M T S

The Michigan Terminal System

Volume 7: PL/I in MTS

September 1982

| Updated September 1985 (Update 1)

The University of Michigan Computing Center
Ann Arbor, Michigan

```
****************************************************
*                                                  *
*       This obsoletes the July 1977 edition.      *
*                                                  *
****************************************************
```

DISCLAIMER

The MTS Manual is intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the Computing Center Newsletter, Computing Center Memos, and future Updates to this volume for the latest information about changes to MTS.

PREFACE

The software developed by the Computing Center staff for the operation of the high-speed processor computer can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Manual is a series of volumes that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Computing Center and the physical facilities provided are described in a separate publication entitled Introduction to the Computing Center.

The MTS volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are updated by means of CCMemos, users should check the Memo List, copy the files *CCMEMOS or *CCPUBLICATIONS, or watch for announcements in the Computing Center Newsletter, to ensure that their MTS volumes are fully up to date.

Volume 1:	The Michigan Terminal System, January 1984
Volume 2:	Public File Descriptions, April 1982
Volume 3:	System Subroutine Descriptions, April 1981
Volume 4:	Terminals and Networks in MTS, March 1984
Volume 5:	System Services, May 1983
Volume 6:	FORTRAN in MTS, October 1983
Volume 7:	PL/I in MTS, September 1982
Volume 8:	LISP and SLIP in MTS, June 1976
Volume 9:	SNOBOL4 in MTS, September 1975
Volume 10:	BASIC in MTS, December 1980
Volume 11:	Plot Description System, August 1978
Volume 12:	PIL/2 in MTS, December 1974
Volume 13:	The Symbolic Debugging System, September 1985
Volume 14:	360/370 Assemblers in MTS, May 1983
Volume 15:	FORMAT and TEXT360, April 1977
Volume 16:	ALGOL W in MTS, September 1980
Volume 17:	Integrated Graphics System, December 1980
Volume 18:	The MTS File Editor, August 1985
Volume 19:	Tapes and Floppy Disks, February 1983

Other volumes are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to

MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury

General Editor

September 1982

PREFACE TO REVISED VOLUME 7

The September 1982 revision reflects the changes that have been made to MTS since July 1977. Some of these changes were described in Updates 1 (February 1979) and 2 (October 1980) and which are incorporated in this revision. However, with the addition of MTS PL/I Optimizing compiler, it was felt that a complete revision of this volume was in order. The revision bars have been deleted and the pages have been renumbered to facilitate the future issuing of updates.

Acknowledgments for the descriptions contained in this volume are as follows:

The sections "PL/I Optimizing Compiler," "Run-Time Options," and "Program Checkout" are reprinted with permission from the IBM publication, OS PL/I Optimizing Compiler: Programmer's Guide, form SC33-0007.

The remainder of the descriptions in this volume were either produced or extensively modified from other documentation by the editorial and programming staffs of the University of Michigan Computing Center.

Contents

OVERVIEW OF PL/I

This volume describes the three PL/I language processors available in MTS:

(1) the PL/I (F) compiler
(2) the PL/I Optimizing compiler
(3) the PL/C compiler

The MTS version of the PL/I (F) compiler is derived from the IBM OS/360 F-level PL/I compiler (version 5). This compiler resides in the public file *PL1. The PL/I language supported by this compiler is described in the IBM publication, _IBM System/360 Operating System PL/I (F) Language Reference Manual_, form GC28-8201. Extensions and restrictions in MTS are given in the sections that follow.

The MTS version of the PL/I Optimizing compiler is derived from the IBM OS PL/I Optimizing compiler (release 1.3). This compiler resides in the public file *PL1OPT. The PL/I language supported by this compiler is described in the IBM publication, _OS PL/I Checkout and Optimizing Compilers: Language Reference Manual_, form GC33-0009.

The differences between the PL/I (F) compiler and the PL/I Optimizing compiler are given in the IBM publication, _OS PL/I Optimizing Compiler: General Information_, form GC33-0001.

The PL/C compiler is a compile-and-execute processor developed at Cornell University. This compiler resides in the public file *PLC. PL/C recognizes only a subset of the PL/I language as described in the above IBM publication. However, PL/C has extended the language in certain areas and also has superior compile-time and execution-time error-checking facilities.

Object modules produced by the PL/I (F) compiler may be debugged using SDS (the Symbolic Debugging System) and saved for later execution. PL/C, not being a true compiler, does not produce object modules; hence, PL/C-compiled programs cannot be debugged via SDS. Since each use of a PL/C program requires recompilation of the source program, programs compiled using PL/I (F) or Optimizing compilers are generally more suitable for production work.

In addition to the descriptions of the two language processors, this volume also contains descriptions of several auxiliary programs and PL/I subroutines available for PL/I users and a bibliography of other PL/I reference materials.

This volume is not intended as a replacement for other texts describing the PL/I language specifications. Except for the sections describing data representation and introductory input/output, only the differences between the OS and MTS implementations of the PL/I language are described.

COMPILING A PL/I (F) PROGRAM

The public file *PL1 contains the IBM System/360 PL/I (F) compiler. An alternate name for *PL1 is *PL/1. This compiler loads and executes under its supervision a series of phases that translate PL/I source statements into an object module, a set of machine instructions required to represent the source program. During compilation, *PL1 produces a listing which contains information about the source program and the object module, together with possible diagnostic messages. In addition, the compiler has a facility, the preprocessor or compile-time processor, which modifies source statements before the compilation.

The PL/I compiler is invoked as follows:

```
$RUN *PL1 [I/O unit assignments] [PAR=compiler options]
```

The information in brackets is optional and is explained below. For example,

```
$RUN *PL1 SCARDS=T SPUNCH=TT SPRINT=*PRINT* PAR=OPT=2,DIAG
```

MTS LOGICAL I/O UNITS

The compiler uses five logical I/O units:

SCARDS	- PL/I source language statements. This defaults to *SOURCE*.
SPRINT	- compiler output listings, including error messages and diagnostics. This defaults to *SINK*.
SERCOM	- compiler error messages and diagnostics when the DIAG option is in effect.
SPUNCH	- preprocessor-generated output source when the MACDCK option is specified or the object module when the DECK option (the default) is specified.
0	- the object module when the LOAD option is specified.

COMPILER OPTIONS

Compiler options can be specified in the PAR field of the MTS $RUN command. For example:

```
$RUN *PL1 SCARDS=SOU SPUNCH=OBJ SPRINT=-LIST PAR=LIST,DIAG
```

specifies the LIST and DIAG options. Options must be separated by commas and/or blanks and may be specified in any order. Most options may be abbreviated as indicated in the following table. These options are divided into six groups:

(1) control options used to set the conditions for compilation (e.g., size of text and dictionary blocks);

(2) preprocessor options used to request the preprocessor and to specify how its output is to be handled;

(3) input options used to specify the format of the input;

(4) object options used to specify the output of the object module and the manner in which it is handled;

(5) listing options used to specify the information to be included in the compiler listing;

(6) diagnostic options used to specify the diagnostics to be printed and the manner in which they are handled.

The table below lists all options with abbreviated forms and default values.

Compiler Options

Compiler Options		Abbreviated Name	Default
Control Options	SIZE=nP	SIZE=nP	SIZE=4P
	EXTDIC/NOEXTDIC	ED/NED	NOEXTDIC
Preprocessor Options	MACRO/NOMACRO	M/NM	NOMACRO
	SOURCE2/NOSOURCE2	S2/NS2	SOURCE2[1]
	MACDCK/NOMACDCK	MD/NMD	NOMACDCK
	COMP/NOCOMP	C/NC	COMP
Input Options	EBCDIC/BCD/TTY	EB/B/T	EBCDIC
	SORMGIN=(m,n,c)/FREE	SM=(m,n,c)/F	SM=(1,72)
	CHAR48/CHAR60	C48/C60	CHAR60
Object Options	DECK/NODECK	D/ND	DECK
	LOAD/NOLOAD	LD/NLD	NOLOAD
	STMT/NOSTMT	ST/NST	STMT
	OPT=n	O=n	OPT=1
	TEST/NOTEST	TEST/NOTEST	NOTEST
	MTS/OS	MTS/OS	MTS
Listing Options	LINECNT=nn	LC=nn	LINECNT=60
	OPLIST/NOOPLIST	OL/NOL	OPLIST[1]
	SOURCE2/NOSOURCE2	S2/NS2	SOURCE2[1]
	SOURCE/NOSOURCE	S/NS	SOURCE[1]
	NEST/NONEST	NT/NNT	NEST
	NUM/NONUM	NUM/NONUM	NUM
	ATR/NOATR	A/NA	ATR[1]
	XREF/NOXREF	X/NX	XREF[1]
	EXTREF/NOEXTREF	E/NE	NOEXTREF
	LIST/NOLIST	L/NL	NOLIST
	DUMP	DP	
Diagnostic Options	DIAG/NODIAG	DIAG/NODIAG	DIAG[2]
	FLAGW/FLAGE/FLAGS	FW/FE/FS	FLAGW
	SYNCHKE/SYNCHKS/SYNCHKT	SKE/SKS/SKT	SYNCHKS

[1]This is the default except when SPRINT output is assigned to the terminal by default.
[2]DIAG is the default if SPRINT and SERCOM do not refer to the same file or device or if SPRINT output is to be suppressed; otherwise, the default is NODIAG.

In the option descriptions which follow, the default values are underlined. The prefix NO, where applicable, reverses the effect of the option.

Control Options

SIZE=nP

The SIZE option is used to specify the page-size of the text and dictionary blocks allocated by the compiler. These blocks may be one, two, or four pages according to the SIZE option (SIZE=1P, SIZE=2P, SIZE=4P). The default is SIZE=4P. The text blocks should be allocated large enough so that each program statement will fit into the block; DECLARE statements may be subdivided into parts, each ending with a level-one comma or a semicolon. In specifying the SIZE option, consideration should also be given to the expansion of constants such as

```
DECLARE A PICTURE '(4000)X';
```

This statement fits into one complete dictionary block of SIZE=1P; however, if the statement is written as

```
DECLARE A PICTURE '(8000)X';
```

the statement will not fit into a block of this size. When this occurs, the following error message is generated:

```
IEM3844I  IMPLEMENTATION RESTRICTION.  DICTIONARY ENTRY
FOR STRING CONSTANT, PICTURE,  DOPE  VECTOR  OR  STATIC
INITIAL STRING IS TOO LONG FOR THIS SIZE OPTION.
```

In this case, the SIZE option must be increased.

EXTDIC or NOEXTDIC

The EXTDIC option is used to extend the dictionary to 3.5 times the normal dictionary capacity. If the following message is generated:

```
IEM3853I  IMPLEMENTATION  RESTRICTION.   SOURCE PROGRAM
TOO LARGE.  DICTIONARY IS FULL.
```

the program should be recompiled with the EXTDIC option. In most cases, the compilation will be successful. If, however, the following message is generated:

```
IEM3909I  EXTENDED  DICTIONARY  EXCEEDED.   COMPILATION
TERMINATED.
```

the program must be subdivided and recompiled. The EXTDIC option is recommended for large programs. The default is NOEXTDIC. Increasing the SIZE option will also alleviate this condition.

Preprocessor Options

MACRO or NOMACRO

The MACRO option must be specified if the source contains the compile-time statements as listed in the chapter "Compile-Time Facilities" of the IBM System/360 Operating System PL/I (F) Language Reference Manual, form number GC28-8201. The sole MTS restriction to the compile-time statements is that the %INCLUDE statement must be given in the following format:

```
%INCLUDE identifier [, ..., identifier ];
```

where "identifier" refers to an MTS file. The name of the file, however, must conform to the PL/I nomenclature of identifiers. That is, the first letter must be alphabetic, and only 31 alphanumeric characters are allowed. The default is NOMACRO.

In general, instead of using the %INCLUDE statement, it is preferable to use the MTS $CONTINUE WITH facility. This allows more freedom for file names and does not require the MACRO option, thus decreasing the cost of the compilation.

SOURCE2 or NOSOURCE2

The SOURCE2 option specifies that a listing of the input to the preprocessor is to be generated. NOSOURCE2 suppresses the listing. The default is SOURCE2 if SPRINT output is not assigned to the terminal by default.

MACDCK or NOMACDCK

The MACDCK option specifies that the preprocessor source output is to be written to the file attached to the logical I/O unit SPUNCH. The default is NOMACDCK. If this output deck is to be further processed, the source margins must be specified as SORMGIN=(2,72). The program *PL1TIDY may be invoked to remove irrelevant blanks and to properly indent statements. The standard MACDCK output format is:

Column 1	blank
Columns 2-72	the generated source field
Columns 73-77	the input line number from which the source statement was generated. This corresponds to the line number in the preprocessor input listing.

Columns 78-79	a two-digit number giving the maximum depth of replacement for this line. If no replacement occurred, these columns are blank.
Column 80	E signifies that an error occurred during the replacement. If no error occurred, this column is blank.

The logical I/O unit SPUNCH is also used for the DECK option. If the object module must be separated from the preprocessor source output, the option sequence "MACRO,LOAD,NODECK,MACDCK" should be specified. This causes the compiler to write the object module to logical I/O unit 0, and the preprocessor output to the unit SPUNCH.

COMP or NOCOMP

The COMP option specifies that the compiler should compile the PL/I source output produced by the preprocessor. If NOCOMP is specified, the PL/I source output is not compiled. The default is COMP.

Input Options

EBCDIC, BCD, or TTY

The compiler accepts source written in one of three codes: EBCDIC (Extended Binary-Coded Decimal Interchange Code), BCD (Binary-Coded Decimal), or TTY (Teletype). The default is EBCDIC. EBCDIC is the code used by the IBM 029 keypunch and most terminals supported by MTS. Whenever possible, EBCDIC should be used. BCD is used for source statements punched on an IBM 026 keypunch. TTY is the same as EBCDIC except for three characters that are not generally available on Teletypes. These are represented by other characters as shown below:

For "|" (vertical bar), enter "\" (reverse slant) with L-shift.

For "_" (underscore), enter either "←" (left-arrow) or "_" both with O-shift.

For "¬" (not-sign), enter either "↑" (up-arrow) or "ˆ" (caret) both with N-shift.

CHAR48 or CHAR60

This option specifies which character set is to be used for the source statements: the 48-character set (CHAR48) or the 60-character set (CHAR60). The default is CHAR60. The CHAR48 option should be regarded as a preprocessor option; this option must be specified if the source to the compiler was written in the

48-character set instead of the standard 60-character set. With the 48-character set, many PL/I symbols are represented by multiple characters. For example, "GE" represents ">=", "CAT" represents "||", ".." represents ":"; these special representations are listed in "Section B: Character Sets with EBCDIC and Card Punch Codes" in the IBM PL/I (F) Language Reference Manual. There are several restrictions in using the 48-character set; therefore, it is preferable to use the 60-character set (CHAR60 option), as this option completely skips the CHAR48 preprocessor, thus reducing significantly the cost of the compilation.

SORMGIN=(m,n,c) or FREE

The SORMGIN option specifies the beginning and ending column position of each input record. The compiler will not process any data outside these limits. Optionally, SORMGIN can specify the position to be used as the carriage-control character in the source listing. The general format of the SORMGIN option is

```
SORMGIN=(m,n[,c])
```

where "m" represents the beginning column position,
"n" represents the ending column position, and
"c" represents the carriage-control column position

There is a restriction that $1 \leq m \leq n \leq 100$. The compiler terminates the compilation if it encounters a record with more than 100 characters. Records of length less than the beginning column position "m" are flagged with an "*" in the source listing and are ignored. The default is SORMGIN=(1,72), which means the compiler will use only the first 72 characters of each record, although the complete record will be printed if the SOURCE option is in effect.

Optionally, "c" may be added to specify the position of the carriage-control character in each record. This must be outside the range (m,n). For example, SORMGIN=(2,72,1) indicates that the first position is taken as the carriage control and that positions 2 through 72 indicate the field of the source statements. Valid carriage-control characters are:

1	start a new page before printing
ƀ	skip one line before printing
0	skip two lines before printing
-	skip three lines before printing
+	suppress spacing before printing

Another option, FREE, is equivalent to SORMGIN=(1,100). FREE specifies free-formatted source; the source margins are set to the first and the last characters of the largest record that the compiler can handle.

Object Options

DECK or NODECK

The DECK option specifies that an object module is to be generated and written to the logical I/O unit SPUNCH. Note that the unit SPUNCH is also used by the preprocessor MACDCK option. The default is DECK.

LOAD or NOLOAD

The LOAD option specifies that an object module is to be generated and written to logical I/O unit 0. Note that it is possible for the compiler to produce two identical object modules if both DECK and LOAD options are specified. The default is NOLOAD.

MTS or OS

The MTS option specifies that an object module is to be generated for the MTS system. The OS option specifies that an object module is to be generated for the OS system. The default is MTS.

STMT or NOSTMT

The STMT option specifies that the object module include the necessary instructions for inserting a statement number into every executable statement. This statement number is printed in error messages at execution time. The STMT option slightly increases the execution time and the size of the object module. The default is STMT.

OPT=n

The OPT specifies the optimization level. There are three levels of optimization that can affect the object module:

OPT=0	The execution-time storage requirements are kept to a minimum at the expense of the object-program execution time.
OPT=1	The execution speed is improved at the expense of storage. OPT=1 is the default.
OPT=2	This is same as OPT=1, but in addition DO-loops and subscript expressions are optimized. This specification increases the compilation time but improves the execution time.

For the complete description of optimization, see the chapter "Optimization and Efficient Performance" in the IBM PL/I (F) Language Reference Manual.

TEST or <u>NOTEST</u>

The TEST option specifies that the compiler produce SYM records containing the symbolic data suitable for the Symbolic Debugging System (SDS). This option is handy for debugging PL/I programs. The default is NOTEST.

<u>Listing Options</u>

LINECNT=nn

The LINECNT option specifies the maximum number of lines to be printed for each page of the compiler listing. The default is LINECNT=60.

<u>OPLIST</u> or NOOPLIST

The OPLIST option specifies that a complete list of options in effect is to be printed at the start of the compiler listing. The default is OPLIST if SPRINT output is not assigned to the terminal by default.

<u>SOURCE</u> or NOSOURCE

The SOURCE option specifies that a listing of either the original source program or the output from the preprocessor is to be generated if SPRINT output is not assigned to the terminal by default. The default is SOURCE. Each source record is printed with the associated statement number and the MTS line number. If the NEST option is in effect, the PROCEDURE and BEGIN block levels and DO-group levels are printed. If the NUM option is in effect, the line numbers of the source are also printed. The table of storage requirements and the statistics are also printed. The following is a sample source listing for a typical source program.

Example:

```
STMT LEVEL NEST
  1              1.000 SAMPLE: PROCEDURE OPTIONS(MAIN);
  2     1        2.000         DECLARE ARRAY(15) FIXED BINARY,
                 3.000                 XYZ FLOAT;
  3     1        4.000           BEGIN;
  4     2        5.000 LABEL:      DO;
  5     2   1    6.000               DO I=1 TO 15;
  6     2   2    7.000               ARRAY(I)=I;
  7     2   2    8.000             END LABEL;
  9     2        9.000           END;
 10     1       10.000         PUT LIST(ARRAY) ;
 11     1       11.000         END;
```

Below is a sample listing of storage requirements and statistics.

Example:

```
          STORAGE REQUIREMENTS

          THE STORAGE AREA FOR THE PROCEDURE  LABELED  SAMPLE
          IS 224 BYTES LONG.

          THE STORAGE AREA (IN STATIC) FOR THE BEGIN BLOCK AT
          STATEMENT NO. 3 IS 180 BYTES LONG.

          THE  PROGRAM CSECT IS NAMED SAMPLE AND IS 454 BYTES
          LONG.

          THE STATIC CSECT IS NAMED *SAMPLEA AND IS 292 BYTES
          LONG.

   *STATISTICS* SOURCE RECORDS = 11, PROG TEXT STMNTS  =  11,
   OBJECT BYTES = 454
```

The storage areas are shown for every procedure, begin block, and ON-unit. Some storage areas are already initialized to save execution time; they are shown by the words "(IN STATIC)". The lengths of two control sections are shown in bytes. The first is the program control section, which contains all the machine instructions. The second is the static control section, which contains all internal static variables, static dynamic save areas, and constants.

The last two lines in the example above show the number of source records, the number of statements, and the size of the object modules in bytes. If the preprocessor was used, the number of input records to the preprocessor is also shown.

<u>ATR</u> or NOATR

The ATR option specifies that the compiler produce a table of all identifiers, in alphabetic order, with their attributes. The default is ATR if SPRINT output is not assigned to the terminal by default. Identifiers with precision FIXED BINARY (15,0) or less are flagged with "********", which indicates that the identifiers contain binary 16-bit halfwords (15 binary bits plus a sign bit). If the variable has been declared, the statement number of the DECLARE statement is shown under the heading "DCL NO." The attributes INTERNAL and REAL may be assumed unless the conflicting attributes EXTERNAL or COMPLEX appear in the attribute listing. If the variable is an array, its dimension is printed first but expressions within bounds are replaced by asterisks. Similarly, if the variable is a character or bit string, its length is shown just after the word STRING; however, if the length was determined by an expression, it is shown by an asterisk.

In addition, the compiler produces the Aggregate Length Table, giving the length in bytes for arrays and structures. If the length is not known at compilation time because an aggregate contains elements with adjustable lengths or dimensions, or because the aggregate is dynamically defined, the length is shown as either ADJUSTABLE or DEFINED.

