINESIZE OPENING
                     OPTION IS LESS THAN 1.
219  UNDEFINEDFILE   OPEN: $ DATA RECORD DOES NOT CONTAIN PROPER IDENTIFICATION
                     IN COLUMN 1-7.
220  UNDEFINEDFILE   OPEN: $ DATA RECORD DOES NOT CONTAIN PROPER IDENTIFICATION
                     IN COLUMN 8-15.
221  UNDEFINEDFILE   OPEN: $ DATA RECORD DESCRIBING DATA FILE CONTAINS AN IMPROPER
                     ARGUMENT OR IS MISSING A PROPER FC ARGUMENT.
222  UNDEFINEDFILE   OPEN:$ DATA FILE ILLEGALLY CONTAINS MORE THAN ONE DATA OR
                     RECORD DESCRIPTION.
223  UNDEFINEDFILE   OPEN: $ DATA DESCRIBING RECORD CONTAINS AN IMPROPER
                     ARGUMENT, OR IS MISSING A PROPER RECSZ ARGUMENT.
224  UNDEFINEDFILE   OPEN: $ DATA RECORD DESCRIBING RECORD:
                       1.  CONTAINS KEYSZ AND/OR KEYOFF BUT ORGANIZATION
                           IS NOT INDEXED.
                       2.  KEYSZ OR KEYOFF MISSING OR IMPROPER.
                       3.  THE KEY IS NOT CONTAINED WITHIN THE RECORD.
                       4.  THE VALUES CANNOT BE CONVERTED TO AN EQUIVALENT
                           ASCII SIZE OR OFFSET.
                       5.  THE KEYSZ IS ZERO OR GREATER THAN 32 ASCII
                           CHARACTERS.
225  UNDEFINEDFILE   OPEN: THE DEVICE ASSIGNED TO THIS FILE IS NOT CAPABLE OF
                     PERFORMING THE FUNCTIONALITY DEFINED BY THE FILE
                     ATTRIBUTES.
226  UNDEFINEDFILE   OPEN: THE RECORD SIZE IS LESS THAN ONE CHARACTER.
227  UNDEFINEDFILE   OPEN: THE TITLE VALUE REFERS TO SYSPRINT, BUT THE OUTPUT
                     ATTRIBUTE IS NOT SPECIFIED. THE TITLE VALUE REFERS
                     TO SYSIN, BUT THE INPUT ATTRIBUTE IS NOT SPECIFIED.
228  UNDEFINEDFILE   CLOSE-BY-NAME: THE FILE STATE BLOCK TO BE CLOSED, IDENTIFIED BY FILE
                     NAME, IS EITHER NON-EXISTENT OR THE FILE HAS NEVER BEEN
                     OPENED.
229  UNDEFINEDFILE   REC-IO: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE PROHIBIT
                     THE EXECUTION OF THIS STATEMENT.
```

230	KEY	REC←IO: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE PROHIBIT THE USAGE OF THE KEY OPTION IN THIS STATEMENT.
231	KEY	DELETE,REWRITE: THE RECORD TO BE OPERATED ON CANNNOT BE DETERMINED. NO KEY OPTION IS SPECIFIED AND NO CURRENT RECORD IS ESTABLISHED.
232	KEY	LOCATE,WRITE: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE PROHIBIT THE USAGE OF THE KEYFROM OPTION IN THIS STATEMENT.
233	KEY	LOCATE,WRITE: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE REQUIRE THE USAGE OF THE KEYFROM OPTION IN THIS STATEMENT.
234	KEY	REC←IO: THE VALUE SPECIFIED FOR THE REGIONAL KEY IS NOT WITHIN THE RANGE OF ACCEPTABLE KEY VALUES.
235	RECORD	LOCATE,READ,WRITE,REWRITE: THE SIZE OF THE RECORD VARIABLE IS SMALLER THAN THE EXPECTED ACTUAL OR MAXIMUM RECORD SIZE.
236	RECORD	LOCATE,READ,WRITE,REWRITE: THE SIZE OF THE RECORD VARIABLE IS LARGER THAN THE EXPECTED ACTUAL OR MAXIMUM RECORD SIZE.
237	RECORD	LOCATE,READ,WRITE,REWRITE: THE SIZE OF THE RECORD VARIABLE IS LESS THAN OR EQUAL TO ZERO.
238	ERROR	READ: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE PROHIBIT THE USAGE OF THE IGNORE OPTION IN THE READ STATEMENT.