<u>XREF</u> or NOXREF

The XREF option specifies that the compiler produce a table of all identifiers in alphabetical order, together with statement numbers of all statements in which they occur if SPRINT output is not assigned to the terminal by default. The default is XREF. If both ATR and XREF are specified (the default), the two tables are combined into one.

The sample source program shown above will produce a table of attributes and cross-references as follows.

Example:

```
                    ATTRIBUTES AND CROSS-REFERENCE TABLE

DCL NO.         IDENTIFIER   ATTRIBUTES AND REFERENCES

  2   ********  ARRAY      (15)AUTOMATIC,ALIGNED,BINARY,FIXED(15,0)
                           6,10

      ********  I          AUTOMATIC,ALIGNED,BINARY,FIXED(15,0)
                           5,6,6

  4             LABEL      STATEMENT LABEL CONSTANT

  1             SAMPLE     ENTRY,DECIMAL,FLOAT(SINGLE)

                SPRINT     FILE,EXTERNAL
                           10

  2             XYZ        AUTOMATIC,ALIGNED,DECIMAL,FLOAT(SINGLE)


                     AGGREGATE LENGTH TABLE

STATEMENT NO.   IDENTIFIER   LENGTH IN BYTES

  2             ARRAY            30
```

Note that labels referenced by END statements are not included in the cross-reference listing. Here, the compiler replaces the statement "END LABEL;" by two "END;" statements.

EXTREF or NOEXTREF

The option EXTREF specifies that the compiler list the External Symbol Dictionary (ESD). The default is NOEXTREF. The first ESD in the listing is the program control section, usually named as the first label of the external procedure statement. The second ESD entry is the static control section. The standard ESD entries follow. Finally, all entry labels, external static variables, controlled variables, pseudo-registers for each PROCEDURE or BEGIN statement, and all external routines are listed in the ESD listing.

Example:

```
     EXTERNAL SYMBOL DICTIONARY
 SYMBOL   TYPE  ID    ADDR   LENGTH

SAMPLE     SD  0001  000000  0001C6
*SAMPLEA   SD  0002  000000  000124
IHEQINV    PR  0003  000000  000004
IHESADA    ER  0004  000000
IHESADB    ER  0005  000000
IHEQERR    PR  0006  000000  000004
IHEQTIC    PR  0007  000000  000004
IHEMAIN    SD  0008  000000  000004
IHENTRY    SD  0009  000000  00000C
IHESAPC    ER  000A  000000
IHEQLWF    PR  000B  000000  000004
IHEQSLA    PR  000C  000000  000004
IHEQLW0    PR  000D  000000  000004
*SAMPLEB   PR  000E  000000  000004
*SAMPLEC   PR  000F  000000  000004
IHELDOB    ER  0010  000000
IHEIOBT    ER  0011  000000
IHEIOBA    ER  0012  000000
IHESAFA    ER  0013  000000
IHESPRT    SD  0014  000000  000038
IHEQSPR    PR  0015  000000  000004
```

The headings in an ESD table have the following meanings:

SYMBOL The eight-character name of the external symbol.

TYPE The two-character type of the external symbol:

SD - section definition
LD - label definition
PR - pseudo-register
ER - external reference
CM - common section

ID A four-digit hexadecimal number, numbered sequentially starting from 0001.

ADDR	The six-digit hexadecimal address of the symbol, always zero.
LENGTH	The six-digit hexadecimal length of the symbol.

LIST or NOLIST

The LIST option specifies that a complete list of the object module is to be generated. The default is NOLIST. The list contains a map of the static control sections and a list of the machine instructions as described in IBM System/370 Principles of Operation, form number GA22-7000. The list is printed in double-column format, unless the LINECNT option specifies more than 72 lines per page or the total lines remaining are less than the LINECNT. In these cases, single-column format is used.

For full details of the object module listing, the reader should consult the IBM System/360 Operating System PL/I (F) Programmer's Guide, form number GC28-6594.

If SOURCE, NOSTMT, and NOLIST are in effect, the compiler produces a table of offsets and statement numbers with procedures and ON-units. This should be useful, for example, if an execution error shows the offsets from a certain procedure but gives no statement number.

Example:

```
TABLE OF OFFSETS AND STATEMENT NUMBERS WITHIN PROCEDURE SAMPLE

OFFSET (HEX)   0000  0068  0068  0070  0082  009A  009A  00A0
STATEMENT NO.     1     4     5     6     7     8     9    10

OFFSET (HEX)   00E0
STATEMENT NO.    11
```

DUMP

The DUMP option, used only in case of a compiler failure, specifies that the compiler should dump on SPRINT a listing of the compiler modules, compiler storage, and all text and dictionary blocks.

Diagnostic Options

DIAG or NODIAG

The DIAG option specifies that the compiler print the diagnostic messages on SERCOM. The default is DIAG if SPRINT and SERCOM do not refer to the same file or device or if SPRINT output is to be suppressed. Otherwise, the the default is NODIAG.

Example:

```
#$RUN *PL1 SCARDS=T SPUNCH=Q SPRINT=*PRINT* PAR=DIAG
#EXECUTION BEGINS

 PROCEDURE T: SYNTAX CHECK COMPLETED. COMPILATION CONTINUES.

     COMPILER DIAGNOSTICS.

 WARNINGS.

     IEM0227I         NO FILE/STRING  OPTION  SPECIFIED
     IN  ONE OR MORE GET/PUT STATEMENTS.  SCARDS/SPRINT
     HAS BEEN ASSUMED IN EACH CASE.

 END OF DIAGNOSTICS.
#EXECUTION TERMINATED
```

The compiler generates diagnostic messages beginning with "IEMnnnI" where "nnn" represents the message number. The IBM PL/I (F) Programmer's Guide lists all messages in numeric order. If one or both of the CHAR48 or MACRO options are in effect, the compiler also produces preprocessor diagnostic messages immediately after the listing of the source to the preprocessor.

The messages are grouped according to their severity:

WARNING	(W)	A warning calling attention to a possible error.
ERROR	(E)	An error was found in a statement, but was corrected by the compiler.
SEVERE ERROR	(S)	A severe error was found which cannot be corrected by the compiler. A partial or whole statement is deleted.
TERMINATION ERROR	(T)	An error was found which forces termination of the compilation.

FLAGW, FLAGE, or FLAGS

The FLAG option controls printing of the diagnostics by setting the minimum severity.

FLAGW	prints all diagnostics.
FLAGE	prints all diagnostics except warnings.
FLAGS	prints only severe and termination errors.

The default is FLAGW.

SYNCHKE, SYNCHKS, or SYNCHKT

After the syntax checking, the compiler tests the severity of errors produced. It prints one of two messages:

```
PROCEDURE p: SYNTAX CHECK COMPLETED. COMPILATION CONTINUES.
```

or

```
PROCEDURE p: SYNTAX CHECK COMPLETED. COMPILATION TERMINATED.
```

where "p" is the first label of the external procedure.

If the compiler terminates compilation, no object module is produced. The conditions for terminating depend on the choice of options:

SYNCHKE -	terminates compilation if there are any errors of severity ERROR or above.
SYNCHKS -	terminates compilation if errors of severity SEVERE ERROR or above exist.
SYNCHKT -	terminates compilation only if there are TERMINATION errors.

The default is SYNCHKS.

MULTIPLE COMPILATION

More than one program may be processed during a single run of the compiler. This is achieved by placing, before the second and subsequent external procedures, a %PROCESS statement of the form:

```
%PROCESS('options');
```

where "options" indicates a list of compiler options, enclosed within primes. The percent sign must be in the first character of the record. There can be any number of blanks

(1) between the percent sign and the word PROCESS;
(2) between the word PROCESS and the left parenthesis "(";
(3) between a parenthesis and a prime;
(4) between the right parenthesis ")" and the semicolon ";".

If no options are to be specified, the statement may be given as

```
%PROCESS;
```

The option values will be carried over from the preceding option list, whether specified in a PAR field or in a previous %PROCESS statement.

Example:

```
$RUN *PL1 SPUNCH=OBJ1 0=OBJ2

     (external procedure 1)

%PROCESS('LOAD,NODECK');

     (external procedure 2)

%PROCESS;

     (external procedure 3)

$ENDFILE
```

In this example, there is only one $RUN command; thus, all I/O assignments necessary for the three external procedures must be included on it. The file OBJ1, assigned to the unit SPUNCH, has the object module for the first external procedure, and the file OBJ2, assigned to the unit 0, has the object module of the second and last external procedures.

RETURN CODES

At the end of single or multiple compilation, the compiler sets the return code, which is printed on the message beginning with "EXECUTION TERMINATED" provided that the MTS RCPRINT option is not set to OFF. The return codes and their meanings are indicated below:

Code	Meaning
0	No diagnostic messages issued; compilation completed without error; successful execution anticipated.
4	Warning messages only issued; compilation completed; successful execution probable.
8	Error messages issued; compilation completed, but with errors; execution may fail.
12	Severe error messages issued; compilation may have been completed, but with errors; successful execution improbable.
16	Termination error messages issued; compilation terminated abnormally; successful execution impossible.

PL/I OPTIMIZING COMPILER

The public file *PL1OPT contains the MTS version of IBM OS PL/I Optimizing Compiler (release 3.1 with PTF 070). This compiler loads and executes, under its supervision, a series of phases that translate PL/I source statements into an object module, a set of machine instructions required to represent the source program. During compilation, *PL1OPT produces a listing that contains information about the source program and the object module, together with possible diagnostic messages. In addition, the compiler has a facility, the _preprocessor_ or _compile-time processor_, that modifies source statements before the compilation.

The compiler provides a number of options, both at compile-time and at run-time. Options that can be specified at compile-time are known as _compiler options_. Options that can be specified at run-time are known as _run-time options_.

Compiler options, their abbreviated forms, and their defaults are shown in Figures 1 and 2; run-time options are shown in Figure 1 in the section "Run-Time Options."

Also provided is the ability to pass an argument to the PL/I main procedure. This facility is described in "Specifying Run-Time Options" in the section "Run-Time Options."

The PL/I Optimizing Compiler is invoked as follows:

```
$RUN *PL1OPT [I/O unit assignments] [PAR=compiler options]
```

The information in brackets is optional and is explained below. For example,

```
$RUN *PL1OPT SCARDS=T SPUNCH=TT SPRINT=*PRINT* PAR=OPT(TIME),SMSG
```

MTS LOGICAL I/O UNITS

The compiler uses the following logical I/O units:

SCARDS	- PL/I source language statements. This defaults to *SOURCE*.
SPRINT	- compiler output listings, including diagnostics. This defaults to *SINK*.

SERCOM	- output for TERMINAL option, especially diagnostics. This defaults to *MSINK*.
SPUNCH	- preprocessor-generated output source when the MDECK is specified, or the object module when the DECK (the default) is specified.
0	- the object module when the OBJECT option is specified.
1-19	- macro libraries for %INCLUDE statements.

In addition, the compiler also uses the temporary file -##SYSUT1 for a _spill file_ as a logical extension to main storage. If the SIZE option is too small, some parts of main storage will be spilled to the file. These parts consist of text and dictionary information.

SPECIFYING COMPILER OPTIONS

For each compilation, the default for a compiler option will apply unless it is overridden by specifying the option in a PROCESS statement or in the PAR field of a $RUN command.

An option specified in the PAR field overrides the default value, and an option specified in a PROCESS statement overrides both that specified in the PAR field and the default value.

Where conflicting attributes are specified, either explicitly or implicitly by the specification of other options, the latest implied or explicit option is accepted. No diagnostic message is issued to indicate that any options are overridden in this way.

Specifying Compiler Options in the $RUN Command

To specify options in the $RUN command, code PAR= followed by the list of options, in any order, separating the options with commas or blanks. For example:

```
     $RUN *PL1OPT PAR=OBJECT,LIST
```

Specifying Compiler Options in the /PROCESS statement

To specify options in the PROCESS statement, code as follows:

```
     / PROCESS options;
```

where "options" is a list of compiler options. The list of options must be terminated with a semicolon and should not extend beyond column 72. The slash must appear in column 1. (An asterisk "*" may be used instead of the slash.) The keyword PROCESS may follow in column 2 or after any number of blanks. Option keywords must be separated by a comma and/or at least one blank.

Blanks are permitted before and after any nonblank delimiter in the list, with the exception of strings within quotation marks, for example MARGINI('*'), in which padding blanks should not be inserted.

The number of characters is limited only by the length of the record. If no options are to be specified, code:

```
     / PROCESS ;
```

Should it be necessary to continue the PROCESS statement onto the next card or record, terminate the first part of the list after any delimiter, up to column 72, and continue on the next card or record. Option keywords or keyword arguments may be split, if required, when continuing onto the next record, provided that the keyword or argument string terminates in column 72, and the remainder of the string starts in column 1. A PROCESS statement may be continued in several statements, or a new PROCESS statement started. For the use of the PROCESS statement with multiple compilation, see "Multiple Compilation" later in this section.

COMPILER OPTIONS

The compiler options are of the following types:

(1) Simple pairs of keywords: a positive form, e.g., NEST, that requests a facility, and an alternative negative form, e.g., NONEST, that rejects that facility.

(2) Keywords that allow a value-list that qualifies the option, e.g., NOCOMPILE(E).

(3) A combination of (1) and (2) above.

The following paragraphs describe the options in alphabetic order. For those options that specify that the compiler is to list information, only a brief description is included.

Figure 1 lists all the compiler options with their abbreviated forms and their default values. Figure 2 lists the options by function so that the user can, for example, determine the options that are applicable to preprocessing.

Compiler Option	Abbreviation	Default
AGGREGATE|NOAGGREGATE	AG|NAG	NOAGGREGATE
ATTRIBUTES[(FULL|SHORT)]| NOATTRIBUTES	A[(F|S)]|NA	Print: ATTRIBUTES (FULL) Nonprint: NOATTRIBUTES
CHARSET([48|60] [EBCDIC|BCD])	CS([48|60] [EB|B])	CHARSET (60 EB)
COMPILE|NOCOMPILE[(W|E|S)]	C|NC[(W|E|S)]	NOCOMPILE(S)
COUNT|NOCOUNT	CT|NCT	NOCOUNT
DECK|NODECK	D|ND	DECK
DUMP|NODUMP	DU|NDU	NODUMP
ESD|NOESD	-	NOESD
FLAG[(I|W|E|S)]	F[(I|W|E|S)]	Print: FLAG(I) Nonprint: FLAG(W)
FLOW[(n,m)]	-	NOFLOW
GONUMBER|NOGONUMBER	GN|NGN	NOGONUMBER
GOSTMT|NOGOSTMT	GS|NGS	GOSTMT
INCLUDE|NOINCLUDE	INC|NINC	NOINCLUDE
INSOURCE|NOINSOURCE	IS|NIS	Print: INSOURCE Nonprint: NOINSOURCE
INTERRUPT|NOINTERRUPT	INT|NINT	NOINTERRUPT
LINECOUNT(n)	LC(n)	LC(60)
LIST[(m[,n])]|NOLIST	-	NOLIST
LMESSAGE|SMESSAGE	LMSG|SMSG	Print: LMESSAGE Nonprint: SMESSAGE
MACRO|NOMACRO	M|NM	NOMACRO
MAP|NOMAP	-	NOMAP
MARGINI(’c’)|NOMARGINI	MI(’c’)|NMI	MARGINI(’|’)
MARGINS(m,n[,c])	MAR(m,n[,c])	MAR(1,72)
MDECK|NOMDECK	MD|NMD	NOMDECK
NEST|NONEST	-	NEST
NUMBER|NONUMBER	NUM|NNUM	NONUMBER
OBJECT|NOOBJECT	OBJ|NOBJ	NOOBJECT
OFFSET|NOOFFSET	OF|NOF	NOOFFSET
OPTIMIZE(TIME|0|2)|NOOPTIMIZE	OPT(TIME|0|2)|NOPT	NOOPTIMIZE

Figure 1 (Part 1 of 2). Compiler options, abbreviations, and defaults.

Compiler Option	Abbreviation	Default
OPTIONS|NOOPTIONS	OP|NOP	Print: OPTIONS Nonprint: NOOPTIONS
SEQUENCE(m,n)|NOSEQUENCE	SEQ(m,n)|NSEQ	NOSEQUENCE
SIZE([-]yyyyy[K|P]|MAX)	SZ([-]yyyyy[K|P]|MAX)	SIZE(50P)
SOURCE|NOSOURCE	S|NS	Print: SOURCE Nonprint: NOSOURCE
STMT|NOSTMT	-	STMT
STORAGE|NOSTORAGE	STG|NSTG	NOSTORAGE
SYNTAX|NOSYNTAX[(W|E|S)]	SYN|NSYN[(W|E|S)]	NOSYNTAX(S)
TERMINAL[(optlist)]|NOTERMINAL	TERM[(optlist)] NTERM	Batch: NOTERMINAL Terminal: TERMINAL
XREF[(FULL|SHORT)]	X[(F|S)]|NX	Print: XREF(FULL) Nonprint: NOXREF

Figure 1 (Part 2 of 2). Compiler options, abbreviations, and defaults.

Compiler Options Listed by Function (Part 1)

LISTING OPTIONS	
Control listings produced	
AGGREGATE	list of aggregates and their sizes.
ATTRIBUTES[(FULL|SHORT)]	list of attributes of identifiers.
ESD	list of external symbol dictionary.
INSOURCE	list of preprocessor input.
FLAG(I|W|E|S)	suppress diagnostics messages below a certain severity.
LIST	list compiled code produced by compiler.
MAP	lists offsets of variables in static control sections and DSAs.
OFFSET	list of statement numbers with their associated offsets.
OPTIONS	list of options used.
SOURCE	list of source program or preprocessor output.
STORAGE	list of storage used.
XREF[(SHORT|FULL)]	list of statements in which each identifier is used.
Improve readability of source listing	
NEST	indicates do-group and block level by numbering in margin.
MARGINI	highlights any source outside margins.
Control lines per page of listing	
LINECOUNT	specifies number of lines per page on listing.
INPUT OPTIONS	
CHARSET	identify the character set used in source.
MARGINS	identify position of PL/I source and a carriage control character.
SEQUENCE	specify the columns used for sequence numbers.

Figure 2 (Part 1 of 3). Compiler options arranged by function.

Compiler Options Listed by Function (Part 2)	
OPTIONS TO PREVENT UNNECESSARY PROCESSING	
NOSYNTAX(W|E|S)	stop processing after errors are found in preprocessing.
NOCOMPILE(W|E|S)	stop processing after errors are found in syntax checking.
OPTIONS FOR PREPROCESSING	
INCLUDE	allows secondary input to be included without using preprocessor.
MACRO	allows preprocessor to be used.
MDECK	produces a source deck from preprocessor output.
OPTIONS TO IMPROVE PERFORMANCE	
OPTIMIZE/NOOPTIMIZE	OPTIMIZE improves execution performance but increases compilation time. NOOPTIMIZE does the reverse.
OPTIONS TO USE WHEN PRODUCING AN OBJECT MODULE	
OBJECT	produce an object module from compiled output.
DECK	produce an object module in punch card format.
OBJECT TO CONTROL STORAGE USED	
SIZE	controls the amount of storage used by the compiler.
OPTIONS TO IMPROVE USABILITY AT A TERMINAL	
TERMINAL	specifies how much of listing is transmitted to terminal.
SMESSAGE/LMESSAGE	specifies concise or full message format.
OPTIONS TO SPECIFY STATEMENT NUMBERING SYSTEM USED	
NUMBER & GONUMBER	numbers statements according to line on which they start.
STMT & GOSTMT	numbers statements sequentially.

Figure 2 (Part 2 of 3). Compiler options arranged by function.

Compiler Options Listed by Function (Part 3)	
OPTIONS FOR USE WHEN DEBUGGING	
COUNT	generate code that, if run-time COUNT is specified, will result in a count of the number of times each statement is executed.
FLOW	generate code that, if run-time FLOW is specified, will result in a trace of statements executed being retained.
OPTIONS TO CONTROL EFFECT OF ATTENTION INTERRUPTS	
INTERRUPT	specifies that the ATTENTION condition will be raised after an attention interrupt occurs.
OPTIONS FOR USE WHEN DEBUGGING COMPILER	
DUMP	produces a dump if the compiler terminates abnormally (ignored if used in /PROCESS statement).

Figure 2 (Part 3 of 3). Compiler options arranged by function.

AGGREGATE Option

The AGGREGATE option specifies that the compiler is to include in the compiler listing an aggregate length table, giving the lengths of all arrays and major structures in the source program.

Output example:

```
                      AGGREGATE LENGTH TABLE

DCL NO.   IDENTIFIER   LVL    DIMS       OFFSET  ELEMENT    TOTAL
                                                  LENGTH.    LENGTH.
2         ARRAY                  1                     2         30
                                   SUM OF CONSTANT LENGTHS       30
```

ATTRIBUTES[(FULL|SHORT)] Option

The ATTRIBUTES option specifies that the compiler is to include in the compiler listing a table of source-program identifiers and their attributes. If both ATTRIBUTES and XREF apply, the two tables are combined.

Unreferenced identifiers are marked by a series of asterisks.

If SHORT is specified, unreferenced identifiers are omitted, making the listing more manageable.

If both ATTRIBUTES and XREF apply, and there is a conflict between SHORT and FULL, the usage is determined by the last option found. For example, ATTRIBUTES(SHORT) XREF(FULL) results in FULL applying to the combined listing.

The suboption default FULL means that FULL applies, if the option is specified with no suboption.

Output example:

```
                    ATTRIBUTE AND CROSS-REFERENCE TABLE (FULL)

DCL NO.    IDENTIFIER    ATTRIBUTES AND REFERENCES

2          ARRAY         (15) AUTOMATIC ALIGNED BINARY FIXED (15,0)
                         9
                         6
********   I             AUTOMATIC ALIGNED BINARY FIXED (15,0)
                         5,5,6,6
4          LABEL         /* STATEMENT LABEL CONSTANT */
1          SAMPLE        EXTERNAL ENTRY RETURNS(DECIMAL
                         /* SINGLE */ FLOAT (6))
********   SYSPRINT      EXTERNAL FILE PRINT
                         9
2          XYZ           AUTOMATIC ALIGNED DECIMAL
                         /* SINGLE */ FLOAT (6)
```

CHARSET Option

The CHARSET option specifies the character set and data code that is used to create the source program. The compiler will accept source programs written in the 60-character set or the 48-character set, and in the Extended Binary Coded Decimal Interchange Code (EBCDIC) or Binary Coded Decimal (BCD).

60- or 48-character Set: If the source program is written in the 60-character set, specify CHARSET(60); if it is written in the 48-character set, specify CHARSET(48). The language reference manual for this compiler lists both of these character sets. (The compiler will accept source programs written in either character set if CHARSET(48) is specified. However, if the reserved key-words, for example CAT or LE, are used as identifiers, errors may occur.)

BCD or EBCDIC: If the source program is written in BCD, specify CHARSET(BCD); if it is written in EBCDIC, specify CHARSET(EBCDIC). The language reference manual for this compiler lists the EBCDIC representation of both the 48-character set and the 60-character set.

If both arguments (48 or 60, EBCDIC or BCD) are specified, they may be in any order and should be separated by a blank or a comma.

COMPILE Option

The COMPILE option specifies that the compiler is to compile the source program unless an unrecoverable error was detected during preprocessing or syntax checking. The NOCOMPILE option without an argument causes processing to stop unconditionally after syntax checking. With an argument, continuation depends on the severity of errors detected so far, as follows:

NOCOMPILE(W)	No compilation if a warning, error, severe error, or unrecoverable error is detected.
NOCOMPILE(E)	No compilation if error, severe error, or unrecoverable error is detected.
NOCOMPILE(S)	No compilation if a severe error or unrecoverable error is detected.

If the compilation is terminated by the NOCOMPILE option, the cross-reference listing and attribute listing may be produced; the other listings that follow the source program will not be produced.

COUNT Option

The COUNT option specifies (1) that the compiler is to produce code that, when the run-time COUNT (or FLOW) option is specified, counts and the lists the number of times each statement is executed, and (2) that the default run-time option for COUNT/NOCOUNT be set to COUNT.

The COUNT option implies the GOSTMT option if the STMT option applies, or the GONUMBER option if the NUMBER option applies.

DECK Option

The DECK option specifies that the compiler is to produce an object module in the form of 80-column card images and write it to the MTS logical I/O unit SPUNCH. Columns 73-76 of each card contain a code to identify the object module; this code comprises the first four characters of the first label in the external procedure represented by the object module. Columns 77-80 contain a 4-digit decimal number: the first card is numbered 0001, the second 0002, and so on.

DUMP Option

The DUMP option specifies that the compiler is to produce a formatted dump of main storage, if the compilation terminates abnormally (usually due to a compiler error). This dump is written on the MTS logical I/O unit SPRINT.

ESD Option

The ESD option specifies that the external symbol dictionary (ESD) is to be listed in the compiler listing. The default is NOESD.

Output example:

```
                    EXTERNAL SYMBOL DICTIONARY

              SYMBOL         TYPE     ID      ADDR      LENGTH

             PLISTART         SD     0001    000000    000050
             *SAMPLE1         SD     0002    000000    000150
             *SAMPLE2         SD     0003    000000    000108
             PLITABS          WX     0004    000000
             PLIXOPT          WX     0005    000000
             IBMBPOPT         WX     0006    000000
             PLIXHD           WX     0007    000000
             IBMBEATA         WX     0008    000000
             PLIFLOW          WX     0009    000000
             PLICOUNT         WX     000A    000000
             IBMBPIRA         ER     000B    000000
             IBMBPIRB         ER     000C    000000
             IBMBPIRC         ER     000D    000000
             PLICALLA         LD             000006
             PLICALLB         LD             00000A
             PLIMAIN          SD     000E    000000    000008
             IBMBSLOA         ER     000F    000000
             IBMBCACA         ER     0010    000000
             IBMBCHFD         ER     0011    000000
             IBMBCWDH         ER     0012    000000
             IBMBOCLA         ER     0013    000000
             IBMBOCLC         WX     0014    000000
             IBMBSIOA         ER     0015    000000
             IBMBSIOT         WX     0016    000000
             IBMBSLOB         WX     0017    000000
             IBMBSXCA         WX     0018    000000
             IBMBSXCB         WX     0019    000000
             IBMBSIST         WX     001A    000000
             SAMPLE           LD             000008
             SYSPINT          SD     001B    000000    000020
```

The headings in an ESD table have the following meanings:

SYMBOL   The eight-character name of the external symbol.

TYPE   The two-character type of the external symbol:

SD - section definition
LD - label definition
PR - pseudo-register
ER - external reference

WX - weak external reference
CM - common section

ID	A four-digit hexadecimal number, numbered sequentially starting from 0001.
ADDR	The six-digit hexadecimal address of the symbol.
LENGTH	The six-digit hexadecimal length of the symbol.

FLAG Option

The FLAG option specifies the minimum severity of error that requires a message to be listed in the compiler listing. The format of the FLAG option is shown below.

FLAG(I)	List all messages.
FLAG(W)	List all except informatory messages. If FLAG is specified, FLAG(W) is assumed.
FLAG(E)	List all except warning and informatory messages.
FLAG(S)	List only severe error and unrecoverable error messages.

FLOW Option

The FLOW option specifies (1) that the compiler is to produce code that, when the run-time FLOW option is specified, lists the flow of control when the program is executed, and (2) that the default run-time option for FLOW|NOFLOW be set to FLOW. The format of the FLOW option is:

```
FLOW[(n,m)]
```

where

n	is the maximum number of entries to be included in the lists. It should not exceed 32767.
m	is the maximum number of procedures for which the lists are to be generated. It should not exceed 32767.

The default, if (n,m) is not specified, is (25,10).

The output produced by the FLOW option is described under "Run-Time FLOW Option" in the section "Run-Time Options."

GONUMBER Option

The GONUMBER option specifies that the compiler is to produce additional information that will allow line numbers from the source programs to be included in run-time messages. Alternatively, these line numbers can be derived by using the offset address, which is always included in run-time messages, and the table produced by the OFFSET option. (The NUMBER option must also apply.)

Use of the GONUMBER option implies NUMBER, NOSTMT, and NOGOSTMT. If NUMBER applies, GONUMBER is forced by the COUNT option.

GOSTMT Option

The GOSTMT option specifies that the compiler is to produce additional information that will allow statement numbers to be included in run-time messages. Alternatively, these statement numbers can be derived by using the offset addresses, which are always included in run-time messages, and the table produced by the OFFSET option. (The STMT option must also apply.)

Use of the GOSTMT option implies STMT, NONUMBER, and NOGONUMBER. If STMT applies, GOSTMT is forced by the COUNT option.

INCLUDE Option

The INCLUDE option requests the compiler to handle the inclusion of PL/I source statements for programs that use the %INCLUDE statement. For programs that use the %INCLUDE statement but no other PL/I preprocessor statements, this method is faster than using the preprocessor. If the MACRO option is also specified, the INCLUDE option has no effect.

INSOURCE Option

The INSOURCE option specifies that the compiler is to include a listing of the source program (including preprocessor statements) in the compiler listing. This option is applicable only when the preprocessor is used, therefore the MACRO option must also apply.

INTERRUPT Option

If INTERRUPT was in effect during compilation, an established ON ATTENTION on-unit will be executed when an attention interrupt occurs. If the on-unit is not established, a message is printed. The compiler inserts code to check the occurrence of an attention interrupt at execution of stream I/O, at branching points. Due to overhead, it is recommended that the compiler option INTERRUPT not be specified and the run-time option ATTN be specified instead.

If NOINTERRUPT was in effect during compilation, then attention interrupts will be handled by the PL/I error handler provided that the run-time option ATTN is specified.

LINECOUNT Option

The LINECOUNT option specifies the number of lines, including heading lines and blank lines, to be included in each page of the compiler listing. The format of the LINECOUNT option is:

```
LINECOUNT(n)
```

where

n	is the number of lines. It must be in the range 1 through 32767, but only headings are generated if less than 7 is specified.

LIST Option

The LIST option specifies that the compiler is to include a listing of the object module (in a form similar to IBM System/360 assembler language instructions) in the compiler listing. The format of the LIST option is:

```
LIST[(m[,n])]
```

where "m" is the number of the first, or only, source statement for which an object listing is required and "n" is the number of the last source statement for which an object listing is required. If "n" is omitted, only statement m is listed. If the option NUMBER applies, m and n must be specified as line numbers.

If LIST is used in conjunction with MAP, additional listings of static storage are produced (see the MAP option).

LMESSAGE Option

The LMESSAGE and SMESSAGE options specify that the compiler is to produce messages in a long form (specify LMESSAGE) or in a short form (specify SMESSAGE). Short messages can have advantages due to the comparatively slow printing speed of a terminal.

<u>LMESSAGE</u> output example:

```
COMPILER DIAGNOSTIC MESSAGES

ERROR ID L   STMT    MESSAGE DESCRIPTION

WARNING DIAGNOSTIC MESSAGES

IEL0916I W   1       ITEM(S) 'I' MAY BE UNINITIALIZED WHEN USED IN
                     THIS BLOCK.
IEL0385I W   7       MULTIPLE CLOSURE OF BLOCK.    1 EXTRA 'END'
                     STATEMENT(S) ASSUMED.

COMPILER INFORMATORY MESSAGES

IEL0533I I           NO 'DECLARE' STATEMENT(S) FOR 'SYSPRINT','I'.
IEL0541I I   1, 3    'ORDER' OPTION APPLIES TO THIS BLOCK.
                     OPTIMIZATION MAY BE INHIBITED.

END OF COMPILER DIAGNOSTIC MESSAGES

COMPILE TIME    0.00 MINS    SPILL FILE:     0 RECORDS, SIZE  4051
```

<u>SMESSAGE</u> output example:

```
COMPILER DIAGNOSTIC MESSAGES

ERROR ID L   STMT    MESSAGE DESCRIPTION

WARNING DIAGNOSTIC MESSAGES

IEL0916I W   1       ITEM(S) 'I' MAY BE UNINITIALIZED.
IEL0385I W   7       1 EXTRA 'END' STATEMENT(S) ASSUMED.

COMPILER INFORMATORY MESSAGES

IEL0533I I           NO 'DECLARE' STATEMENT(S) FOR 'SYSPRINT','I'.
IEL0541I I   1, 3    'ORDER' MAY INHIBIT OPTIMIZATION.

END OF COMPILER DIAGNOSTIC MESSAGES

COMPILE TIME    0.00 MINS    SPILL FILE:     0 RECORDS, SIZE  4051
```

<u>MACRO Option</u>

The MACRO option specifies that the source program is to be processed by the preprocessor.

<u>MAP Option</u>

The MAP option specifies that the compiler is to produce tables showing the organization of the static storage for the object module. A table showing the mapping of static and automatic variables with offsets from their defining bases is always produced. If the LIST option is also used, a map of the static internal and external control sections is also generated.

Output example:

```
              VARIABLE STORAGE MAP

IDENTIFIER      LEVEL    OFFSET     (HEX)    CLASS      BLOCK

ARRAY               1       200        C8    AUTO       SAMPLE
XYZ                 1       192        C0    AUTO       SAMPLE
I                   1       196        C4    AUTO       SAMPLE
```

MARGINI Option

The MARGINI option specifies that the compiler is to include a specified character in the column preceding the left-hand margin, and in the column following the right-hand margin of the listings resulting from the INSOURCE and SOURCE options. Any text in the source input which precedes the left-hand margin will be shifted left one column, and any text that follows the right-hand margin will be shifted right one column. For input records that do not extend as far as the right-hand margin, the character is inserted in the column following the end of the record. Thus, text outside the source margins can be easily detected.

The MARGINI option has the format:

```
MARGINI('c')
```

where "c" is the character to be printed as the margin indicator.

MARGINS Option

The MARGINS option specifies the part of each input record that contains PL/I statements. The compiler will not process data that is outside these limits (but it will include it in the source listings).

The option can also specify the position of a printer control character to format the listing produced if the SOURCE option applies. This is an alternative to using %PAGE and %SKIP statements (described in the language reference manual for this compiler). If neither method is used, the input records will be listed without any intervening blank lines. The format of the MARGINS option is:

```
MARGINS(m,n[,c])
```

where

m is the column number of the leftmost character that will be processed by the compiler. It should not exceed 100.

n is the column number of the rightmost character that will be processed by the compiler. It should be greater than m, but not greater than 100.

c is the column number of the printer control character. It should not exceed 100 and should be outside the values specified for m and n. Only the following control characters can be used:

(blank)	Skip one line before printing.
0	Skip two lines before printing.
-	Skip three lines before printing.
+	No skip before printing.
1	Start new page.

The default is MARGINS(1,72). This specifies that there is <u>no</u> printer control character.

MDECK Option

The MDECK option specifies that the preprocessor is to produce a copy of its output (see the MACRO option) and write it to the MTS logical I/O unit SPUNCH. The option MARGINS(2,72) should be specified, if the output deck is to be compiled.

NEST Option

The NEST option specifies that the listing resulting from the SOURCE option will indicate, for each statement, the block level and the do-group level.

NUMBER Option

The NUMBER option specifies that the numbers specified in the sequence fields in the source input records are to be used to derive the statement numbers in the listings resulting from the AGGREGATE, ATTRIBUTES, LIST, OFFSET, SOURCE, and XREF options.

If NONUMBER is specified, STMT and NOGONUMBER are implied. NUMBER is implied by NOSTMT or GONUMBER.

The position of the sequence field can be specified in the SEQUENCE option, usually the last 8 columns for source input records.

It is necessary to specify the SEQUENCE option, or change the MARGINS defaults. Note that the preprocessor has fixed-length records irrespective of the original input. Any sequence numbers in the input are repositioned in columns 73-80.

The line number is calculated from the five right-hand characters of the sequence number (or the number specified, if less than five). These characters are converted to decimal digits if necessary. Each time a sequence number is found that is not greater than the preceding line number, a new line number is formed by adding the minimum integral multiple of 100,000 necessary to produce a line number that is greater than the preceding one. If the sequence field consists only of blanks, a new line number is formed by adding 10 to the preceding one. The maximum line number permitted by the compiler is 134,000,000, or, when FLOW/COUNT is specified, the maximum becomes 33,000,000; numbers that would normally exceed this are set to this maximum value. Only eight digits are printed in the source listing; line numbers of 100,000, 000 or over will be printed without the leading "1" digit.

OBJECT Option

The OBJECT option specifies that the compiler is to write the object module that it creates to the MTS logical I/O unit 0.

OFFSET Option

The OFFSET option specifies that the compiler is to print a table of statement or line numbers for each procedure with their offset addresses relative to the primary entry point of the procedure. This information is of use in identifying the statement being executed when an error occurs and a listing of the object module (obtained by using the LIST option) is available. If GOSTMT applies, statement numbers, as well as offset addresses, will be included in run-time messages. If GONUMBER applies, line numbers, as well as offset addresses, will be included in run-time messages.

Output example:

```
              TABLES OF OFFSETS AND STATEMENT NUMBERS

              WITHIN PROCEDURE SAMPLE

OFFSET (HEX)       0      66      70      B4
STATEMENT NO.      1       3       9      10

              WITHIN BEGIN BLOCK

OFFSET (HEX)       0      4E      56      62      66      6A
STATEMENT NO.      3       4       5       6       5       6

OFFSET (HEX)      6E      6E      70      78      78
STATEMENT NO.      7       6       7       5       8
```

OPTIMIZE Option

The OPTIMIZE option specifies the type of optimization required:

NOOPTIMIZE	specifies fast compilation speed, but inhibits optimization for faster execution and reduced main storage requirements.
OPTIMIZE(TIME)	specifies that the compiler is to optimize the machine instructions generated to produce a very efficient object program. A secondary effect of this type of optimization can be a reduction in the amount of main storage required for the object module. The use of OPTIMIZE(TIME) could result in a substantial increase in compile time over NOOPTIMIZE.
OPTIMIZE(0)	is the equivalent of NOOPTIMIZE.
OPTIMIZE(2)	is the equivalent of OPTIMIZE(TIME).

The language reference manual for this compiler includes a full discussion of optimization.

OPTIONS Option

The OPTIONS option specifies that the compiler is to include in the compiler listing, a list showing the compiler options to be used during this compilation. This list includes all those applied by default, those specified in the PAR field of a $RUN command, and those specified in a PROCESS statement.

Output example:

```
OPTIONS SPECIFIED

AG,ESD,NMI,OPT(TIME),STORAGE,OFFSET,LIST;

OPTIONS USED

AGGREGATE      NOCOUNT        ATTRIBUTES(FULL)
DECK           NOFLOW         CHARSET(60,EBCDIC)
ESD            NOGONUMBER     NOCOMPILE(S)
GOSTMT         NOIMPRECISE    FLAG(I)
INSOURCE       NOINCLUDE      LINECOUNT(60)
LIST           NOINTERRUPT    MARGINS(1,72,0)
LMESSAGE       NOMACRO        OPTIMIZE(TIME)
NEST           NOMAP          SIZE(50P)
OFFSET         NOMARGINI      NOSYNTAX(S)
OPTIONS        NOMDECK        XREF(FULL)
SOURCE         NONUMBER       TERMINAL(NOAGGREGATE,
STMT           NOOBJECT                NOATTRIBUTES,
STORAGE        NOSEQUENCE              NOESD,
                                       NOINSOURCE,
                                       NOLIST,
                                       NOMAP,
                                       NOOFFSET,
                                       NOOPTIONS,
                                       NOSOURCE,
                                       NOSTORAGE,
                                     NOXREF)

SCARDS    = sample
SPRINT    = sample.p
SPUNCH    = sample.o
SERCOM    = *MSINK*
```

SEQUENCE Option

The SEQUENCE option specifies the extent of the part of each input line or record that contains a sequence number. This number is included in the source listings produced by the INSOURCE and SOURCE option. Also, if the NUMBER option applies, line numbers will be derived from these sequence numbers and will be included in the source listings in place of statement numbers. No attempt is made to sort the input lines or records into the specified sequence. The SEQUENCE option has the format:

```
SEQUENCE(m,n)
```

where

m   specifies the column number of the left-hand margin.

n specifies the column number of the right-hand margin.

The extent specified should not overlap with the source program (as specified in the MARGINS option).

The default, NOSEQUENCE, indicates the absence of the sequence numbers.

SIZE Option

This option can be used to limit the amount of main storage used by the compiler. This is of value, for example, when dynamically invoking the compiler, to ensure that space is left for other purposes. The SIZE option can be expressed in seven forms:

SIZE(yyyyyyy)	specifies that "yyyyyyy" bytes of main storage are to be requested. Leading zeros are not required.
SIZE(yyyyK)	specifies that "yyyyK" bytes of main storage are to be requested (1K=1024). Leading zeros are not required.
SIZE(yyyP)	specifies that "yyyP" bytes of main storage are to be requested (1P=4096). Leading zeros are not required.
SIZE(-yyyyyyy)	specifies that the compiler is to obtain as much main storage as it can, and then release "yyyyyyy" bytes to the operating system. Leading zeros are not required.
SIZE(-yyyyK)	specifies that the compiler is to obtain as much main storage as it can, and then release "yyyyK" bytes to the operating system (1K=1024). Leading zeros are not required.
SIZE(-yyyP)	specifies that the compiler is to obtain as much main storage as it can, and then release "yyyP" bytes to the operating system (1P=4096). Leading zeros are not required.
SIZE(MAX)	specifies that the compiler is to obtain as much main storage as it can. This is currently 256 pages (or 1,048,576 bytes).

The default is SIZE(50P), which permits the compiler to allocate 50 pages.

The value, once determined, cannot be changed after processing has begun. This means, that in a multiple compilation, the value established when the compiler is invoked cannot be changed for later external procedures. Thus, it is ignored if specified in a PROCESS statement.

SMESSAGE Option

See LMESSAGE option.

SOURCE Option

The SOURCE option specifies that the compiler is to include in the compiler listing a listing of the source program. The source program is either the original source input or, if the MACRO option applies, the output from the preprocessor.

Output example:

```
                              SOURCE LISTING
STMT LEV NT   MTS LINE#

   1       0       1.    SAMPLE: PROCEDURE OPTIONS(MAIN);
   2   1   0       2.            DECLARE ARRAY(15) FIXED BINARY,
                   3.                    XYZ FLOAT;
   3   1   0       4.              BEGIN;
   4   2   0       5.    LABEL:      DO;
   5   2   1       6.                  DO I=1 TO 15;
   6   2   2       7.                  ARRAY(I)=I;
   7   2   2       8.                END LABEL;
   8   2   0       9.              END;
   9   1   0      10.            PUT LIST(ARRAY);
  10   1   0      11.            END;
```

STMT Option

The STMT option specifies that statements in the source program are to be counted, and that this "statement number" is used to identify statements in the compiler listings resulting from the AGGREGATE, ATTRIBUTES, LIST, OFFSET, SOURCE, and XREF options. STMT is implied by NONUMBER or GOSTMT. If NOSTMT is specified, NUMBER and NOGOSTMT are implied.