239	ERROR	READ: THE VALUE OF THE EXPRESSION IN THE IGNORE OPTION IS NOT GREATER THAN ZERO.
240	ERROR	READ,REWRITE: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE REQUIRE THE USAGE OF THE KEY OPTION IN THIS STATEMENT.
241	ERROR	READ: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE PROHIBIT THE USAGE OF THE KEYTO OPTION IN A READ STATEMENT.
242	ERROR	READ: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE PROHIBIT THE USAGE OF THE SET OPTION IN THIS READ STATEMENT,
243	ERROR	WRITE: THE ATTRIBUTES CURRENTLY DEFINED FOR THE FILE REQUIRE THE USAGE OF THE FORM OPTION IN THIS STATEMENT.
244	ERROR	REWRITE: THE FROM OPTION IS NOT SPECIFIED FOR A REWRITE STATEMENT AND THE PREVIOUS INPUT OPERATION ON THIS FILE WAS NOT A READ STATEMENT CONTAINING A SET OPTION.
245	KEY	WRITE,LOCATE: THE KEY VALUE IN THE KEYFROM OPTION DOESN'T MATCH WITH THE KEY EMBEDDED IN THE RECORD TO THE INDEXED ORGANIZATION.
250	UNDEFINEDFILE	OPEN: $ DATA RECORD DESCRIBING DATA FILE CONTAINS ILLEGAL OPTION FOR THIS FILE.
251	UNDEFINEDFILE	OPEN: $ DATA RECORD DESCRIBING DATA FILE CONTAINS ILLEGAL TAB OPTION.
252	UNDEFINEDFILE	OPEN: $ DATA RECORD DESCRIBING RECORD ILLEGALLY CONTAINS MORE THAN ONE RECSZ AND/OR CHARSZ OPTION.
315	ERROR	DECIMAL←OP: THE SECOND ARGUMENT OF ROUND BUILTIN FUNCTION HAS ILLEGAL VALUE.
316	ERROR	SIGNAL: CONVERSION←ERROR SIGNALLER INCORECTLY CALLED INTERNALLY BY PLIO.
317	ERROR	SIGNAL: ILLEGAL ERROR MESSAGE NUMBER.
318	ERROR	SIGNAL: NO ERROR MESSAGE WITH THIS NUMBER EXIST.
319	ERROR	SIGNAL: SYNTAX ERROR IN FILE PLIO←MESSAGE.
320	ZERODIVIDE	DECIMALOP: ATTEMPT TO DIVIDE BY ZERO.
321	FIXEDOVERFLOW	DECIMALOP: INTERMEDIATE FIXED POINT VALUE OF PRECISION > 63 GENERATED.
322	ERROR	GET←DATA: ILLEGAL SYNTAX - DATA VALUE MISSING - COMMA OR SEMICOLON FOLLOWS EQUAL SIGN.
323	ERROR	GET←DATA: BNC>BLC IN SCAN OF DATA IDENTIFIER CONTACT PL/1 MAINTAINER.