STORAGE Option

The STORAGE option specifies that the compiler is to include in the compiler listing a table giving the main storage requirements for the object module.

Output example:

```
                    STORAGE REQUIREMENTS

BLOCK, SECTION  TYPE             LENGTH   (HEX)    DSA SIZE   (HEX)
  OR STATEMENT

*SAMPLE1        PROGRAM CSECT       336     150
*SAMPLE2        STATIC CSECT        264     108
SAMPLE          PROCEDURE BLOCK     204      CC         304     130
BLOCK.02        BEGIN BLOCK         130      82         224      E0
```

SYNTAX Option

The SYNTAX option specifies that the compiler is to continue into syntax checking after initialization (or after preprocessing, if the MACRO option applies) unless an unrecoverable error is detected. The NOSYNTAX option without an argument causes processing to stop unconditionally after initialization (or preprocessing). With an argument, continuation depends on the severity of errors detected so far, as follows:

NOSYNTAX(W)	No syntax checking if a warning, error, severe error, or unrecoverable error is detected.
NOSYNTAX(E)	No syntax checking if an error, severe error, or unrecoverable error is detected.
NOSYNTAX(S)	No syntax checking if a severe error or unrecoverable error is detected.

If the SOURCE option applies, the compiler will generate a compiler listing even if syntax checking is not performed.

If the compilation is terminated by the NOSYNTAX option, the cross-reference listing, attribute listing, and other listings that follow the source program will not be produced.

The use of this option can prevent wasted runs when debugging a PL/I program that uses the preprocessor.

TERMINAL Option

The TERMINAL option specifies that a subset of or all of the compiler listing produced during compilation is to be printed on the MTS logical I/O unit SERCOM (usually at the terminal). If TERMINAL is specified without an argument, diagnostic and informatory messages are printed on SERCOM. An argument can be added, which takes the form of an option list, to specify other parts of the compiler listing that are to be printed on SERCOM.

The listing on SERCOM is independent of that written on SPRINT. However, if SPRINT is associated with SERCOM, only one copy of each option requested will be printed even if it is requested in the TERMINAL option and also as an independent option. The following option keywords, their negative forms, or their abbreviated forms, can be specified in the option list:

AGGREGATE, ATTRIBUTES, ESD, INSOURCE, LIST,
MAP, OPTIONS, SOURCE, STORAGE, and XREF.

If the option does not apply to the SPRINT listing, specifying it in the TERMINAL option has no effect. The other options that relate to the listing (that is, FLAG, GONUMBER, LINECOUNT, LMESSAGE/SMESSAGE, MARGINI, NEST, NUMBER, and the SHORT and FULL suboptions of ATTRIBUTES and XREF) will be the same as for the SPRINT listing.

XREF[(SHORT|FULL)] Option

The XREF option specifies that the compiler is to include in the compiler listing a cross-reference table of names used in the program together with the numbers of the statements in which they are declared or referenced.

If the suboption SHORT is specified, unreferenced identifiers are not listed, making the listing more manageable.

The default suboption FULL means that FULL applies if the option is specified with no suboption.

If both XREF and ATTRIBUTES are specified, the two listings are combined. If there is a conflict between SHORT and FULL, the usage is determined by the last option specified. For example, ATTRIBUTES(SHORT) XREF(FULL) results in FULL applying to the combined listing.

For an example of XREF table, see the ATTRIBUTES option.

MESSAGES

Messages are generated automatically if the preprocessor or the compiler detects an error, or the possibility of an error. Messages generated by the preprocessor appear in the listing immediately after the listing of the statements processed by the preprocessor. The user may generate messages in the preprocessing stage by use of the %NOTE statement. Such messages might be used to show how many times a particular replacement had been made. Messages generated by the compiler appear at the end of the listing. All messages are graded according to their severity, as follows:

- An informatory (I) message calls attention to a possible inefficiency in the program or gives other information generated by the compiler that may be of interest to the programmer.

- A warning (W) message calls attention to a possible error, although the statement to which it refers is syntactically valid.

- An error (E) message describes an error detected by the compiler for which the compiler has applied a "fix-up" with confidence. The resulting program will execute and will probably give correct results.

- A severe error (S) message specifies an error detected by the compiler for which the compiler cannot apply a "fix-up" with confidence. The resulting program will execute but will not give correct results.

- An unrecoverable error (U) message describes an error that forces termination of the compilation.

The compiler lists only those messages with a severity equal to or greater than that specified by the FLAG option, as shown in Figure 3.

Each message is identified by an 8-character code of the form IELnnnnI, where:

- The first three characters "IEL" identify the message as coming from the optimizing compiler.

- The next four characters are a 4-digit message number.

- The last character "I" is an operating system code indicating that the message is for information only.

The text of each message, an explanation, and any recommended response, are given in the messages publication for this compiler.

Type of message	Option
Informatory	FLAG(I)
Warning	FLAG(W)
Error	FLAG(E)
Severe Error	FLAG(S)
Unrecoverable Error	Always listed

Figure 3. Selecting the lowest severity of messages to be printed, using the FLAG option.

RETURN CODES

For every compilation run, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved. This code is printed on the message beginning with "Execution terminated" provided that the MTS RCPRINT option is not set to OFF.

This code can also be referenced as RUNRC in the MTS $IF command. The preprocessor %NOTE statement may also be used to set the return code for user-generated messages. The meanings of the codes are given in Figure 4.

Return Code	Meaning
0	No error detected; compilation completed; successful execution anticipated.
4	Possible error (warning) detected; compilation completed; successful execution probable.
8	Error detected; compilation completed; successful execution probable.
12	Severe error detected; compilation may have been completed; successful execution improbable.
16	Unrecoverable error detected; compilation terminated abnormally; successful execution impossible.

Figure 4. Return codes from compilation of a PL/I program.

MULTIPLE COMPILATION

Multiple compilation allows the compiler to compile more than one external PL/I procedure in a single $RUN command. The compiler creates an object module for each external procedure and stores it sequentially on logical I/O units SPUNCH or 0. Multiple compilation can increase compiler throughput by reducing operating system and compiler initialization overheads.

To specify multiple compilation, include a compiler PROCESS statement as the first statement of each external procedure except possibly the first. The PROCESS statements identify the start of each external procedure and allow compiler options to be specified individually for each compilation. The first procedure may require a PROCESS statement of its own, because the options in the PAR field of the $RUN command apply to all external procedures, and may conflict with the requirements of subsequent procedures.

The method of coding a PROCESS statement and the options that may be included are described under "Optional Facilities," earlier in this section. The options specified in a PROCESS statement apply to the compilation of the source statements between that PROCESS statement and the next PROCESS statement. Options other than these, either the defaults or those specified in the PAR field, will also apply to the compilation of these source statements.

SIZE Option

In a multiple compilation, the SIZE specified in the first external procedure (by a PROCESS statement or a PAR field of the $RUN command, or by default) is used throughout. If SIZE is specified in subsequent external procedures, it is diagnosed and ignored. The compiler does not reorganize its storage between external procedures.

Return Codes in Multiple Compilation

The return code generated by a multiple compilation is the highest code that would be returned if the procedures were compiled separately.

Example:

```
     $RUN *PL1OPT SPUNCH=OBJ1 0=OBJ2
     /PROCESS DECK;

          First PL/I source program

     /PROCESS NODECK,OBJECT;

          Second PL/I source program

     /PROCESS;

          Third PL/I source program

     $ENDFILE
```

In this example, there is only one $RUN command; thus, all I/O assignments necessary for the three external procedures must be included on it. Since the compiler options are not specified in the $RUN command, the defaults DECK and NOOBJECT apply to all external procedures, unless overridden by a PROCESS statement. Hence, the file OBJ1, assigned to the unit SPUNCH, has the object module for the first and last external procedures, and the file OBJ2, assigned to the unit 0, has the object module of the second external procedures.

COMPILE-TIME PROCESSING (PREPROCESSING)

The preprocessing facilities of the compiler are described in the language reference manual for this compiler. Statements can be included in a PL/I program that, when executed by the preprocessor stage of the compiler, modify the source program or cause additional source statements to be included from a library. The following discussion supplements the information contained in the language reference manual by providing some illustrations of the use of the preprocessor and explaining how to establish and use source statement libraries.

Invoking the Preprocessor

The preprocessor stage of the compiler is executed if the MACRO compiler option is specified. The compiler and the preprocessor use a line file named -##SYSUT1 during the processing. They also use this file to store the preprocessed source program until compilation begins. This file is automatically created and emptied by the compiler whenever necessary.

The term MACRO owes its origin to the similarity of some applications of the preprocessor to the macro language available with such processors as the IBM System/360 Assembler. Such a macro language allows the user to write a single instruction in a program to represent a sequence of instructions that have previously been defined. The format of the preprocessor output is given in Figure 5.

Three other compiler options, MDECK, INSOURCE, and SYNTAX, are meaningful only when the MACRO option is specified.

A simple example of the use of the preprocessor to produce a source deck for a procedure SUBFUN is shown in Figure 6; according to the value assigned to the preprocessor variable USE, the source statements will represent either a subroutine or a function. The preprocessor output is written to the logical I/O unit SPUNCH. Note that the program *MACUTIL is invoked to generate the actual source library. Normally compilation would continue and the preprocessor output would be compiled. If the object module is desired, NODECK and OBJECT should be specified so that the logical I/O unit SPUNCH will not be a mixture of the preprocessor output and object modules.

Column 1	Printer control character, if any, transferred from the position specified in the MARGINS option.
Columns 2-72	Source program. If the original source program used more than 71 columns, then additional lines are included for any lines that need continuation. If the original source program used less than 71 columns, then extra blanks are added on the right.
Columns 73-80	Sequence number, right-aligned. If either SEQUENCE or NUMBER apply, this is taken from the sequence number field. Otherwise, it is a preprocessor generated number in the range 1 through 99999. This sequence number will be used in the listing produced by the INSOURCE and SOURCE options and in any preprocessor diagnostic messages.
Column 81	blank
Columns 82,83	Two-digit number giving the maximum depth of the replacement by the preprocessor for this line. If no replacement occurs, the columns are blank.
Column 84	"E" signifying that an error has occurred while replacement is being attempted. If no error has occurred, the column is blank.

Figure 5. Format of the preprocessor output

```
$empty -source
$run *pl1opt spunch=-source par=MACRO,NOSYNTAX,MDECK
 SUBFUN: PROCEDURE(CITY);
    DECLARE IN FILE RECORD,
        1 DATA,
          2 NAME CHARACTER(10),
          2 POP FIXED(7),
        CITY CHARACTER(10);
    %DECLARE USE CHARACTER;
    %USE='FUN';  /* FOR SUBROUTINE, SUBSTITUTE %USE='SUB' */
    OPEN FILE(IN);
 NEXT: READ FILE(IN) INTO(DATA);
    IF NAME=CITY THEN DO;
    CLOSE FILE(IN);
    %IF USE='FUN' %THEN %GOTO L1;
    PUT FILE(SYSPRINT) SKIP LIST(DATA); END;
    %GO TO L2;
 %L1:; RETURN (POP); END;
 %L2:;
 END SUBFUN;
$ENDFILE
$run *macutil 0=newlib guser=*source*
update@name=fun@¬hdrgen -source
stop
$ENDFILE
```

Figure 6. Using the preprocessor to produce a source deck which is placed on a source program library.

The %INCLUDE Statement

The language reference manual for this compiler describes how to use the %INCLUDE statement to incorporate source text from a library into a PL/I program. (A library is a file that consists of "macros".) Source text that is inserted into a PL/I program by means of a %INCLUDE statement must exist either as a "macro" within a library or as a file with the same name. Source libraries used by the compiler can be defined as logical I/O units 1 through 19.

The syntax of %INCLUDE statements is different than in the OS version. The %INCLUDE statement may include one or more identifiers. Each identifier may be up to 31 characters. The identifier is usually a "macro" in a library. If no macro exists, the PL/I Optimizing compiler will search for a file that has the same name as the identifier. For example,

```
%INCLUDE INVERT, LOOPX;
```

specifies that the source statements in "macros" INVERT and LOOPX are to be inserted consecutively into the source program.

A PROCESS statement in source text included by a %INCLUDE statement will result in an error in the compilation.

The use of a %INCLUDE statement to include the source statements for SUBFUN in the procedure TEST is shown in Figure 7. The library NEWLIB is defined in the logical I/O unit 1.

```
$empty -obj
$empty -prt
$run *pl1opt 1=newlib spunch=-obj sprint=-prt par=M
 TEST: PROCEDURE OPTIONS(MAIN);
        DECLARE NAME CHAR(10),
                NO FIXED(7);
        ON ENDFILE(SYSIN) GO TO FINISH;
 AGAIN: GET FILE(SYSIN) LIST(NAME);
        NO=SUBFUN(NAME);
        PUT DATA(NAME,NO);
        GO TO AGAIN;
  %INCLUDE FUN;
  FINISH: END TEST;
$ENDFILE
```

Figure 7.   Including source statements from a library.

LOADING A PL/I PROGRAM

The PL/I object program should be concatenated with one of two standard PL/I libraries. The PL/I (F) library resides in the public file *PL1LIB (an alternate name is *PL/1LIB); the PL/I Optimizing Compiler library resides in the public file *PL1OPTLIB. *PL1LIB can only be used with the PL/I (F) programs, and *PL1OPTLIB with the programs produced by the PL/I Optimizing compiler. A typical command showing such concatenation is given below:

```
     $RUN object+*PL1LIB SCARDS=input SPRINT=output
```

where "object" is the file or device containing the PL/I object deck, "input" is any file or device name (defaults to *SOURCE*) to be attached to the logical I/O unit SCARDS, and "output" is a file or device name for the logical I/O unit SPRINT (defaults to *SINK*). Other commands that load a PL/I program are $LOAD and $DEBUG. $LOAD loads the program without executing it; execution may be started by the $START command. $DEBUG initiates a debugging session using the Symbolic Debugging System (SDS).

Should the user forget to specify *PL1LIB or *PL1OPTLIB, he will be prompted as follows:

```
     $run pgm
     .  THERE ARE 9 UNDEFINED SYMBOLS
     .  ENTER LOCN OF MORE LOADING INPUT, "CANCEL", "IGNORE",
     .  "USMSG", "UXREF", OR "MAP":
     ?*pl1lib
```

Here, the MTS loader loads the object file PGM. The message "THERE ARE 9 UNDEFINED SYMBOLS" indicates that the user forgot to add the PL/I library required to resolve these undefined PL/I library symbols.

There are two methods to run a PL/I program without having to explicitly specify the PL/I library. One method is to issue the following MTS command before running a program:

```
     $SET LIBSRCH=*PL1LIB
```

or

```
     $SET LIBSRCH=*PL1OPTLIB
```

This causes the loader to automatically search *PL1LIB when there are unresolved symbols. (Setting the LIBSRCH option for PL/I (F) programs is not recommended if the user will also be running non-PL/I programs since *PL1LIB contains a special version of the subroutine SYSTEM that will cause a program interrupt if called from a non-PL/I program. SYSTEM, on the other hand, does not reside in *PL1OPTLIB.) The other method is to use implicit concatenation by inserting the following line at the end of the user's object program file:

```
$CONTINUE WITH *PL1LIB
```

or

```
$CONTINUE WITH *PL1OPTLIB
```

The object file is then implicitly concatenated with *PL1LIB or *PL1OPTLIB. There must be <u>exactly</u> <u>one</u> <u>blank</u> between the words $CONTINUE and WITH and between WITH and *PL1LIB or *PL1OPTLIB. The word $CONTINUE must start in the first column position.

To ensure that the program will run, only one external procedure with OPTIONS(MAIN) should exist. If none exists, execution will promptly be terminated by the following message:

```
IHE006I - NO MAIN PROCEDURE
```

or

```
IBM006I - NO MAIN PROCEDURE, PROGRAM NOT EXECUTED
```

The following is the list of reserved symbols that cannot be declared EXTERNAL during any PL/I (F) compilation:

```
DDEF#         PGNTTRP       SCANSTOR      SPIE
ERROR#        PL1SYM        SERCOM        STDDMP
GETSPACE      QUIT$         SNAP          SYSTEM#
LCSYMBOL
```

These, plus all symbols beginning with letters IHE, are normally called by PL/I (F) routines.

The following is the list of reserved symbols that cannot be declared EXTERNAL other than EXTERNAL ENTRY during any compilation of a program by the PL/I Optimizing Compiler:

```
ATTNTRP       GUINFO        LOAD          PGNTTRP
ERROR#        GUSER         LOADINFO      SERCOM#
FREESPAC      LCSYMBOL      MTS#          UNLOAD
GETSPACE
```

These, plus all symbols beginning with letters IBM, are normally called by PL/I Optimizer routines.

In addition, there is a limit for the number of pseudo-registers used. A pseudo-register is assigned for every file and every controlled variable. The PL/I (F) compiler assigns a pseudo-register for every PROCEDURE statement and for every BEGIN statement. The PL/I Optimizing compiler assigns a pseudo-register for every fetched procedure. The sum of these pseudo-registers may not exceed 1000. If this number is exceeded, the following message is generated:

```
     IHE005I - PSEUDO-REGISTER VECTOR TOO LONG - PROGRAM NOT EXECUTED.
```

or

```
     IBM005I - TOO MANY FILES AND CONTROLLED VARIABLES
```

The PL/I compiler also changes the names of some external files, so that the file names are not confused with actual MTS subroutines. The list is:

File	PL/I (F)	PL/I Optimizer
GUSER	IHEGUSR	_GUSER
SCARDS	IHESCDS	_SCARDS
SERCOM	IHESRCM	_SERCOM
SPRINT	IHESPRT	_SPRINT
SPUNCH	IHESPCH	_SPUNCH

Link-Editing a PL/I Program

WARNING: There exist current serious problems with *LINK-EDIT with object programs produced by PL/I Optimizing compiler. A message "IBM534I PROTECTION EXCEPTION" may result during the execution of the link-edited programs.

Object modules produced by the PL/I compiler are particularly noncompact, i.e., they are not optimized either with respect to loading time or file-storage requirements. The basic compiler-generated loader records consist of a 16-byte header and a variable-length field which can be up to 240 bytes in length. However, the PL/I compiler produces card-image object modules with an average length that is considerably less than 80 bytes. Hence, a very simple but effective method of reducing both the loading time and file-storage requirements of a PL/I program is to reduce the total number of records by increasing the average record length, i.e., by making the object modules as compact as possible. The object-file editor (*OBJUTIL) and the linkage editor (*LINKEDIT) are programs available for this purpose.

Users who only want to reformat their PL/I object modules may simply issue the command

```
$RUN *LINKEDIT SCARDS=inFDname SPUNCH=outFDname
```

In this case, object modules are read from "inFDname", converted into maximum-sized records, and written to "outFDname". A 2800-line PL/I test program, for example, was compiled into an object module with 2696 records. By using the linkage editor to reformat the module, this module was compressed into 265 records in a line file. The effect on loading time (on the Amdahl 470V/8) and file-storage requirements was as follows:

	Load Time	File Storage
Original PL/I module	0.42 seconds	62 pages
Compressed PL/I module	0.15 seconds	39 pages

Either the object-file editor or the linkage editor may be used to compress an object file. The linkage editor is more expensive, but it produces a slightly more compressed object module since it reorders the RLD items. However, only the linkage editor may be used to combine object modules (see below).

A second, and more complex, method of reducing the loading time and file-storage requirements is to combine into one module a collection of object modules which are always loaded together. This can be very effective for programs that consist of a large number of subprograms which were written and compiled independently for debugging purposes, and are now reliable enough to be heavily used. This further optimization may be accomplished by use of the linkage editor command language. The following sequence of commands will convert the PL/I object program in "inFDname" to a completely optimized form in "outFDname":

```
$RUN *LINKEDIT
INCLUDE inFDname
COMBINE
PURGE ALLBUT IHEMAIN IHENTRY IHESPRT
PUNCH outFDname
STOP
```

It should be noted that this process is irreversible. No information is retained concerning the previously independent status of a module. As an example of this further optimization, consider a collection of 11 independent modules that were compressed into a line file by the linkage editor:

	Records	Load Time	File Storage
Original PL/I module	8063 lines	1.02 seconds	182 pages
Compressed module	981 lines	0.38 seconds	116 pages
Compressed and combined	145 lines	0.20 seconds	69 pages

In the linkage editor command sequence above, the PURGE command is used to remove the entry point names that were required by the dynamic loader to link the independent modules. These names are no longer required after the combination operation performed by the COMBINE command. However, the symbols IHEMAIN and IHENTRY which are referenced by the PL/I library should not be purged. This can be achieved by the use of the ALLBUT option, e.g.,

```
     PURGE ALLBUT IHEMAIN IHENTRY
```

In addition, if only part of a PL/I object program is combined into one module, those PL/I external variables to be shared with other routines must not be purged. For example, consider a program consisting of four modules A, B, C, and D, all of which share a variable X declared EXTERNAL. If modules A, B, and C are combined into one module by the command

```
     COMBINE A B C
```

the symbol X must not be purged since the module D refers to it. In this case, the PURGE command should be specified as

```
     PURGE ALLBUT X IHEMAIN IHENTRY
```

Complete details on the object-file editor and linkage editor are given in MTS Volume 5, System Services.

RUNNING A PL/I PROGRAM

The PL/I program may be executed with the MTS command:

```
     $RUN object+pl1lib [logical I/O specifications]
          [PAR=[run-time options][;program parameters]]
```

The components of the $RUN command are as follows:

Object Program

"object" is the MTS file or device containing the object code compiled by PL/I (F) or PL/I Optimizing compiler.

"pl1lib" is either *PL1LIB or *PL1OPTLIB according to whether the "object" was compiled by the PL/I (F) or the PL/I Optimizing compiler, respectively.

Logical I/O Specifications

"logical I/O specifications" are MTS logical I/O units, each with its assigned file or device. Record formats should not be explicitly specified. The list of valid MTS logical I/O units plus the default assignments are:

SCARDS=*SOURCE*	
SPRINT=*SINK*	
SPUNCH=*PUNCH*	(default in batch mode, if global card output estimate is greater than zero; otherwise, no default)
SERCOM=*MSINK*	(output produced by the DISPLAY statement)
GUSER=*MSOURCE*	(input read by the REPLY option of the DISPLAY statement)

The PL/I Optimizing compiler equates the standard PL/I files SYSIN and SYSPRINT to MTS logical I/O units SCARDS and SPRINT, respectively. The PL/I (F) compiler uses the PL/I file names SCARDS and SPRINT, where GET and PUT statements do not specify the FILE options.

The logical I/O units 0 through 99 have no default specifications and can be used in PL/I only if a PL/I file is opened with the TITLE option, e.g.,

```
          OPEN FILE(TEMP) TITLE('0');
```

This is due to a severe PL/I nomenclature requiring that each PL/I identifier, including file names, start with a letter instead of a number.

Run-Time Options

The only run-time options allowed for PL/I (F) users are program file specifications discussed below. In addition to the program file specifications, PL/I Optimizing compiler allows Run-Time Options discussed in the next section.

Program File Specifications

PL/I routines allow the user freedom in choosing PL/I file names other than the 105 MTS logical I/O units. Each PL/I file should be associated with an MTS file or device name, and each PL/I file specification must be separated from the other by one or more blanks (commas are not allowed as separators). For example, a program having two PL/I file names INPUT and OUTPUT may be specified thus:

```
$RUN PGM+*PL1LIB PAR=INPUT=*SOURCE* OUTPUT=*SINK*
```

Here *SOURCE* is attached to the PL/I file INPUT, and *SINK* to the PL/I file OUTPUT. If a terminal PL/I (F) user neglects to specify PL/I files in the $RUN command, he will be prompted to specify them; thus,

```
#$run pgm+*pl1lib
#EXECUTION BEGINS
 INPUT   - SPECIFY FDNAME OR SEND END-OF-FILE
?*source*
 OUTPUT  - SPECIFY FDNAME OR SEND END-OF-FILE
?*sink*
 ...
```

Here the PL/I routines print the PL/I file name INPUT and the message "SPECIFY FDNAME OR SEND END-OF-FILE". The user should enter an MTS file or device name following the question mark prefix. Should the user enter an end-of-file, the UNDEFINEDFILE condition will be raised and unless the program has been written to handle this condition, it will be terminated with an error comment. This will always happen in batch mode if such a file is not specified in the PAR field of the $RUN command.

Alternatively, the user may call the subroutine ATTACH to attach FDnames to PL/I files. For example, he can insert in the program:

```
CALL ATTACH ('PROMPT=*MSOURCE*');
```

which attaches *MSOURCE* to the PL/I file PROMPT.

The PL/I Optimizing Compiler always raises the UNDEFINEDFILE condition if the user neglects to specify the PL/I files in the $RUN command. The following statements can prompt the user:

```
     ON UNDEFINEDFILE(PROMPT) BEGIN;
        DECLARE CANREPLY ENTRY OPTIONS(RETCODE ASSEMBLER),
                PLIRETV BUILTIN,
|               ATTACH ENTRY CHARACTER(*)),
                ANSWER  CHARACTER(72) VARYING;
        ON UNDEFINEDFILE(PROMPT) SYSTEM;
        CALL CANREPLY;
        DISPLAY ('PROMPT FILE IS NOT SPECIFIED');
        IF PLIRETV=0 THEN
           DISPLAY ('SPECIFY FDNAME OR CANCEL')
                   REPLY (ANSWER);
|          IF ANSWER='CANCEL' THEN SIGNAL UNDEFINEDFILE(PROMPT);
           CALL ATTACH ('PROMPT='||ANSWER);
           OPEN FILE (PROMPT);
|       ELSE SIGNAL UNDEFINEDFILE(PROMPT);
        END;
```

Program Parameters

The semicolon must be used to separate the program file specifications from the program parameters. (The IBM equivalent for the semicolon is the slash "/". Since, however, slashes are allowed for MTS FDnames, the semicolon is chosen for MTS implementation). This is true even if no PL/I file assignments are made in the PAR field. The parameter string may contain any options that the PL/I program offers.

For PL/I (F) programs, the parameter string in the PAR field of the $RUN command is passed intact to the main procedure. This parameter string includes PL/I file specifications such as below:

```
     $RUN PGM+*PL1LIB PAR=SPRINT=FILEOUT;ABC
```

Here the parameter string passed to the main procedure is "SPRINT=FILEOUT;ABC".

For PL/I Optimizer programs, only the string after the semicolon, if any, in the PAR field is passed, e.g., "ABC".

If the PL/I (F) user desires to separate in the parameter string the file specifications and the program parameters, he may code the following in the program.

```
     PGM:     PROCEDURE (STRING) OPTIONS(MAIN);
              /* Specify the string declarations */
              DECLARE (STRING, PARSTRING)
                      CHARACTER(255) VARYING;
```

```
            /* Search for the semicolon */
                 I = INDEX ( STRING, ';' );
            /* Check if the semicolon really exists */
                 IF I = 0 |    /* None */
                    I = LENGTH(STRING); /* Or just at end */
                    THEN PARSTRING='';
                         /* It is a null string */
                 ELSE /* Otherwise, set it to a substring
                         after the semicolon */
                    PARSTRING=SUBSTR(STRING,I+1);
                         .
                         .
                         .
```

For the PL/I Optimizer programs, program parameters are very simple. The above program should be replaced as follows:

```
     PGM:      PROCEDURE (PARSTRING) OPTIONS (MAIN);
               DECLARE PARSTRING CHARACTER(255) VARYING;
                    .
                    .
                    .
               END PGM;        /* End of program */
```

If the user wants only the MTS logical I/O units and does not want any PL/I file specifications (or the semicolon) in the PAR field, two methods exist depending on whether the program was compiled by the PL/I (F) compiler or the PL/I Optimizing compiler.

If the program was compiled by the PL/I (F) compiler, the user may run *LINKEDIT and rename IHESAPA or IHESAPC as IHESAPE. IHESAPA is invoked if no optimization is requested, i.e., OPT=0, and IHESAPC is used if either OPT=1 or OPT=2 is specified during the compilation of the PL/I program. Both IHESAPA and IHESAPC process the PL/I file specifications in the PAR field; IHESAPE, on the other hand, does not process them but passes the entire PAR field to the main procedure.

If the program was compiled by the PL/I Optimizing compiler, the user should copy the object program in the file PL1:PASS_PAR to the front of his program. This causes PL/I routines to treat the PAR field as the user's PAR field. The assembler source lines are in the negative lines of the file as follows:

```
     PASS_PAR CSECT
              USING PASS_PAR,15
              MVC   LOCPAR,0(1)     Move locator.
              NI    LOCPAR,X'7F'
              L     1,0(0,1)        Need to copy length.
              LH    1,0(0,1)
              STH   1,LOCLEN
              LA    1,MYPAR         Points to the locator.
              L     15,PLICALL      Now call it.
```

```
                   BR     15
          PLICALL  DC     V(PLICALLA)
          MYPAR    DC     X'80',AL3(LOCPAR)
          LOCPAR   DS     A                Where is the string.
          LOCLEN   DS     H                The length.
                   DC     X'8000'          It's a varying string.
                   END    PASS_PAR
```

PL/I FILE SPECIFICATIONS

Each PL/I file name can be associated with an MTS file or device name; thus,

```
filename=FDname[(b,e,i)][@modifiers •••][+•••]
```

where:

filename	is a PL/I file name or corresponding TITLE option of an OPEN statement. Only the first eight characters of the PL/I file name are used.
FDname	is the name of a file or device.
(b,e,i)	is the line number range. "b" is the beginning line number, "e" the ending line number, and "i" the increment. The defaults are (1.000, 99999.999, 1.000).
@modifiers	are any legal MTS I/O modifiers (see MTS Volume 1) or PL/I record format modifiers. If PL/I record format modifiers are given, the assignment must be made in the PAR field of the $RUN command rather than as a standard MTS logical I/O unit assignment.
+•••	is an explicit concatenation of FDnames composed of the above parts.

PL/I record formats are usually specified either at run time or in the ENVIRONMENT attribute of the PL/I file. The following record formats are supported:

```
U[A|M][(maximum blocksize)]
V[B][S][A|M][(maximum blocksize[,maximum recordsize])]
F[B][A|M][(maximum blocksize[,recordsize])]
```

PL/I Optimizer users can also add:

```
D[B][A][(maximum blocksize[,recordsize])]
RECSIZE(recordsize)
BLKSIZE(maximum blocksize)
ASCII
BUFOFF(buffer_offset)
```

The use of record formats is explained in the section "PL/I Input/Output in MTS."

Record format specification at run time allows the user to change the default record formats of PL/I input and output files without having to recompile the program. In the following example, the record formats are specified at execution time in the PAR field of the $RUN command.

```
     $RUN FMAINT+*PL1LIB PAR=INTER=*TAPE*@FB(2400,600)
          MASTER=*OLDMAS*@U(132) EXCEPTN=E@U(300) SCARDS=*SOURCE*@F(80)
```

Note that the specifications are separated by blanks, not by commas.

Alternatively, record format specifications can be given in the source program, using the ENVIRONMENT attribute of a PL/I file. Refer to the section "PL/I Input/Output in MTS" or to the language reference manual for the compiler being used, for details.

UNDEFINEDFILE CONDITION

When a PL/I file is not specified or is incorrectly specified, the PL/I Optimizer routines will raise the UNDEFINEDFILE condition. The user may provide an ON UNDEFINEDFILE unit to prompt the file specification. This uses the ONCODE built-in function, which returns an oncode that indicates what happened. The oncodes are:

Oncode	Condition
80	UNDEFINEDFILE condition was raised by the SIGNAL statement.
81	Conflict in file attributes between attributes in a DECLARE statement and in an OPEN statement.
82	Conflict in file attributes with actual file or device, e.g., read-only file opened with the output attribute or terminal opened with the backwards attribute.
83	Incomplete file or device specification. No block size, no record format, or no key length. Usually does not occur since defaults apply.
84	PL/I file is not specified with a file or device. The file can be specified in a PLIXOPT external static varying character string, in the PAR field, or by a call to the ATTACH subroutine.
86	Line size greater than the maximum or invalid value in an ENVIRONMENT option such as invalid KEYLOC or BUFOFF.
87	Invalid record or block size; conflict with record format.
92	Nonexistent or invalid file or device, access not allowed, wait interrupt, or file deadlock.
93	Files attributes that cannot be implemented in MTS, e.g., KEYED files, VSAM environment option, etc.

RUN-TIME OPTIONS

For each execution of a PL/I Optimizer program, the default for an run-time option will apply, unless it is overridden by a PLIXOPT string in the source program or by the PAR field of the $RUN command.

An option specified in the PLIXOPT string overrides the default value, and an option specified in the PAR field overrides that specified in the PLIXOPT string.

SPECIFYING RUN-TIME OPTIONS IN THE PLIXOPT STRING

Run-time options can be specified in a source program by means of the following declaration:

```
     DECLARE PLIXOPT CHARACTER(len) VARYING
             INITIAL('strg') STATIC EXTERNAL;
```

where "strg" is a list of options separated by commas, and "len" is a constant equal to or greater than the length of "strg".

If more than one external procedure in a $RUN command declares PLIXOPT as STATIC EXTERNAL, only the first string will be used.

SPECIFYING RUN-TIME OPTIONS AND MAIN-PROCEDURE PARAMETERS IN THE $RUN COMMAND

Run-time options may be specified in the PAR field as follows:

```
     $RUN OPT+*PL1OPTLIB PAR=ISA(10P)
```

The PAR field also can be used to pass an argument to the PL/I main procedure. To do so, place the argument, preceded by a semicolon, after the run-time options. For example:

```
     $RUN OPT+*PL1OPTLIB PAR=ISA(10P);ARGUMENT
```

If an argument is to be passed without specifying options, it must be preceded by a semicolon. For example:

```
     $RUN OPT+*PL1OPTLIB PAR=;ARGUMENT
```

RUN-TIME OPTIONS

The following paragraphs describe the run-time options, which can be specified in the PAR field of the $RUN command or in the PLIXOPT string. Figure 1 lists the options by function.

Run-Time Options Listed by Function

OPTION(default underlined)	USE
Storage Control	
ISASIZE	Control initial allocation of working storage.
REPORT|NOREPORT	Generate report of storage usage.
Debugging	
COUNT[1]|NOCOUNT	List number of times each statement is executed.
FLOW(n,m)[1]|NOFLOW	List last "n" branches and "m" changes of procedure.
Error Handling	
PGNT|NOPGNT	Allow program check interrupts to be handled by PL/I (PGNT) or passed to system (NOPGNT).
ATTN[2]|NOATTN	Allow attention interrupts to be handled by PL/I (ATTN) or passed to system (NOATTN).
Program File Specification	
PL1FILE=MTSFILE	If an equal sign is present after an "run-time option", then the option is considered as the name of a PL/I file and specifies an MTS file/device to be attached.

[1]Only works if the FLOW or COUNT was specified at compile time. Default is what was specified at compile time.
[2]ATTN option is ignored if INTERRUPT was specified at compile time.

Figure 1. Run-time options

COUNT	specifies that a count is to be kept of the number of times each statement in the program is executed and the results are to be printed when the program terminates. This option is discussed in greater detail under "Run-Time COUNT Option" later in this section.
NOCOUNT	specifies that statement counting is not to be performed.
FLOW[(n,m)]	specifies that a list of the most recent transfers of control in the execution of the program is to be generated. This option is discussed in greater detail under "Run-Time FLOW Option" later in this section.
NOFLOW	specifies that a flow list is not to be produced.

ISASIZE specifies the size of the Initial Storage Area (ISA). The ISA is the main storage acquired by the PL/I program and retained for particular uses during execution. This option may be abbreviated to ISA. The option has the format:

```
ISASIZE([-]x|[-]xK|[-]xP|0)
```

For example:

```
ISASIZE(2000) or ISASIZE(2P) or ISASIZE(-8000)
```

where:

x	is the size of the initial storage area. If "x" is positive, it specifies the ISASIZE. If it is negative, the maximum storage amount (currently 256 pages) is first obtained and then reduced by the "x" bytes. If "x" is postfix by "K" or "P", then "x" is multiplied by 1024 or 4096, respectively.
0	Obtains the maximum storage amount. This is currently 256 pages or 1,048,576 bytes.

REPORT specifies that a report of the use of storage by a program will be generated and placed on the MTS logical I/O unit SERCOM at the end of execution. This option may be abbreviated to R. A description of the output and how to make use of it is given later in this section under "Run-Time Storage Requirements".

REPORT output is headed by the name of the main procedure and the time and date at the end of execution. The user’s own identifier also can be supplied using the PLIXHD string (see "Using PLIXHD to identify COUNT and REPORT Output").

The use of the REPORT option degrades performance.

NOREPORT	specifies that a report is not required. This option may be abbreviated to NR.
PGNT	specifies that when a program interrupt occurs, the PL/I error handler is to be invoked. Under certain circumstances the ERROR condition will be raised.
NOPGNT	specifies that program interrupts will not be trapped by the PL/I error handler.
ATTN	specifies that when an attention interrupt occurs, the PL/I error handler is to be invoked. If there is no ON ATTENTION unit, or if the ON ATTENTION unit specifies SYSTEM, the following is printed:

```
IBM091I ONCODE='0400'. 'ATTENTION' CONDITION RAISED.
```

And then MTS is called. The user may issue a $RESTART command to resume the execution.

If, on the other hand, there exists an ON ATTENTION unit without SYSTEM, the unit is entered; upon return, the program is restarted unless the ON-unit issues a GOTO statement.

If a program happens to be compiled with an option INTERRUPT, only a switch indicating an attention interrupt occurred is set on. The ATTENTION condition can only be raised at the breakpoints inserted by the compiler, and the stream I/O routines. The ATTENTION condition can be raised nowhere else; so it is recommended that all external procedures should be compiled with NOINTERRUPT option.

NOATTN	specifies that attention interrupts will not be trapped by the PL/I error handler.

Program file specifications

These are indicated by the presence of the equal sign. For example, FLOW=OUTPUT indicates that an MTS file named OUTPUT is to be attached to a PL/I file FLOW. Without the presence of the equal sign, they are considered as run-time options, e.g., FLOW(25,10).

USING PLIXHD TO IDENTIFY COUNT AND REPORT OUTPUT

When COUNT or REPORT output is generated and the program contains a static external character variable called PLIXHD, the value in PLIXHD is printed at the head of the output after the name of the main procedure and the date and time of execution. This allows an identifier to be supplied for such output.

To do this, PLIXHD must be declared as STATIC EXTERNAL CHARACTER VARYING. (STATIC may be omitted because all EXTERNAL data is STATIC by default). For example:

```
     DECLARE PLIXHD EXTERNAL CHARACTER(50) VARYING
             INITIAL ('THIS IS A PLIXHD MESSAGE');
```

The printed output of PLIXHD is limited to one line and is truncated, if necessary. The result of using PLIXHD as shown above would be:

```
     STORAGE MANAGEMENT REPORT FOR PROCEDURE P
     DATE 21 JULY 1982 TIME 14.47.15.00
     THIS IS A PLIXHD MESSAGE
     (Report Output goes here)
```

RUN-TIME STORAGE REQUIREMENTS

During the execution of a program, the storage is divided into three areas: the program itself, the ISA (Initial Storage Area), and the remainder, called residual storage in this discussion.

The program, including the compiled code, constants, and storage for STATIC variables, occupies the first area. The second area (ISA) is used for storage of all variables that are not STATIC and certain housekeeping fields. These are referred to as PL/I storage. The third area, residual storage, is used as an overflow area for the ISA and, consequently, may be used for PL/I storage.

The ISA is acquired by the PL/I program at the start of execution and retained until termination. Consequently, obtaining and freeing of storage within it can be managed by the PL/I program without resorting to system facilities. Thus, the overhead of obtaining and freeing storage within the ISA are small compared with using the residual area where GETSPACE and FREESPAC subroutines have to be called. Execution is, therefore, faster if all PL/I storage is contained in the ISA. However, if significant parts of the ISA remain unused throughout long periods during the execution of a program, space is wasted. The fact that ISA storage is quickly acquired and freed, but conversely may only be used for certain items makes the choice of ISA size a critical factor in determining both the time and space requirements of the program.

When the REPORT option is in force, the use of storage is monitored and a report is generated at the end of the program. The report is transmitted to the file associated with the MTS logical I/O unit SERCOM and is identified by the name of the main procedure and the date and time of execution. Optionally, the user can generate a further report identifier by use of PLIXHD. The REPORT option should only be used while the ISA size is being determined. Report generation involves a considerable execution-time overhead and should be removed as soon as possible. REPORT should be used after COUNT and FLOW have been removed, because COUNT and FLOW use extra storage and thus make the report inaccurate.

USING THE REPORT OPTION

When using the REPORT option, the best strategy to ensure satisfactory results is to specify a very large ISASIZE so that the chances of all PL/I storage being within the ISA are high. This gives the most accurate estimate of PL/I storage used, and thus the most accurate indication of the ISA size required. The ISA size should then be set to the size of the PL/I storage used, and the program should be run again with the REPORT option to determine if the ISA size is satisfactory. Bear in mind that different data or different paths through the program may result in different storage requirements. If it is impractical to specify a large ISA, an alternative is to specify a value of 1. This results in the minimum acceptable ISASIZE being used. This minimum is such that PL/I storage for the first and all subsequent blocks will be met from residual storage. The disadvantage of this method is that it may slightly overestimate the total amount of PL/I storage used. Because of the method of measurement used, an ISASIZE where PL/I storage is partly inside and partly outside the ISA gives the least satisfactory result.

The output caused by the REPORT option for a program is shown with explanatory notes in Figure 2.

USING THE REPORT OUTPUT

Figure 2 shows the output from the REPORT option. An ISA size equal to the "Amount of PL/I Storage Required" value in the report will give the fastest execution time, because it will allow all PL/I storage to be obtained within the ISA. However, the overall size requirements may be increased, for example, if a program uses large BASED or CONTROLLED variables for a short time during execution. If a seldom used subroutine contains a number of large variables, use of an ISASIZE equal to the "PL/I Storage Required" figure may be uneconomical as it will require an unnecessarily large storage area.