324	ERROR	GET←DATA: MORE THAN 128 SUBSCRIPTS NOT ALLOWED IN THIS IMPLEMENTATION.
325	ERROR	GET←DATA: SEMICOLON APPEARS ILLEGALLY IN DATUM IDENTIFIER.
326	NAME	GET←DATA: MISCELLANEOUS CHARACTER APPEARS ILLEGALLY IN DATUM IDENTIFIER.
327	NAME	GET←DATA: NUMERIC CHARACTER APPEARS ILLEGALLY IN DATUM IDENTIFIER.
328	NAME	GET←DATA: SIGN CHARACTER APPEARS ILLEGALLY IN DATUM IDENTIFIER.
329	NAME	GET←DATA: COMMA APPEARS ILLEGALLY IN DATUM IDENTIFIER.
330	NAME	GET←DATA: EQUAL SIGN APPEARS ILLEGALLY IN DATUM IDENTIFIER.
331	NAME	GET←DATA: OPEN OR CLOSE PARENTHESIS CHARACTER APPEARS ILLEGALLY IN DATUM IDENTIFIER.
332	NAME	GET←DATA: UNDER←SCORE CHARACTER APPEARS ILLEGALLY IN DATUM IDENTIFIER.
333	NAME	GET←DATA; DOT('.') CHARACTER APPEARS ILLEGALLY IN DATUM IDENTIFIER.
334	NAME	GET←DATA: ALPHABETIC CHARACTER (OR '←' OR '$') APPEARS ILLEGALLY IN DATUM IDENTIFIER.
335	NAME	GET←DATA: IDENTIFIER NOT FOUND IN SYMBOL TABLE.
336	NAME	GET←DATA:IDENTIFIER NOT FOUND IN DATA LIST OF GET DATA STATEMENT.
337	NAME	GET←DATA; NUMBER OF SUBSCRIPTS IN IDENTIFIER DOES NOT MATCH NUMBER SPECIFIED IN SYMBOL TABLE.
338	ERROR	GET←DATA: ERROR RETURN MADE BY STU←$GET←ADDRESS. CONTACT PL/1 MAINTENANCE PERSONNEL.
339	ERROR	GET←DATA; ERROR RETJRN BY STU←DECODE←VALUE. CONTACT PL/1 MAINTENANCE PERSONNEL.
340	NAME	GET←DATA: SUBSCRIPT RANGE ERROR ACCORDING TO SYMBOL TABLE.
341	ERROR	BUILTIN: LINENO OR PAGENO BUILTIN FUNCTION REFERENCES A FILE THAT IS NOT OPEN.
342	ERROR	BUILTIN: LINENO OR PAGENO BUILTIN FUNCTION REFERENCES A FILE LACKING THE 'PRINT' ATTRIBUTE.
343	ERROR	STREAM: THIS IMPLEMENTATION WILL NOT EXTRACT A FIELD OF LENGTH > 256 CHARACTERS.
346	ERROR	GET←EDIT: ILLEGAL FORMAT ITEM(I.E., PAGE OR LINE).
347	ERROR	GET←EDIT; ILLEGAL NUMERIC FORMAT ITEM(I.E., NOT C OR F OR E). CONTACT PL/1 MAINTENANCE PERSONNEL.
348	ERROR	GET←EDIT: TOO FEW PARAMETERS IN FORMAT ITEM.
349	ERROR	GET←EDIT: THIS IMPLEMENTATION WILL NOT EXTRACT A FIELD OF LENGTH > 256 CHARACTERS.
350	CONVERSION	GET←EDIT; EXPONENT NOT ALLOWED WITH F-FORMAT.
351	CONVERSION	GET←EDIT: B-FORMAT FIELD IS ILLEGAL; IT ENDS IN SOMETHING OTHER THAN TRAILLING BLANKS.
352	CONVERSION	GET←EDIT: B-FORMAT FIELD CONTAINS NON-BLANK CHARACTER OTHER THAN '0' OR '1'.
353	TRANSMIT	GET: VALUE STORED MAY BE IN ERROR DUE TO PREVIOUS ERROR IN TRANSMISSION.
354	CONVERSION	GET←EDIT: ILLEGAL EXPONENTIAL FORMAT IN INPUT UNDER E-FORMAT.
355	CONVERSION	GET←EDIT; BINARY NUMBER REPRESENTATION (E.G., '10.1B') NOT ALLOWED UNDER E- OR F-FORMAT.

356	CONVERSION	GET-EDIT: IMAGINARY NUMBER REPRESENTATION (E.G., '5I') NOT ALLOWED UNDER E- OR F-FORMAT.
357	ERROR	GET-LIST: SEMICOLON IS ILLEGAL SEPARATOR.
358	SIZE	DECIMALOP: LOSS OF HIGH ORDER DIGIT(S).
359	OVERFLOW	DECIMALOP: EXPONENT > 127 GENERATED.
360	UNDERFLOW	DECIMALOP: EXPONENT < -128 GENERATED.
361	ERROR	GET-UTIL: 'GET EDIT' IN THIS IMPLEMENTATION CANNOT OBTAIN A FIELD OF WIDTH > 256 CHARACTERS.
362	ERROR	GET-UTIL: THE STRING SUPPLIED WITH STRING OPTION HAS INSUFFICIENT DATA FOR THIS 'GET' STATEMENT
363	ENDFILE	GET-UTIL: END-OF-FILE ENCOUNTERED WHILE EXECUTING 'GET' STATEMENT.
364	ENDFILE	GET-UTIL: ATTEMPT IS MADE TO READ FROM FILE ALREADY AT END-OF-FILE.
365	ERROR	GET-UTIL: A 'GET' STATEMENT REQUIRES A FILE WITH 'STREAM' AND 'INPUT' ATTRIBUTES.
366	ERROR	GET-UTIL: END OF THE DATA STREAM IS ENCOUNTERD DURING THE SCAN.
367	TRANSMIT	GET-UTIL: ACTUAL RECORD SIZE IS GREATER THAN MAX-RECORD SIZE.
368	CONVERSION	LDI: AN ALL BLANK FIELD MAY NOT REPLACE A NON-BLANK FIELD FOLLOWING CONVERSION ERROR.
370	CONVERSION	CTN: ILLEGAL FLOATING POINT FORMAT IN THIS FIELD.
376	CONVERSION	CTX: REPRESENTATION OF BIT STRING OR BINARY NUMBER CONTAINS CHARACTER OTHER THAN '0' OR '1'.
383	CONVERSION	LDI: ATTEMPT TO CONVERT APPARENT REPRESENTATION OF CHARACTER (OR BIT) STRING FAILS DUE TO LACK OF CLOSING QUOTE.
391	CONVERSION	GENCONV: CHARACTER REPRESENTATION OF BIT STRING CONTAINS CHARACTER OTHER THAN '0' AND '1'.
394	ERROR	FORMAT: COMPLEX FORMAT ITEM ENCOUNTERED CONTAINING OTHER THAN E, F OR P FORMAT.
395	ERROR	FORMAT: CALL TO STU-$DECODE-VALUE FAILS. CONTACT PL/1 MAINTENANCE PERSONNEL.
396	ERROR	FORMAT: LABEL TENDERED AS FORMAT LABEL WAS NOT THE LABEL OF A FORMAT LIST; I.E., ITS FIRST WORD IS NOT ZERO.
397	ERROR	FORMAT: THIS IMPLEMENTATION DOES NOT PERMIT NESTING OF PARENTHESIS IN FORMAT LISTS TO A DEPTH GREATER THAN EIGHT (8). THIS INCLUDES NESTING PRODUCED BY EXPANSION OF REMOTE FORMAT ITEMS.
398	CONVERSION	GENCONV: CONVERSION FROM BIT STRING TO ARITHMETIC PRODUCES INTERMEDIATE FIXED BIN VALUE WITH PRECISION GREATER THAN 71.
399	ERROR	CTN: IMPROPER TARGET DESCRIPTOR. CHECK ARGUMENT 4 AND CONTACT PL/1 MAINTENANCE PERSONNEL.
400	CONVERSION	CTN: FIELD BEGINS WITH ILLEGAL CHARACTER. I.E., A CHARACTER OTHER THAN +-.0123456789.
401	CONVERSION	CTN: FIELD TERMINATES WHILE ANOTHER CHARACTER IS STILL REQUIRED. E.Q., DIGIT AFTER 'E' IN EXPONENTIAL FIELD.
402	CONVERSION	LDI: CLOSING QUOTE IN STRING REPRESENTATION IS FOLLOWED BY ILLEGAL CHARACTER. I.E., A CHARACTER OTHER THAN QUOTE('), 'B', OR SEPARATOR.
403	CONVERSION	CTN: ILLEGAL CHARACTER FOLLOWS A NUMERIC FIELD.
404	CONVERSION	CTN: TOO MANY DECIMAL POINTS IN A NUMERIC FIELD.
405	CONVERSION	CTN: CHARACTER OTHER THAN SIGN OR DIGIT FOLLOWS 'E' OF EXPONENT FIELD.