Example of Report Output

```
STORAGE MANAGEMENT REPORT FOR PROCEDURE J
DATE 4 JULY 1976 12.00.00.00

ISASIZE SPECIFIED 2680 BYTES     The size specified in the ISASIZE
                                 option.  If  the  option  is  not
                                 used, 0 is given.
LENGTH OF INITIAL STORAGE AREA (ISA) 2680 BYTES
                                 The  length used.  Normally, this
                                 is the length  specified  or  the
                                 default.  However, if this is not
                                 large enough for the requirements
                                 of the first block, another value
                                 is used.
AMOUNT OF PL/I STORAGE REQUIRED 2680 BYTES
                                 The  maximum  amount  of  storage
                                 that could have used the ISA.  It
                                 is the optimum  ISASIZE  in  most
                                 conditions,   but  see  text  for
                                 provisos.
AMOUNT OF STORAGE OBTAINED OUTSIDE THE ISA 0 BYTES
                                 Overflow of ISA, if any.  0 means
                                 none.
NUMBER OF GETMAINS 0             Number of times ISA overflowed.
NUMBER OF FREEMAINS 0            Number of times ISA overflow  was
                                 freed.
NUMBER OF GET NON-LIFO REQUESTS 2
NUMBER OF FREE NON-LIFO REQUESTS 1
                                 Non-LIFO  storage is storage that
                                 is not attached to  a  block,  as
                                 opposed to AUTOMATIC storage that
                                 is.   For example, BASED and CON-
                                 TROLLED storage.  For a full  de-
                                 scription,    see   the  Execution
                                 Logic manual.
```

Figure 2. Report output and its meaning.

If a program has to run in the smallest possible area, it is normally best to use an ISA size of 1. This results in all storage requests being made within the residual area, thus all spare storage is available for all purposes. However, this method does have a disadvantage when a large number of small items, such as based variables, have to be allocated since each item requires eight additional bytes for chaining.

When an optimum ISA size has been determined, the program should be rerun with this size specified and the REPORT option still in force so that the results can be checked. When they are satisfactory, the REPORT option should be removed.

RUN-TIME COUNT OPTION

Statement count information can be obtained at run time only if one of the compiler options COUNT or FLOW was specified at compile time (see "Compiler Options" earlier in this section.) If FLOW but not COUNT was specified at compile time, COUNT must be specified at execution to obtain count information. If COUNT was specified at compile time, count information will be produced unless NOCOUNT is specified at run time.

Count information can be produced only when a statement number table exists. If COUNT is specified at compile time, a table is automatically produced. If only FLOW is specified at compile time, and COUNT is specified at run time, then to obtain count information, GOSTMT or GONUMBER must also be specified at compile time.

Count output is written on the PLIDUMP file or, if no dump file is provided, on the SPRINT file. The output has the following format:

```
     PROCEDURE name1
         FROM       TO   COUNT
            1       20       1
           21       30      10
            .        .       .
            .        .       .
          200      210       1

     PROCEDURE name2
         FROM       TO   COUNT
            1       10       5
            .        .       .
            .        .       .
```

Three such columns are printed per page.

To draw attention to statements that have not been executed, ranges for which the count is zero are listed separately after the main tables.

The count tables are printed when the program terminates.

Count output is headed by the name of the main procedure and the time and date the output was generated. The user's own identifier can be supplied for the output using the PLIXHD string (see "Using PLIXHD to Identify COUNT and REPORT Output").

Count and flow output can be produced only for the main procedure and inner procedures compiled with it. When control is passed to a separate external PL/I procedure, any COUNT or FLOW options in force are suspended until control is returned to the main procedure. Only the compiler options that applied for compilation of the main procedure have any effect on run-time COUNT and FLOW facilities.

RUN-TIME FLOW OPTION

Flow information can be obtained at run time only if one of the compiler options COUNT or FLOW was specified at compile time (see "Compiler Options" earlier in this section.) If FLOW was not specified at compile time, it must be specified at run time to obtain flow information. If FLOW was specified at compile time, flow information will be produced unless NOFLOW is specified at run time.

The format of the run-time FLOW option is the same as that of the compile-time FLOW option, that is:

```
FLOW[(n,m)]
```

where "n" is the maximum number of entries to be made in the flow output, and "m" is the maximum number of procedures for which entries are to be made. Neither "n" or "m" may exceed 32,767.

If "n" and "m" are not specified at run time, they are set as follows:

- If FLOW was specified or defaulted at compile time, the values of "n" and "m" specified or defaulted at compile time are used at run time.
- If FLOW was specified at compile time without subparameters (n,m), the default values (25,10) are used.
- If NOFLOW was specified or defaulted at compile time, the default values, (25,10), are used.

FLOW output is written on the SPRINT file whenever an on-unit with the SNAP option is executed. It is also included as part of PLIDUMP output, if "T" is included in the dump options string.

The format of each line of flow output is:

```
sn1  TO   sn2  [IN name]
```

where

sn1	is the number of the statement from which the branch was made (the <u>branch</u> <u>out</u> point).
sn2	is the number of the statement to which the branch was made (the <u>branch</u> <u>in</u> point).
name	is the name of the procedure or the type of the on-unit that contains "sn2", if this is different from that containing "sn1".

The branches are listed in the order in which they occur. The last "n" branch-in/branch-out points and the last "m" procedures or on-units are listed. If more than "m" procedures or on-units are entered in the course of "n" branches, changes prior to the last "m" procedures or on-units are indicated by printing "UNKNOWN" for "name".

DEBUGGING PL/I (F) PROGRAMS

INTRODUCTION TO DEBUG MODE FOR PL/I

The Symbolic Debugging System (SDS) is a conversational facility for testing and debugging programs. This facility was originally provided for assembly language programs, but it has now been extended to include PL/I (F) programs. Using SDS, the user may initiate the execution of a program and monitor its performance by displaying or modifying variables at strategic points in the program. This section provides a brief introduction to the debug mode command language for PL/I users. A small sample PL/I program is given to illustrate the use of SDS. The complete description of SDS is given in the section "Debug Mode" in MTS Volume 1.

> Users debugging programs produced by PL/I Optimizing Compiler should refer to the section "Program Checkout", since the compiler does not produce SYM (symbol table) records.

Figure 1 is a sample program to compute the mean and standard deviation of an array of real numbers. The program consists of two procedures: the main procedure MAIN which reads in the data values and prints the final results and the internal procedure CALC which computes the desired quantities.

This program is compiled by the PL/I (F) compiler in *PL1 using the MTS command

```
     $RUN *PL1 SCARDS=MEANPROG SPUNCH=MEAN PAR=TEST
```

The source for the program is read from the file MEANPROG and the compiled object module is written into the file MEAN. The TEST parameter must be specified when use of SDS is expected in order to have the PL/I compiler produce SYM (symbol table) records in the object module. These symbol table records are used by SDS and are necessary to enable the user to debug a program symbolically.

The most common method of invoking SDS for debugging this sample program is with the MTS command

```
     $DEBUG MEAN+*PL1LIB
```

The $DEBUG command is the same as the MTS $RUN command in the manner in which logical I/O units and the parameter field are specified. Here it is assumed that the program uses SCARDS for reading the input data and SPRINT for printing the output results. For the present purpose of debugging this program interactively, all input test data will be entered from the terminal (*SOURCE*) and all output results will printed on the terminal (*SINK*). If the user wishes to assign these units to files, he may specify them on the $DEBUG command, e.g.,

```
$DEBUG MEAN+*PL1LIB SCARDS=INPUTFILE SPRINT=OUTPUTFILE
```

SDS signals its readiness to accept a command by printing the prefix character "+" in column one. This prefix character precedes all SDS messages and diagnostics.

When the program has been successfully loaded, the message

```
+READY
+
```

is printed, at which point SDS is ready to accept its first debug command.

```
     STMT
       1      MAIN:    PROCEDURE OPTIONS (MAIN);
       2               DECLARE DATA (50) FLOAT BINARY (16),
                              (MEAN,STD) FLOAT BINARY (16),
                             N FIXED BINARY (31);
       3               ON ENDFILE (SCARDS) STOP;
       5      DATAIN:  PUT FILE (SPRINT) EDIT
                               ('ENTER NUMBER OF DATA POINTS') (A);
       6               PUT SKIP;
       7               GET FILE (SCARDS) LIST (N);
       8               PUT FILE (SPRINT) EDIT
                               ('ENTER DATA POINTS') (A);
       9               PUT SKIP;
      10               GET FILE (SCARDS) LIST ((DATA(I) DO I=1 TO N));
      11               CALL CALC(MEAN,STD);
      12               PUT FILE (SPRINT) LIST ('MEAN=',MEAN);
      13               PUT SKIP;
      14               PUT FILE (SPRINT) LIST ('STD=',STD);
      15               PUT SKIP;
      16               GO TO DATAIN;

      17      CALC:    PROCEDURE (MEAN,STD);
      18               DECLARE (MEAN,STD) FLOAT BINARY (16),
                                (MEAN2,X,Y) FLOAT BINARY (16);
      19               X = 0.0;
      20               Y = 0.0;
      21               DO I = 1 TO N;
      22                  X = X + DATA(I);
      23                  Y = Y + DATA(I)*2;
      24                  END;
      25               MEAN = X/N;
      26               MEAN2 = Y/N - MEAN**2;
      27               STD = SQRT(MEAN2);
      28               END CALC;

      29               END MAIN;
```

Figure 1. Sample Program

Figure 2 gives the sample output from a sequence of commands used to debug the program. Input from the user is given in lowercase and output from SDS and the program is given in uppercase.

Since most users are incurable optimists when it comes to running a program for the first time, the RUN debug command is given to determine what the program will do on the first try. The comments "ENTER NUMBER OF DATA POINTS" and "ENTER DATA POINTS" are produced by the program, and therefore these two lines in the sample output do not start with the "+" prefix character. The program requires as a response an integer N giving the number of data points to be used in the program. The input points are read into the array DATA.

```
     #debug mean+*pl1lib
     +READY
     +run

      ENTER NUMBER OF DATA POINTS
      2,
      ENTER DATA POINTS
      4.0,4.0,

      IHE200I IHESQL X LT 0 IN SQRT (X) IN STATEMENT 00027
              AT OFFSET +00170 FROM ENTRY POINT CALC

     +USER PROGRAM RETURN
     +READY
     +break #19 #27
     +DONE.
     +run

      ENTER NUMBER OF DATA POINTS
      2,
      ENTER DATA POINTS
      4.0,4.0,
     +*** AT BREAKPOINT #0019 IN SECTION MAIN
     +READY
     +display n data(1) data(2)
     +N  'F'  +2    (4 BYTES)
     +DATA(1)  'E'  4.    (4 BYTES)
     +DATA(2)  'E'  4.    (4 BYTES)
     +continue
     +*** AT BREAKPOINT #0027 IN SECTION MAIN
     +READY
     +display mean mean2
     +MEAN  'E'  4.    (4 BYTES)
     +MEAN2  'E'  -8.    (4 BYTES)
     +modify mean2 e'0.0'
     +MEAN2  'E'   WAS -8.    NOW 0.
     +continue
      MEAN=                   4.0000E+00
      STD=                    0.0000E+00
      ENTER NUMBER OF DATA POINTS
      $endfile

     +USER PROGRAM RETURN
     +READY
     +stop
     #
```

Figure 2. Sample Output

A very simple set of test data is chosen for the first run. The size of the data set is 2 and consists of the points 4.0 and 4.0. This data set, using a simple mental calculation, will yield the results of 4.0 for the mean and 0.0 for the standard deviation. In choosing a test data set, it is wise to choose data which will give an obvious and simple answer so that any errors in the program will be readily apparent.

After the program is run, a PL/I error message appears, indicating that an erroneous call to the SQRT library subroutine was made in the CALC procedure. The PL/I library has intercepted the call to SQRT and produced the message indicating that the value of the variable MEAN2 was negative. SDS intercepted the PL/I library return and returned control to debug mode. Whenever any type of abnormal condition occurs during the execution of the program, such as a program interrupt or attention interrupt, SDS will step in and return control to debug command mode. This will also happen in the event of a call by the user’s program to the system library subroutines SYSTEM, MTS, or ERROR.

At this point, if the user has a serially reusable program, he may rerun it and monitor its performance more closely. For a program to be serially reusable, it must be capable of being rerun several times without being reloaded. All locations which contain constant values which are changed by the program must be initialized by the program <u>during execution</u>. For example, a program containing the statements

```
     DECLARE I FIXED BINARY (31) STATIC INITIAL(3);
     K = I;
       .
       .
       .
     I = 6;
```

would not be reusable, since I would not be reinitialized to a value of 3; but a program containing

```
     I = 3;
     K = I;
       .
       .
       .
     I = 6;
```

would be reusable, since I is set to 3 each time the program is used. In general, serially reusable programs are easier to debug with SDS than are nonserially reusable programs, since they can be rerun several times without being reloaded. If the program were not serially reusable, then the user would have to reload the program again using the $DEBUG command.

As an aid to monitoring the execution of the program, SDS provides the capability of setting breakpoints. When a breakpoint is encountered during execution of the program, execution is stopped, and control is returned to debug mode. The instruction at which the breakpoint is set has <u>not yet been executed</u> when execution is stopped.

The BREAK command may be used to set breakpoints by specifying the statement numbers or statement labels at which execution is to be stopped. To refer to a statement number in a PL/I program, the statement number must be prefixed by a "#", e.g.,

```
BREAK #5
```

sets a breakpoint at statement number 5. If the statement has a label, the statement label may also be used, e.g.,

```
BREAK DATAIN
```

sets a breakpoint at the statement labeled DATAIN. Only those statements which define _executable_ PL/I statements may be used to set breakpoints. All others, such as those defining DECLARE, FORMAT, PROCEDURE, and ENTRY statements will be undefined. Statement numbers must be specified _without_ _leading_ _zeros_.

The breakpoints at the statements 19 and 27 of CALC were chosen so as to allow a closer inspection of the program near the area where the error was indicated. At statement 19, the input data may be examined before any actual calculations are made. At statement 27, the argument to the SQRT call may be examined.

After the breakpoints are set, the program is rerun. When the breakpoint at statement 19 is reached, execution is stopped and the message

```
*** AT BREAKPOINT #0019 IN SECTION MAIN
```

is printed. At this point, the user may enter another debug command.

The DISPLAY command may be used to display variable locations in the program. Scalar variables are displayed by giving the variable name; e.g.,

```
DISPLAY MEAN
```

will display the contents of the variable MEAN converted according to its type and length. In this case, MEAN is a float binary variable and its value is printed as

```
MEAN 'E' 4.    (4 BYTES)
```

The code E indicates that the variable is floating-point. Other common codes for PL/I variables are:

E	Float Decimal and Binary Real (floating-point)
P	Fixed Decimal (packed-decimal)
F	Fixed Binary (fixed-point)
C	Character String

Array variables are displayed by giving the array name and its subscripts in the same manner as in the PL/I program; e.g.,

```
     DISPLAY DATA(1)
```

will display the contents of the first element in the array DATA.

After the breakpoint at statement 19 has been reached, the next step is to display some of the input data values for the program to determine whether or not everything seems to be in reasonable order. The values of 2 for N and 4.0 for DATA(1) and DATA(2) indicate that the input data was correctly entered.

A CONTINUE command may then be given to resume execution of the program. After the breakpoint at statement 27 is reached, the user can again check the progress of the program. By displaying MEAN and MEAN2, the user discovers that the values are 4.0 and -8.0, respectively. A quick arithmetic check using the appropriate formulas

$$MEAN = (DATA(1)+DATA(2))/N$$

and

$$MEAN2 = (DATA(1)^2+DATA(2)^2)/N-MEAN^2$$

yields the values 4.0 and 0.0, respectively. Hence, the value -8.0 is in error.

Looking back over the sample program, the user can see that this error was introduced in statement 23 of CALC. That statement should read

```
     Y = Y + DATA(I)**2
```

Since it is not possible to recompile the program in SDS, the best that can be done at this point is to modify MEAN2 to contain the correct value. The MODIFY command may be used to do this. The first parameter for this command gives the name of the variable to be modified. The second parameter gives the value to be used in the modification; the value must be enclosed in primes, e.g.,

```
     MODIFY MEAN2 E'0.0'
```

The value for MEAN2 is now modified to 0.0, and execution of the program may be resumed to determine if the remainder of the program seems to be correct. This time, the correct values for the test data are printed by the program.

Instead of entering a second set of test data, the user will probably want to recompile the program to correct the error in CALC. To terminate the program, the user enters a $ENDFILE (or equivalent). SDS intercepts the termination of the program and returns control to debug mode. The STOP command may be then used to return control to MTS.

The user may use the RESTORE and CLEAN commands to remove breakpoints from the program that were set by the BREAK command. The RESTORE command will remove a specified breakpoint; e.g.,

```
     RESTORE #27
```

will remove the breakpoint set at statement 27 in CALC. The CLEAN command will remove all breakpoints that are set in the program.

Multidimensioned arrays are specified in the same manner as linear arrays. For example, the third element in the array specified by the PL/I source statement

```
     DECLARE ALPHA(10,10) FLOAT BINARY (16);
```

may be displayed by

```
     DISPLAY ALPHA(3,1)
```

A sequence of elements of an array may be displayed using the block notation format. For example, to display the first ten elements of ALPHA, the user may specify

```
     DISPLAY ALPHA(1,1)...(1,10)
```

Arrays may also be displayed using symbolic subscripts. If, in the PL/I program, the variables I and J have the values 2 and 3, respectively, then

```
     DISPLAY ALPHA(I,J)
```

will display the element ALPHA(2,3).

Most debug commands may be given in an abbreviated format. The minimum abbreviations that may be used are underlined in the list below.

BREAK	MODIFY
CLEAN	RESTORE
CONTINUE	RUN
DISPLAY	STOP

An automatic error-dumping facility similar to that provided by the MTS $ERRORDUMP command is provided for batch users. In the event of an error condition occurring during the execution of the program, a symbolic dump will be given of the program. This dump will include all variable locations in the program. This facility may be activated for the sample program by the command sequence

```
     $SET DEBUG=ON
     $SDS SET ERRORDUMP=ON
     $RUN MEAN
     2,
     4.0,4.0,
     $ENDFILE
```

Note that the MTS $RUN command has been given instead of the $DEBUG command. The error-dump facility may be deactivated by the command

```
     $SET DEBUG=OFF
```

The symbolic dump will give the variable storage for the sample program in a format similar to the following:

```
   DUMP OF SECTION MAIN     VA=6003D0  RF=6003D0  LEN=0004D8  SI#=0080

     RA     SYMBOL   TYPE    VALUE                         HEX VALUE

     AUTOMATIC STORAGE  LEVEL=0001  PROCEDURE=MAIN    BASE=6028A8

   0000A0  MEAN      'E'  4.                                41400000
   0000A8  STD       'E'  0.                                00000000
   0000AC  N         'F'  +2                                00000002
   0000B4  DATA(1)   'E'  4.                                41400000
   0000B8  DATA(2)   'E'  4.                                41400000
   0000BC  DATA(3)   'E'  -.699021601E-76                   81818181
    ...      ...             ...
   000178  DATA(50)  'E'  -.699021601E-76                   81818181
```

The following sections give details for more advanced use of SDS with PL/I programs.

ORGANIZATION OF A PL/I (F) PROGRAM

This section describes the basic organization of a PL/I (F) external procedure. Knowledge of this will aid the user in displaying program data variables and managing the program.

An external procedure has several control sections, the most pertinent of which are described below.

The program control section is the first control section and contains all machine language instructions for the procedure. The name of this section is the first label of the external procedure statement. If the label is longer than seven characters, the first four and last three characters of the label are used to form the section name. This is the control section in which breakpoints and at-points are set by the user.

The static internal control section is the second control section and contains storage for all static internal variables and constants. The section name is that of the program control section, extended on the right with a single letter A and padded on the left with asterisks to eight characters, e.g., for the procedure name PROG, the static internal control section name is ***PROGA.

IHEMAIN is a 4-byte control section which contains the address of the main procedure. IHEMAIN is produced only if there is an external procedure with the option MAIN specified.

IHENTRY is a 12-byte control section which is the entry point to the program. IHENTRY is always produced if there is an external procedure. This section immediately transfers to one of six library routines which initialize the PL/I environment before the start of execution in the main procedure.

Static external variables are control section entries if they are initialized, or common section entries if they are not. All variables which are declared as external by the program are in separate sections, one section allocated for each variable declared.

The program control section is subdivided into units called blocks. Each block is a delimited sequence of statements that constitutes a section of the program. There are two kinds of blocks: procedure blocks and begin blocks.

Blocks within an external procedure are either active or inactive. Each time a block is entered, a dynamic storage area (DSA) is allocated for that block; a block is considered active after its DSA has been allocated and before an exit has been made from the block. The DSA contains the control information and the automatic variable storage for the block. When the block is exited, the DSA is released and the block becomes inactive. At this point the automatic storage for the block is released and variables declared as automatic are no longer available to the program or SDS. The DSAs for all blocks that are active within the procedure are chained together. This chaining of DSAs allows SDS to access all of the program's currently allocated automatic data variables at one time.

The following four SDS modifiers are used for specifying the location and block level of program data variables:

The @P=xxx keyword modifier, where "xxx" is the name of an external procedure, may be used to refer to a variable within a particular external procedure. The scope of "xxx" includes all of the control sections of the procedure, all of the internal procedures and blocks contained in the procedure, and all of the static, automatic, based, and controlled data variables declared within the procedure. External variables which are stored in common sections are not included in the scope of "xxx".

The @B=i keyword modifier, where "i" is a block level number, may be used to refer to a variable within a particular block of an external procedure. Each block within an external procedure has a block level number associated with it; this number is given in the compilation listing under the level column. In order for a reference to a particular block to be valid, the block must be active, i.e., the block must have been entered and a DSA must be currently allocated for it.

The @#nn keyword modifier, where "nn" is the compilation statement number of the statement in which the variable was declared, may be used to refer to any variable that was explicitly declared in a DECLARE statement. This modifier is necessary in those cases where there are multiple occurrences of automatic variables of the same name at the same block level or where there are multiple occurrences of static variables of the same name in the same external procedure.

The @I=i keyword modifier, where "i" is an invocation number, may be used to refer to separate invocations of recursive procedures or controlled data variables. The use of this modifier is discussed in more detail below.

DATA VARIABLE SPECIFICATION

All PL/I (F) data variables exist in either static, automatic, based, or controlled storage. The conventions for specifying these different data types are given below.

Static Variables

Static variables (either external or internal) are always available within the program and may be displayed at any time before, during, or after program execution. The @P modifier may be used to specify a particular procedure for an internal variable; the @C modifier may be used to specify a particular common section for an external variable if the specification would otherwise be ambiguous, e.g.,

```
DISPLAY SDATA@P=FIRST (for internal SDATA)
DISPLAY SDATA@C=SDATA (for external SDATA)
```

Automatic Variables

Automatic variables may be displayed only when the blocks declaring them are active. If the block is inactive, the variables are assumed to be unallocated. When the same automatic variable has been declared within several active blocks, the declaration associated with the most recently entered block is assumed unless overridden by the @B modifier; e.g.,

```
DISPLAY ADATA@B=2
```

displays the value of ADATA associated with the second block level. If ADATA was declared in block level 2 at statement 15, the command

```
DISPLAY ADATA@#15
```

could also be used to display its value.

If a block has been entered recursively, the automatic variables associated with the latest entry will be assumed unless overridden by the @I modifier, e.g.,

```
DISPLAY RDATA@I=1
```

displays the value of RDATA associated with the first invocation of the block in which it was declared.

Based Variables

Based variables may be displayed only when they are active, i.e., after they are allocated by an ALLOCATE statement and before they are released by a FREE statement in the program. If the variable is not currently allocated, a message is printed to that effect. Each allocation of a based variable has a pointer variable associated with it. If no pointer variable is specified, the pointer variable given with the declaration statement is assumed. This may be overridden by specifying another pointer variable using the standard PL/I "->" notation, e.g.,

```
DISPLAY PTR->BDATA
```

displays the value of BDATA which has PTR as its pointer variable. The pointer variable name may be qualified with the @P, @B, @#, and @I modifiers to obtain the desired base address; the based variable name may be qualified with the @P modifier to obtain the desired base attributes.

Controlled Variables

Controlled variables may be displayed only when they are active, i.e., after they are allocated by an ALLOCATE statement and before they are released by a FREE statement in the program. If the variable is not currently allocated, a message is printed to that effect. When the same controlled variable has been allocated several times, the most recent allocation is assumed unless it is overridden by the @I modifier, e.g.,

```
DISPLAY CDATA@I=1
```

displays the value of the first invocation of the controlled storage variable CDATA.

The following data-type codes are used for variables in PL/I programs:

E	Float Decimal and Binary Real (floating-point)
M	Float Decimal and Binary Complex (floating-point)
P	Fixed Decimal (packed decimal)
F	Fixed Binary (fixed-point)
C	Character string
B	Bit string
A	Pointer and Label Data (address)
X	Area Data and File Data (hexadecimal)
I	Instruction

SPECIAL DATA SPECIFICATIONS

The following paragraphs describe special considerations that must be followed for certain data variable classifications.

Arrays

Array variables in a PL/I program must be specified with subscripts. An array element specified without a subscript will generate an error message.

Label Variables

Label variables are normally displayed as A-type address constants. If they are displayed in hexadecimal format, they are displayed as 8-byte elements.

Fixed-Decimal Variables

Fixed-decimal variables are currently displayed in packed-decimal format _with_ _no_ _scaling_ _performed_. A fixed-decimal variable declared as

```
DECLARE FDATA FIXED DECIMAL (7,2) INITIAL(6)
```

is displayed in the format

```
FDATA 'P' +0000600
```

Varying-Length Character Strings

Varying-length character strings are displayed at their current length. This may range from zero to the maximum length declared for the string.

Bit Strings

Bit-string variables are displayed as binary constants. An asterisk is used to indicate the offset of the variable within the first byte, e.g.,

```
     BITDATA 'B' ****1110
```

indicates a four-bit variable beginning at bit position 4 (bit positions are numbered 0 through 7). Varying-length bit strings are displayed at their current length. This may range from zero to the maximum length declared for the string.

Picture Data

Pictured-data variables are displayed as character string data using the internal format of the variable. A pictured variable declared as

```
     DECLARE PICDATA PICTURE '$ZZ9V.99' INITIAL('12.34')
```

is displayed in the format

```
     PICDATA 'C' "$ 12.34"
```

Structures

A structured variable must be specified using its fully qualified name even though a partially qualified name is unique within the program. Currently, the total length of the fully qualified name may not exceed 31 characters; if the name is longer than 31 characters, only the first 31 characters are retained in the symbol table and may be used. For structured array elements, all subscripts must appear at the end of the variable name, e.g.,

```
     DISPLAY X.Y.Z(1,2)
```

must be used to display the variable even though X(1).Y.Z(2) might be valid within the program syntax. Aggregate groups of structured elements may not be displayed; each element must be displayed at its lowest level specification.

Statement Labels

Statement labels may be specified either by using a symbolic statement label name or by using the statement number given in the compilation listing. Only statement labels for executable PL/I statements may be specified. Statements such as DECLARE, FORMAT, PROCEDURE, and ENTRY are not defined. Statement numbers are specified in the form "#nn", where "nn" is the compilation listing statement number, e.g.,

```
     BREAK #27
```

sets a breakpoint at statement number 27 in the program. Leading zeros must be omitted.

Area Variables and Offsets

Based area variables may be displayed using either their pointer variables or offsets within the area. When using an offset, the offset must be added to the address of the area variable to form a pointer, i.e.,

```
(area+offset)->variable
```

For example, consider the following sequence of instructions:

```
DECLARE BAREA AREA(256) BASED(APTR),
        1 BAS BASED(BPTR),
          2 OFF OFFSET(BAREA),
          2 VALUE FIXED DECIMAL(6,2),
        QPTR POINTER;

ALLOCATE BAREA;
ALLOCATE BAS IN (BAREA);
ALLOCATE BAS IN (BAREA) SET(QPTR);
BAS.OFF = QPTR;
BPTR -> VALUE = 25;
QPTR -> VALUE = 50;
```

After execution of these instructions, a structured link list is constructed in which the first element has the value 25 and the second element has the value 50. Either of the following commands may be used to display the first element:

```
DISPLAY BAS.VALUE
DISPLAY BPTR->BAS.VALUE
```

Any of the following commands may be used to display the second element:

```
DISPLAY QPTR->BAS.VALUE
DISPLAY (BAREA+BAS.OFF)->BAS.VALUE
DISPLAY (BAREA+(BPTR->BAS.OFF))->BAS.VALUE
```

File Variables

File variables are displayed in hexadecimal format. The region displayed for the file variable is the declare control block (DCLCB) which specifies the attributes of the file. The name of the file is at location 19 (hex) within the DCLCB.

PROGRAM CHECKOUT

Program checkout is the application of diagnostic and test processes to a program. Adequate attention should be given to program checkout during the development of a program so that:

(1) A program becomes fully operational after the fewest possible test runs, thereby minimizing the time and cost of program development.

(2) A program is proved to have fulfilled all the design objectives before it is released for production work.

(3) A program has complete and clear documentation to enable users to use and maintain the program without assistance from the original programmer.

The data used for the checkout of a program should be selected to test all parts of the program. While the data should be sufficiently comprehensive to provide a thorough test of the program, it is easier and more practical to monitor the behavior of the program if data is kept to a minimum.

COMPILE-TIME CHECKOUT

At compile time, both the preprocessor and the compiler can produce diagnostic messages and listings according to the compiler options selected for a particular compilation. The listings and the associated compiler options are discussed in the section "PL/I Optimizing Compiler." The diagnostic messages produced by the optimizing compiler are identified by a number prefixed "IEL". These diagnostic messages are available in both a long form and a short form. The long messages (obtained by the LMESSAGE compiler option) are designed to be as self-explanatory as possible. The short messages are designed for reproduction at a terminal. The short messages are obtained by specifying the SMESSAGE compiler option. Each message is reproduced in the IBM publication: *OS PL/I Optimizing Compiler Messages*. This publication includes explanatory notes, examples, and the corrective action to be taken.

Always check the compilation listing for occurrences of these messages to determine whether the syntax of the program is correct. Messages of greater severity than warning (that is, error, severe error, and unrecoverable error) should be acted upon if the message does not indicate that the compiler has been able to "fix" the error correctly.

The compiler, in making an assumption about the intended meaning of any erroneous statement in the source program, can introduce a further, perhaps more severe, error which in turn can produce yet another error, and so on. When this occurs, the result is that the compiler produces a number of diagnostic messages which are all caused either directly or indirectly by the original error.

Other useful diagnostic aids produced by the compiler are the attribute table and cross-reference table. The attribute table, specified by the ATTRIBUTES option, is useful for checking that program identifiers, especially those whose attributes are contextually and implicitly declared, have the correct attributes. Undeclared identifiers are indicated in the attribute table with a series of asterisks. The cross-reference table is requested by the XREF option and indicates, for each program variable, the number of each statement that refers to the variable.

To prevent the unnecessary waste of time and resources during the early stages of developing programs, use the NOOPTIMIZE, NOSYNTAX, and NOCOMPILE options. The NOOPTIMIZE option will suppress optimization unconditionally, and the remaining options will suppress compilation and execution should the appropriate error conditions be detected.

The NOSYNTAX option specified with the severity level "W", "E", or "S" will cause compilation of the output from the PL/I preprocessor, if used, to be suppressed prior to the syntax-checking stage should the preprocessor issue diagnostic messages at or above the severity level specified in the option.

The NOCOMPILE option specified with the severity level "W", "E", or "S" will cause compilation to be suppressed after the syntax-checking stage if syntax checking or preprocessing causes the compiler to issue diagnostic messages at or above the severity level specified in the option.

RUN-TIME CHECKOUT

At run time, errors can occur in a number of different operations associated with running a program. For instance, an error can cause a program to fail. Most errors that can be detected are indicated by a diagnostic message. The diagnostic messages detected at run time are listed in the IBM publication, OS PL/I Optimizing Compiler: Messages, form SC33-0027, and are identified by the prefix "IBM". The messages are always printed on either MTS logical I/O unit SPRINT or SERCOM.

A failure in the execution of a PL/I program could be caused by one of the following:

- Logical errors in source programs.

- Invalid use of PL/I.
- Unforeseen errors.
- Invalid input data.
- Unidentified program failure.
- A compiler or library subroutine failure.

Logical Errors in Source Programs

Logical errors in source programs can often be difficult to detect. Such errors can sometimes cause a compiler or library failure to be suspected. The more common errors are the failure to convert correctly from arithmetic data, incorrect arithmetic operations and string manipulation operations, and failure to match data lists with their format items.

Invalid Use of PL/I

Often a misunderstanding of the language or a failure to provide the correct environment for using PL/I can result in an apparent failure of a PL/I program. For example, the use of uninitialized variables, the use of controlled variables that have not been allocated, reading records into incorrect structures, the misuse of array subscripts, the misuse of pointer variables, conversion errors, incorrect arithmetic operations, and incorrect string manipulation operations can cause this type of failure.

Unforeseen Errors

If an error is detected during execution of a PL/I program in which no on-unit is provided to terminate execution or attempt recovery, the program will be terminated abnormally. However, the status of a program at the point where the error occurred, can be recorded by the use of an ERROR on-unit that contains the statements:

```
ON ERROR BEGIN;
   ON ERROR SYSTEM;
   PUT DATA;
END;
```

The statement ON ERROR SYSTEM; contained in the on-unit ensures that further errors caused by attempting to transmit uninitialized variables do not result in a permanent loop.

Invalid Input Data

A program should contain checks to ensure that any incorrect input data is detected before it can cause the program to fail.

The COPY option of the GET statement should be used to check values obtained by stream-oriented input. The values will be listed on the file named in the COPY option. If no file name is given, SYSPRINT, which defaults to MTS logical I/O unit SPRINT, is assumed.

Unidentified Program Failure

In most circumstances, an unidentified program failure should not occur when using the optimizing compiler. Exceptions to this could include the following:

- When the program is executed in conjunction with non-PL/I modules, such as FORTRAN.
- When the program obtains, by means of record-oriented transmission, incorrect values for use in label, entry, locator, and file variables.

If execution of a program terminates abnormally without an accompanying PL/I run-time diagnostic message, the error that caused the failure may also inhibit the production of a message. In this situation, it is still possible to check the PL/I source program for errors that could result in overwriting areas of the main storage that contain executable instructions, particularly the communications region, which contains the address tables for the run-time error-handling routine. The types of PL/I program that might cause the main storage to be overwritten erroneously are:

- Assignment of a value to a nonexistent array element. For example:

```
DECLARE ARRAY(10);
  .
  .
  .
DO I = 1 TO 100;
  ARRAY (I) = VALUE;

END;
```

  To detect this type of error, enable the SUBSCRIPTRANGE condition. For each attempt to access an element outside the declared range of subscript values, the SUBSCRIPTRANGE condition will be raised. If there is no on-unit for this condition, a diagnostic message will be printed and the ERROR condition raised. This facility, although expensive in run time and storage space, is a valuable program-checkout aid.

- The use of incorrect locator values for a locator (pointer or offset) variable. This type of error is possible if a locator value is obtained by means of record-oriented transmission. Check that locator values created in a program, transmitted to a file or device, and subsequently retrieved for use in another program are valid for use in the second program.

  An error could also be caused by attempting to free a nonbased variable. This could be caused by freeing a based variable when its qualifying pointer value has been changed. For example:

  ```
  DECLARE A STATIC, B BASED (P);
  ALLOCATE B;
  P = ADDR(A);
  FREE B;
  ```

- The use of incorrect values for label, entry, and file variables. Errors similar to those described above for locator values are possible for label, entry, and file values that are transmitted and subsequently retrieved.

- The use of the SUBSTR pseudo-variable to assign a string to a position beyond the maximum length of the target string. For example:

  ```
  DECLARE X CHARACTER(3);
  I = 3;
  SUBSTR(X,2,I) = 'ABC';
  ```

  The STRINGRANGE condition can be used to detect this type of error.

Compiler or Library Subroutine Failure

If you are absolutely convinced that the failure is caused by a compiler failure or a library subroutine failure, you should notify the Computing Center staff, who will initiate the appropriate action to correct the error. Meanwhile, you can attempt to find an alternative way to perform the operation that is causing the trouble. A bypass is often feasible, since the PL/I language frequently provides an alternative method of performing a given operation.

STATEMENT NUMBERS AND TRACING

The compiler FLOW option provides a valuable program-checkout aid. The FLOW(n,m) option creates a table of the numbers of the last "n" branch-out and branch-in statements and the last "m" procedures and on-units to be entered. (A branch-out statement is a statement that transfers control to a statement other than the one that immediately follows it, such as a GOTO statement. A branch-in statement is a statement that receives control from a statement other than the one that immediately precedes it, such as a PROCEDURE, ENTRY, or any other labeled statement.) The figure chosen for "n" should be large enough to provide a usable trace of the flow of control through the program. Alternatively, if "n" and "m" are not explicitly specified, defaults for the FLOW option will be used.

The trace table can be obtained by any of the methods described below.

The trace is printed whenever an on-unit with the SNAP option is encountered. The trace gives both the statement numbers and the names of the containing procedures or on-units. For example, an ERROR on-unit that results in both the listing of the program variables and the statement number trace can be included in a PL/I program as follows:

```
     ON ERROR SNAP BEGIN;
     ON ERROR SYSTEM;
     PUT DATA;
     END;
```

A flow trace can be specified as part of the output from the PL/I dump facility PLIDUMP, discussed later in this section.

DYNAMIC CHECKING FACILITIES

It is possible for a syntactically-correct program to produce incorrect results without raising any PL/I error conditions. This can be attributed to the use of incorrect logic in the PL/I source program or to invalid input data. Detection of such errors from the resultant output (if any) can be a difficult task. It is sometimes helpful to have a record of each of the values assigned to a variable, particularly label, entry, loop control, and array subscript variables. The CHECK prefix option can be used to obtain this information. Note that, unless care is exercised, the indiscriminate use of the facilities described below will result in a flood of unwanted or unusable information.

A CHECK prefix option can specify program variables in a list. Whenever a variable that has been included in a checklist is assigned a new value, the CHECK condition is raised. The standard system action for the CHECK condition is to print the name and new value of the variable that caused the CHECK condition to be raised. An example of a CHECK prefix options list is:

```
     (CHECK(A,B,C,L)):  /* CHECKOUT PREFIX LIST */
     TEST:  PROCEDURE OPTIONS(MAIN);
     DECLARE A, etc.,
     .
     .
     .
```

If the CHECK condition is to be raised for all the variables used in a program, the CHECK prefix option can be more simply specified without a list of items. This is only possible using the PL/I Optimizing compiler. For example,

```
(CHECK):  TEST:  PROCEDURE;
```

CONTROL OF EXCEPTIONAL CONDITIONS

During execution of a PL/I object program, a number of exceptional conditions can be raised, either as a result of program-defined action, or as a result of exceeding a hardware limitation. PL/I contains facilities for detecting such conditions. These facilities can be used to determine the circumstances of an unexpected interrupt, perform a recovery operation, and permit the program to continue to run. Alternatively, the facilities can be used to detect conditions raised during normal processing and to initiate program-defined actions for the condition. Note that some of the PL/I conditions are enabled by default, some cannot be disabled, and others have to be enabled explicitly in the program. Refer to the IBM language reference manual for this compiler, OS PL/I Optimizing and Checkout Compilers, for a full description of each condition.

Note that the SIGNAL statement can be used to raise any of the PL/I conditions. Such use permits any on-units in the program to be tested during debugging.

The standard system action for the ERROR condition, for which there is no on-unit, is to raise the FINISH condition. The FINISH condition is also raised for the following:

- When a SIGNAL FINISH statement is executed.
- When a PL/I program completes execution normally.
- On completion of an ERROR on-unit that does not return control to the PL/I program by means of a GOTO statement.
- When an EXIT or STOP statement is executed.

The standard system action for the FINISH condition is to terminate the program.

Use of the PL/I Preprocessor in Program Checkout

During program checkout, it is often necessary to use a number of the PL/I conditions (and the on-units associated with them) and subsequently to remove them from the program when it is found to be satisfactory. The PL/I preprocessor can be used to include program-checkout statements from the source statement library. When the program is fully operational, the %INCLUDE statement can be removed, and the resultant object program compiled for execution.

PL/I program checkout statements would include both the enabling of any conditions that are disabled by default and the provision of the appropriate on-units. An %INCLUDE statement that causes the inclusion of the program checkout statements should be placed after any permanent on-units in the program in order to cancel their effect during program checkout.

ON-CODES

On-codes can indicate more precisely what type of error has occurred in those cases in which a condition can be raised by more than one error. For example, the ERROR condition can be raised by a number of different errors, each of which is identified by an on-code. The on-code can be obtained by using the condition built-in function ONCODE in the on-unit. The on-codes are described in the IBM language reference manual for this compiler, OS PL/I Optimizing and Checkout Compilers.

DUMPS

Should the checks given above fail to reveal the cause of the error, it may be necessary to obtain a printout, or dump, of all or part of the storage used by the program. The PL/I Optimizing Compiler produces a run-time dump only by calling PLIDUMP. PL/I (F) users should call IHEDUMP or IHEDUMC (see the subroutine description of IHEDUMP).

Refer to the IBM execution-logic manual, OS PL/I Optimizing Compiler: Execution Logic, form SC33-0025, for information about the organization of the object programs produced by the optimizing compiler, and how to interpret the PLIDUMP outputs.

In batch, if neither PLIDUMP or PL1DUMP was specified, the dump output will be on *SINK* by default. If in conversational mode, the user will be prompted to specify a file or device (such as *PRINT*) for the dump output.

The page size of the PLIDUMP output is taken from the PAGESIZE field of PLITABS.

To obtain a formatted PL/I dump, PLIDUMP must be called. PLIDUMP can be invoked with two optional arguments. The format of the CALL PLIDUMP statement is:

```
CALL PLIDUMP[(options-list[,user-identification])];
```

The first argument, options-list, is a character-string expression that specifies the type of information to be included in the dump. The options-list may include the following:

T	To request a trace of active procedures, begin blocks, on-units, and library modules.
NT	To suppress the output produced by T above.
F	To request a complete set of attributes for all files that are open, and the contents of the buffers used by the files.
NF	To suppress the output produced by F above.
S	To request the termination of the program after the completion of the dump. Note: the FINISH condition is not raised.
C	To request continuation of execution after completion of the dump.
H	To request a hexadecimal dump of the storage used by the program.
NH	To suppress the hexadecimal dump.
B	If T is specified, to produce a separate hexadecimal dump of control blocks such as the TCA and the DSA chain that are used in the trace analysis. If F is specified, to produce a separate hexadecimal dump of control blocks used in the file analysis, such as the FCB.
NB	To suppress hexadecimal dump of control blocks.

The defaults assumed for the above options not specified explicitly are:

```
T  F  C  NH  NB
```

The second argument, user-identification, specifies the identification to be printed at the head of the dump. It can be a character-string expression of up to 90 characters or a decimal constant.

Example

An example of the CALL PLIDUMP statement is:

```
CALL PLIDUMP('TFCNH', 'DUMP AFTER READ');
```

Trace Information

Trace information produced by PLIDUMP includes a trace through all the active DSAs. (DSAs will be present for compiled blocks, such as procedures and on-units, and for library routines.) For on-units, the dump gives the values of any condition built-in functions that could be used in the on-unit, regardless of whether the on-unit actually used the condition built-in function. If a hexadecimal dump is also requested, the trace information will also include:

- The address of each DSA (Dynamic Storage Area).
- The address of the TCA (Task Communications Area).
- The contents of the registers on entry to the PL/I error-handler module.
- The PSW or the address from which the PL/I error-handler module was invoked.
- The addresses of the library module DSAs back to the most recently used compiled code DSA.

DSAs and the TCA are described in the IBM execution logic manual for this compiler. A table of statement numbers indicating the flow of control through the program is produced if the FLOW option is in effect.

File Information

File information produced by PLIDUMP includes the attributes of all open files, and the contents of all buffers that are accessible to the dump routine. The information is given in EBCDIC notation, and, if hexadecimal output is also requested, in hexadecimal notation also. The address and contents of the FCB are then printed. For varying-length records, the RECSIZE is the length of the last processed record.

Hexadecimal Dump

To use a hexadecimal storage dump, the user should know assembler language programming and understand object program organization. The hexadecimal dump is a dump of the region of storage containing the program. The dump is given as three columns of printed output. The left-hand and middle columns contain the contents of storage in hexadecimal notation. The third column contains an EBCDIC translation of the first two columns. For hexadecimal characters that cannot be represented by a printable EBCDIC character, a period is printed.

RUN-TIME RETURN CODES

It is possible to pass a return code from a PL/I program to the program that invoked the PL/I program. For example, if the PL/I program is invoked by the operating system, a return code can be displayed on the "Execution terminated" message or passed as RUNRC for the $IF command.

The return code generated by a PL/I program consists of two elements. One element is specified if the program calls PLIRETC or is set to zero by default. The other element is specified by the program management routines of the PL/I library and indicates the manner in which the program terminated. Unless an error is detected which prevents the PL/I program management routines from operating correctly, the two elements are added together to form a global return code. The thousands digit indicates the manner in which the program terminated; the hundreds, tens, and units digits are set by the program when PLIRETC is called and can be used to allow conditional execution of the next program.

When a PL/I program calls PLIRETC, the argument (return code value) can be either a constant or a variable with the attributes FIXED BINARY(31,0). If a return code greater than 999 is specified, the return code is set to 999 and a diagnostic message is issued.

The meaning of the thousands digit generated by the PL/I program management routines is as follows:

0000	Normal termination.
1000	STOP or EXIT statement, or a call to PLIDUMP with the S option, or insufficient storage in the ISA.
2000	ERROR condition raised and program terminated without return from ERROR or FINISH on-unit.
4000	Error prevented program management routines from functioning correctly. In this situation the remaining digits are used to further identify the error as shown below, and any set by a call to PLIRETC are ignored.
4004	Code returned if the PRV (pseudo-register vector) is too large.
4008	Code returned if PL/I program has no main procedure.
4012	Not enough main storage available.

PL/I INPUT/OUTPUT IN MTS

The following section provides an overview of PL/I input/output and describes how PL/I I/O as implemented in MTS differs from the IBM implementation as described in either the IBM System/360 Operating System PL/I (F) Language Reference Manual, form GC28-8201 or OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, form GC33-0009. These IBM manuals remain the source for information on the many details of PL/I I/O.

There are two basic types of input/output in PL/I: stream I/O, accomplished using the GET and PUT statements, and record I/O, accomplished using the READ, WRITE, REWRITE, LOCATE, and DELETE statements.

With stream I/O, the boundaries between individual records are ignored. (Exceptions are made when during the execution of a PL/I Optimizer program stream files are assigned to a terminal.) Lines entered at a terminal or read from a file or card deck appear to the PL/I program as a continuous stream of characters except when the SKIP option is used on input, e.g., "GET SKIP(2);" skips two lines. Data PUT to a file or device is buffered by PL/I until the current buffer is full or until a SKIP, PAGE, or LINE option is encountered; at this point, the current buffer is written and the buffering process is started again. With record I/O, record boundaries are not ignored. Each READ statement causes a single record to be made available for processing. Each WRITE or REWRITE statement causes one record to be written. Each LOCATE statement points to the next record to be written in the buffer. Each DELETE statement causes a record to be deleted.

Stream I/O does whatever data conversions are necessary to map between the external character form and the internal representation of the data. Record I/O never causes any data conversions. If data conversions are necessary, they must be accomplished within the program after the data is read or written. Record I/O is often used where the data is presented in its internal form and therefore no data conversions are necessary.

Stream I/O is always performed sequentially; record I/O may be performed sequentially or nonsequentially using KEYED files.

STREAM I/O

Stream I/O may be divided into three basic types: list-directed I/O which is similar to the format-free I/O provided by the FORTRAN I/O Library and WATFIV; data-directed I/O which is similar to FORTRAN

NAMELIST I/O; and edit-directed I/O which is similar to FORTRAN formatted I/O.

Stream I/O on Terminals

Stream I/O on a terminal is simplified for a user especially when running a PL/I Optimizer program. The confusion due to a concept of continuous stream of characters is eliminated. Stream files on a terminal are synchronized. For example, a "PUT SKIP;" is no longer required before a GET statement, since the terminal stream-input transmitter will automatically check for any outstanding terminal output stream, which is printed immediately on the terminal.

Terminal Input

When using a GET statement on a terminal, the user will be prompted for the input by a colon prefix (:). The data should then be entered. If data does not complete the GET statement, the user will be further prompted with a two-character prefix (+:).

An input line can also be continued by putting in a hyphen (-) as the last character of the line. This is known as a continuation character. A few more lines can then be entered.

If the GET statement specifies the COPY option and both input file and COPY file are on a terminal, then no copy of data is printed. Note that the default COPY file is SYSPRINT, which is equated to MTS logical I/O unit SPRINT and is assigned to *SINK* by default.

Terminal Output

All stream terminal output will have a PRINT attribute applied. Data on a terminal is not, however, formatted into pages. There is no easy way to tell how many lines will fit a terminal page. For this reason, ENDPAGE is never raised. The PAGELENGTH option, which specifies the length of a page on a terminal, is currently ignored.

Although some terminals have a tabulating facility, tabulating during list-directed and data-directed output is always achieved by transmission of blank characters according to PLITABS (see the section "Tab Control Table").

List-Directed I/O

With list-directed I/O, the data items to be read or written are specified in a data list. It is the data type of the items in this list that determines the conversions that will be necessary to map between the internal and external data representations, i.e., if the internal data type is FIXED BINARY, the external form should be numeric. List-directed I/O is free-format with individual data elements separated by blanks or commas. Strings must be enclosed in primes (’) and character strings that contain primes must use two primes to represent one prime. During input, when the end of the current line is reached, the next line is automatically read. Data elements may be split over more than one line providing that no blanks or commas intervene between data elements. A data element in the list is unchanged if two consecutive commas appear in the input stream in the corresponding place. The number of data items that will appear on an output line is determined by the LINESIZE associated with the PL/I file being used and the program tab settings, if it is a print file (see the section PLITABS below). For example, the following program segment uses list-directed stream I/O to read values from SYSIN and to echo the same values on SYSPRINT:

```
     DECLARE (I,J) FIXED BINARY(15);
     DECLARE TITLE CHARACTER(30) VARYING;
     ON ENDFILE(SYSIN) STOP;
     DO WHILE(’1’B);  /*  FOREVER  */
        GET LIST(TITLE,I,J);
        PUT LIST(TITLE,I,J);
     END;
```

A short terminal session using the above program segment could produce the following (user input is in lowercase, program output is in uppercase):

```
     :’example #1’,10,100
      EXAMPLE #1               10                 100
     :’example #2’
     +:,
     +:200
      EXAMPLE #2               10                 200
     :’example #3’,
     3
     0
      EXAMPLE #3                3                   0
```

For a GET LIST operation from a terminal during the execution of a PL/I Optimizer program, a comma is automatically inserted at the end of a line unless an item is continued by the continuation character (-) as the last character of the line. For this reason, there is no need to enter intervening blanks or commas unless a PL/I (F) program is being run.

Data-Directed I/O

With data-directed I/O as with list-directed I/O, the data items to be read or written are specified in a data list. The conversions that are required to map between the internal and external data representations are determined by the data types of the variables in the data list. Data-directed I/O is free-format with the individual values to be read or written appearing in a form very similar to a series of assignment statements; that is, with the name of the variable to the left of an equal sign and the value of the variable to the right. Individual assignments are separated by blanks or commas, and strings are enclosed in single quotes. Since each assignment contains the name of the variable involved, the order of data in the input stream need not match the order in the data list. A data-directed GET statement continues to process assignments until terminated by a semicolon; the values of any variables included in the data list but omitted from the input stream remain unchanged. Consider the following program segment:

```
     DECLARE (I,J) FIXED BINARY(15);
     DECLARE TITLE CHARACTER(30) VARYING;
     ON ENDFILE(SYSIN) STOP;
     DO WHILE('1'B);  /*  FOREVER  */
        GET DATA(TITLE,I,J);
        PUT DATA(TITLE,I,J);
     END;
```

A short terminal session using the above program segment could produce the following (user input is in lowercase, program output is in uppercase):

```
     :title='example #1', i=10  j=-8 ;
      TITLE='EXAMPLE #1'      I=       10             J=       -8;
     :title='example #2'  i='10'
     +:j=001 ;
      TITLE='EXAMPLE #2'      I=       10             J=        1;
     :title=3, j=4 ;
      TITLE='   3'            I=       10             J=        4;
```

For a GET DATA operation from a terminal during the execution of a PL/I Optimizer program, there is no need to enter intervening blanks or commas. PL/I Optimizer routines will automatically insert a comma at the end of a line unless the line is continued by the continuation character (-) as the last character.

An abbreviated form of data-directed I/O that uses no data list is allowed and is treated as if a data list that included all variables within the current scope of the program were included in the data list. This form of data-directed I/O is often handy during debugging, but can result is a large amount of output in a program with large structures or arrays, e.g.,

```
     PUT DATA;
```

Edit-Directed I/O

With edit-directed I/O, the operations to be performed are specified by a data list and an associated format list. The conversions necessary to convert between internal and external data representations are determined by the data types of the items in the data list and by the format items in the format list. Edit-directed I/O is not free-format, no explicit delimiters are required between data elements, and strings are not enclosed in quotes. An edit-directed I/O statement continues to process data in order until the end of the data list is reached. Consider the following program segment:

```
     DECLARE (I,J) FIXED BINARY(15);
     DECLARE TITLE CHARACTER(30) VARYING;
     ON ENDFILE(SYSIN) STOP;
     DO WHILE('1'B);  /*  FOREVER  */
        GET EDIT(TITLE,I,J) (A(10),F(3),F(3));
        PUT EDIT(TITLE,I,J) (A,2(F(3)));
     END;
```

A short terminal session using the above program segment could produce the following (user input is in lowercase, program output is in uppercase; b̸ represents one blank):

```
     :abcdefghij123456
      ABCDEFGHIJ123456
     :aeiou     1  2b̸b̸
      AEIOU       1  2
     :example #3  3  4
      EXAMPLE #3  3  4
     :example #4
     +:5
     +:-08
     EXAMPLE #4  5 -8
```

For a GET EDIT operation from a terminal during the execution of a PL/I Optimizer program, the last item being entered on a line will be padded to the correct length unless the line has the continuation character (-) as the last character, in which case the item will be continued onto the next input line. For example, the following statement:

```
     GET EDIT (NAME) (A(25));
```

The user can just enter his name:

```
     :John Doe
```

The name "John Doe" is automatically padded to 25 characters with necessary blanks.

While edit-directed I/O and FORTRAN-formatted I/O are very similar, there are some important differences. FORTRAN format items specify both the internal and external form of the data to be converted. PL/I edit-directed I/O uses the data type of the item in the data list to determine the internal form and the type of the items in the format list to determine the external form.

Format Items

There are two types of format items in PL/I: data format items and control format items. Data format items describe data items in the input or output data stream. Control format items specify positioning within the data stream or on the printed page.

The fixed-point format item specifies the appearance of a decimal fixed-point data item. It is given in the form

```
     F(w,d)
```

where "w" represents the width of the field, or the total number of characters, including the sign and decimal point; and "d" represents the number of digits to the right of the decimal point. If "d" is omitted, the decimal point is assumed to be to the right of the rightmost digit. On output, an actual decimal point is inserted, trailing zeros are supplied, if necessary, and a minus sign is inserted if the value of the data item is less than zero. For example, using the format item F(10,3), the number -123.4567 would be specified as

```
     -12345.670
```

The F format item is also used to write fixed-point binary data, which is converted on output to decimal notation.

The floating-point format item specifies the appearance of a decimal floating-point data item. It is given in the form

```
     E(w,d)
```

where "w" represents the width of the field or the total number of characters, including signs, decimal point, and the E exponent flag; and "d" represents the number of digits to the right of the decimal point. On output, a decimal point is inserted and blanks are inserted to the left if the actual number of characters is less than "w". A minus sign is supplied for the exponent if the implied location of the decimal point is to the left of its actual location; a minus sign is inserted to the left of the first character if the value of the data item is less than zero. For example, using the format item E(13,7), the number -123.4567 would be specified as

```
     -.1234567E+03
```

The E format is used on output for either decimal or binary floating-point data. Binary data is always converted to decimal notation.

The character-string format item specifies character strings in the data stream. It is given in the form

```
     A(w)
```

where "w" represents the number of characters in the string. The "w" is always required on input; for output, if "w" is omitted, the length is taken as the actual length of the specified string. Quotation marks should not appear in the input stream because a single quotation mark would be considered as a single character. Quotation marks are not written on output.

The bit-string format item specifies bit strings in the data stream. If is given in the form

```
     B(w)
```

where "w" represents the number of bits in the string. The "w" is always required on input; for output, if "w" is omitted, the length is taken as the actual length of the specified bit string. Neither quotation marks nor the letter B should appear in the input stream. They are not written on output.

The spacing-control format item specifies the relative horizontal spacing in a line. It is given in the form

```
     X(n)
```

On input, it specifies the number of characters "n" to be ignored. On output, it specifies that "n" blanks are to be inserted into the data stream.

The printing-control format items specify how output is to appear on the printed page. They are PAGE, SKIP(n), LINE(n), and COLUMN(n).

The PAGE format item specifies that the next output line is to be written on a new page.

The SKIP(n) format item specifies that n-1 lines are to be skipped and the next data item is to be written on the <u>n</u>th line. If "n" is omitted, it indicates that the next data item is to be written on the next line. The SKIP format item is often used in the PUT SKIP statement to force the current output buffer to be written.

The LINE(n) format item specifies that lines are to be skipped so that the next data item will be written on the <u>n</u>th line of the current page.

The COLUMN(n) format item specifies that blanks are to be inserted so that the first character of the next data item will be the _n_th character of the current line.

Note that the SKIP format item, like the X format item, specifies relative spacing, while LINE and COLUMN specify absolute spacing.

The above paragraphs provide a brief introduction to PL/I stream I/O; however, only the barest details are covered. For further details on the many specialized format items available in PL/I, and details on stream I/O using arrays, structures, etc., the reader should refer to the sections "Stream-Oriented Transmission" and "Edit-Directed Format Items" in one of the IBM PL/I Language Reference Manuals or one of the texts included in the bibliography at the end of this volume.

Use of PRINT Files

For PL/I (F) programs, if the PRINT files are attached to a printer or to HASP, the first character of a line is immediately translated to an equivalent machine carriage control character so that MTS will not automatically issue a page skip. If the PRINT files are attached to files or devices other than printers, the first characters of lines remain unchanged. If these are then copied to a printer, a program *ASA should be run to override the MTS automatic page skip, especially when a page has more than 60 lines.

For PL/I Optimizer programs, any line with a line count greater than 60 will now contain several lines with the carriage control "9". Hence, if PRINT files are attached to files or devices, the output can safely be copied to a printer.

Tab Control Table

Data-directed and list-directed output to a PRINT file is automatically aligned on preset tabulator positions; the tab settings are stored in a table, an assembler language control section, IBMBSTAB (see Figure 1).

```
IBMBSTA1 CSECT
         ENTRY IBMBSTAB
IBMBSTAB EQU   *
         DC    C'IBMBSTAB'
         DC    H'14'          OFFSET OF TAB COUNT
         DC    H'60'          PAGESIZE
         DC    H'120'         LINESIZE
         DC    H'0'           PAGELENGTH FOR TERMINALS
         DC    3H'0'          FILLERS (RESERVED)
         DC    H'5'           TAB COUNT
         DC    H'25'          TAB 1
         DC    H'49'          TAB 2
         DC    H'73'          TAB 3
         DC    H'97'          TAB 4
         DC    H'121'         TAB 5
         END
```

Figure 1. Tab control library module IBMBSTAB.

The standard settings are given in the IBM language reference manual for this compiler. The functions of the fields in the table are as follows:

OFFSET OF TAB COUNT:	Halfword binary integer that gives offset of the field, indicating the number of tabs used.
PAGESIZE:	Halfword binary integer that defines the default page size.
LINESIZE:	Halfword binary integer that defines the default line size.
PAGELENGTH:	Halfword binary integer that defines the default page length for printing at a terminal. The page length is the number of lines between perforations. The default value is zero, a special convention to indicate unformatted output. PAGELENGTH is currently ignored.
FILLERS:	Reserved for future use.
Tab count:	Number of tab position entries in a table (maximum 255). If tab count = 0, any specified tab positions are ignored; each data item is positioned at the start of a new line.
Tab 1-Tab n:	Tab positions within the print line. The first position is numbered 1, and the highest position is

numbered 255. The value of each tab should be greater than that of the tab preceding it in the table; otherwise, it will be ignored. The first data field in the printed output begins at the next available tab position.

Note that the first item on the line is always printed in column 1. The first tab sets the position of the second item.

The standard PL/I tab settings in IBMBSTAB can be overridden. If PLITABS is present, the module IBMBSTAB will not be used. Instead, the stream-oriented input/output routines will refer to the control section PLITABS for the tab settings.

There are two methods of altering the tab settings for a particular program. One method is to create an assembler-language control section called PLITABS and include it with the program. The alternative method is to include a PL/I structure in the source program. The organization of the structure is similar to the assembler-language control section for PLITABS given in Figure 1. The name of the structure must be PLITABS and must be declared STATIC EXTERNAL. An example of a PL/I structure to create three tab settings in positions 30, 60, and 90, and use the defaults for page size and line size is given in Figure 2.

```
DECLARE 1 PLITABS STATIC EXTERNAL,
        2 (OFFSET INITIAL(6),
          PAGESIZE INITIAL(60),
          LINESIZE INITIAL(120),
          NO_OF_TABS INITIAL(3),
          TAB1 INITIAL(30),
          TAB2 INITIAL(60),
          TAB3 INITIAL(90)) FIXED BINARY(15,0);
```

Figure 2. PL/I structure PLITABS for modifying the standard tab settings (alternative method).

The equivalent fields for PAGELENGTH and FILLERS are omitted from the structure, and the value given in the offset field is set to 6.

Note that the PAGESIZE field in PLITABS is used by PLIDUMP to define the page size for the dump output.

RECORD I/O

Record I/O may be used to access MTS files and devices sequentially or randomly. Sequential access may be performed on any valid MTS file or device, but random access only may be performed using MTS line or sequential files. Random access is performed using KEYED files; the MTS implementation of this type of record I/O differs considerably from the standard IBM implementation.

All PL/I stream I/O statements can be used in MTS just as they are described in the IBM PL/I (F) Language Reference Manual. However, there are some restrictions on record I/O statements. A list of the record I/O statements and options which can be used in MTS is given below.

```
                          ┌ INTO (variable)   ┌ KEY (expression) ┐  ┐
   READ FILE(filename)    | SET (pointer)     └ KEYTO (variable) ┘  |;
                          └ IGNORE (expression)                     ┘

   WRITE FILE(filename)  FROM (variable) [ KEYFROM (expression) ];

   LOCATE based_variable FILE (filename) [ SET (pointer) ]
                         [ KEYFROM (expression) ];

   REWRITE FILE (filename) [ FROM (variable) ] [ KEY (expression) ];

   DELETE FILE (filename) [ KEY (expression) ];
```

Note: The PL/I Optimizing compiler does not currently support the DELETE statement and KEY, KEYTO, KEYFROM options.

Use of BACKWARDS Files

BACKWARDS files are implemented only by the PL/I Optimizing compiler. This applies not only to magnetic tapes but also on *DUMMY* and MTS line or sequential files, which can be read backwards. Only READ statements can be used with these files, starting with the last record and ending with the first record. If a file FYLE(1,3) is attached to a backwards file, this file will be read from MTS line 3 to MTS line 1. Any use of BACKWARDS files on other than *DUMMY*, MTS files, and magnetic tapes, will raise an UNDEFINEDFILE condition with ONCODE=82.

Use of KEYED Files

To access a file randomly, KEYED files must be used. This subsection modifies the information on KEYED files given in the *IBM PL/I (F) Language Reference Manual* and is intended to be used in conjunction with the Reference Manual.

The KEYED attribute must be specified for a PL/I file whenever the user desires to use KEY, KEYTO, or KEYFROM options in record I/O statements. Only formats U and F are supported, and records must be unblocked. In MTS, two types of organization in the ENVIRONMENT attribute are recognized for KEYED files: INDEXED, which applies to line files, and CONSECUTIVE which applies to sequential files. REGIONAL organization is not supported in MTS.

There are three types of keyed PL/I files:

(1) FILE KEYED ENVIRONMENT (CONSECUTIVE)
(2) FILE KEYED ENVIRONMENT (INDEXED)
(3) FILE KEYED ENVIRONMENT (INDEXED GENKEY)

Refer to the previous