406	CONVERSION	CTN: CHARACTER OTHER THAN DIGIT FOLLOWS SIGN OF EXPONENT.
407	CONVERSION	CTN: TOO MANY EXPONENTIAL SUB-FIELD IN NUMERIC FIELD.
408	CONVERSION	CTN: CHARACTER OTHER THAN DIGIT FOLLOWS (SIGNED) DECIMAL POINT.
409	CONVERSION	CTN: CHARACTER OTHER THAN DECIMAL POINT OR DIGIT FOLLOWS SIGN.
410	CONVERSION	CTN: CHARACTER OTHER THAN SIGN, 'I' OR A SEPARATOR FOLLOWS 'B' OF BINARY NUMBER.
411	CONVERSION	CTN: CHARACTER OTHER THAN SEPARATOR FOLLOWS 'I' OF IMAGINARY (OR COMPLEX) NUMBER.
412	ERROR	LDI: SEMICOLON NOT ALLOWED EXCEPT FOR 'GET DATA'.
414	CONVERSION	CTN: EXPONENT VALUE > 999 NOT ALLOWED.
415	CONVERSION	CTX: ONLY BLANKS MAY BE USED FOR PADDING.
416	CONVERSION	CTN: EXPONENT OR SCALE > 127 DEVELOPED. A VALUE OF ZERO WILL BE USED.
417	CONVERSION	CTN: EXPONENT OR SCALE < -128 DEVELOPED. A VALUE OF ZERO WILL BE USED.
418	CONVERSION	CTN: PRECISION TOO HIGH - TOO MANY SIGNIFICANT DIGITS OR BITS IN FIELD.
419	CONVERSION	CTN: SECOND HALF OF COMPLEX NUMBER MUST BE IMAGINARY.
420	ERROR	PUT-UTIL: PUT STRING STATEMENT OVERFLOWS THE STRING.
421	ERROR	PUT-UTIL: FILE USED WITH PUT STATEMENT MUST HAVE 'STREAM' AND 'OUTPUT' ATTRIBUTES.
423	ERROR	PUT-UTIL: SKIP OPTION OR SKIP FORMAT ITEM NOT ALLOWED WITH STRING OPTION.
424	ERROR	PUT-UTIL; FOR NON-PRINT OUTPUT FILE, SKIP OPTION OR SKIP FORMAT ITEM WITH COUNT<1 IS ILLEGAL.
425	ENDPAGE	PUT-UTIL: OFF END OF PAGE DOING SKIP OPTION OR SKIP FORMAT ITEM.
426	ERROR	PUT-UTIL: LINE OPTION OR LINE FORMAT ITEM NOT ALLOWED WITH STRING OPTION.
427	ERROR	PUT-UTIL: LINE OPTION OR LINE FORMAT ITEM NOT ALLOWED WITH FILE LACKING PRINT ATTRIBUTES.
428	ENDPAGE	PUT-UTIL: OFF END OF PAGE DOING LINE OPTION OR LINE FORMAT ITEM.
429	ERROR	PUT-UTIL: PAGE OPTION OR PAGE FORMAT ITEM NOT ALLOWED WITH STRING OPTION.
430	ERROR	PUT-UTIL; PAGE OPTION OR PAGE FORMAT ITEM NOT ALLOWED WITH FILE LACKING PRINT ATTRIBUTES.
431	ENDPAGE	PUT-UTIL; OFF END OF PAGE WHILE PUTTING TEXT.
432	ERROR	XTX: BAD TYPE FIELD IN DESCRIPTOR OF INPUT OR OUTPUT ARGUMENT. CONTACT PL/1 MAINTENANCE PERSONNEL.
433	OVERFLOW	XTN: A RESULT IS DEVELOPED WITH EXPONENT OR SCALE > 127. A VALUE OF ZERO WILL BE RETURNED.
434	UNDERFLOW	XTN: A RESULT IS DEVELOPED WITH EXPONENT OR SCALE < -128. A VALUE OF ZERO WILL BE RETURNED.
435	SIZE	XTN: RESULT REQUIRES A GREATER PRECISION THAN THAT DECLARED FOR THE TARGET VARIABLE. A VALUE OF ZERO WILL BE STORED.

436	ERROR	XTN: INTERMEDIATE RESULT IS PRODUCED WITH ILLEGAL SCALE FACTOR (I.E., SCALE>127 OR SCALE<-128), A VALUE OF ZERO WILL BE RETURNED.
437	ERROR	PUT+COPY: NO FILE SPECIFIED FOR COPY OPTION. THE DEFUALT FILE, SYSPRINT, SHOULD HAVE BEEN SPECIFIED. CONTACT PL/1 MAINTENANCE PERSONNEL.
438	ERROR	PUT+DATA: THE NUMBER OF CHARACTERS IN A QUALIFIED NAME MUST NOT EXCEED 256.
439	ERROR	PUT+DATA: ERROR IN STU+$DECODE+VALUE. CONTACT PL/1 MAINTENANCE PERSONNEL.
440	ERROR	PUT+UTIL: ASCII OPTION OF THE $ DATA FILE NOT ALLOWED WITH FILE HAVING PRINT ATTRIBUTE.
442	ERROR	LDO: STRINGS LONGER THAN 256 NOT HANDLED IN THIS IMPLEMENTATION.
443	ERROR	LDO: THE CURRENT VALUE OF THE PICTURE VARIABLE DOES NOT MATCH WITH IT'S PICTURE SPECIFICATION.
459	ERROR	PUT+EDIT: INCORRECT NUMBER OF PARAMETERS IN FORMAT ITEM.
460	ERROR	PUT+EDIT: ILLEGAL FORMAT CODE ASSEMBLED. CONTACT PL/1 MAINTENANCE PERSONNEL.
461	ERROR	PUT+EDIT: SIZE OF FIELD ('W') NOT IN RANGE 0 TO 256.
462	ERROR	PUT+EDIT: BAD PARAMETER VALUES IN E OR F FORMAT ITEM.
463	ERROR	PUT+EDIT: FIELD WIDTH('W') TOO SMALL - RESULTING IN LOSS OF HIGH ORDER DIGITS OR SIGN.
464	STRINGSIZE	:PUT+EDIT: FIELD WIDTH('W') TOO SMALL.
470	OVERFLOW	NTN: EXPONENT OVERFLOW IN CONVERSION FROM BINARY TO BINARY FLOAT.
471	UNDERFLOW	NTN: EXPONENT UNDERFLOW IN CONVERSION FROM BINARY TO BINARY FLOAT.
472	SIZE	NTN: LOSS OF HIGH ORDER BIT(S) IN CONVERSION FROM BINARY TO BINARY FIXED (SCALED).
473	OVERFLOW	NTN: EXPONENT OVERFLOW IN CONVERSION FROM DECIMAL TO DECIMAL FLOAT.
474	UNDERFLOW	NTN: EXPONENT UNDERFLOW IN CONVERSION FROM DECIMAL TO DECIMAL FLOAT.
475	SIZE	NTN: LOSS OF HIGH ORDER DIGIT(S) IN CONVERSION FROM DECIMAL TO DECIMAL FIXED (SCALED).
480	UNDERFLOW	PICTURE: UNDERFLOW IN FLOATING PICTURE EDIT OR UNEDIT.
481	OVERFLOW	PICTURE: OVERFLOW IN FLOATING PICTURE EDIT OR UNEDIT.
482	ERROR	PICTURE: PICTURE CHARACTER STRING ERROR.
483	CONVERSION	PICTURE: CONVERSION ERROR IN CHARACTER TO PICTURE.
601	AREA	PL1+OPERATOR: AREA CONDITION RAISED BY BASED VARIABLE ALLOCATION.
602	STORAGE	PL1+OPERATOR: STORAGE CONDITION RAISED BY BASED VARIABLE ALLOCATION.
701	STRINGRANGE	PL1+OPERATOR: THE LENGTHS OF THE ARGUMENTS FIELD TO COMPLY WITH THE RULES DESCRIBED IN THE SUBSTR BUILTIN FUNCTION.
702	STRINGSIZE	PL1+OPERATOR: STRINGS ARE LOST IN AN ASSIGNMENT OR I/O OPERATION.
703	STRINGSIZE	PL1+OPERATOR: STRINGS ARE LOST IN AN ASSIGNMENT OR I/O OPERATION.
704	SUBSCRIPTRANGE	PL1+OPERATOR: A SUBSCRIPT LIES OUTSIDE ITS SPECIFIED BOUNDS.
705	ERROR	PL1+OPERATOR: THE NUMBER OF ARGUMENTS USED IN A SUBROUTINE REFERENCE OR A FUNCTION REFERENCE DOES NOT MATCH THE NUMBER OF PARAMETERS.
800	ZERODIVIDE	HARDWARE: ZERODIVIDE CONDITION RAISED BY PROGRAM FAULT.
801	FIXEDOVERFLOW	HARDWARE: FIXEDOVERFLOW CONDITION RAISED BY PROGRAM FAULT.
802	OVERFLOW	HARDWARE: OVERFLOW CONDITION RAISED BY PROGRAM FAULT.
803	UNDERFLOW	HARDWARE: UNDERFLOW CONDITION RAISED BY PROGRAM FAULT.

```
 804   ERROR            HARDWARE: ILLEGAL OP CODE FAULT RAISED BY PROGRAM FAULT.
 805   ERROR            HARDWARE: MEMORY ADDRESS FAULT RAISED BY PROGRAM FAULT.
 806   ERROR            HARDWARE: FAULT TAG FAULT RAISED BY PROGRAM FAULT.
 807   SIZE             HARDWARE: LOSS OF HIGH ORDER DIGIT(S).
1000   SIGNAL           SIGNAL: CONDITION RAISED BY SIGNAL STATEMENT.
```

INDEX

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE SERIES 60 (LEVEL 66)/6000
PL/I USER'S GUIDE

ORDER NO. DE04, REV. 0

DATED NOVEMBER 1975

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. □

FROM: NAME ____________________ DATE ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell

The Other Computer Company:

Honeywell

HONEYWELL INFORMATION SYSTEMS

14919
1.5276
Printed in U.S.A.

In the U.S.A.: 200 Smith Street, MS 061, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario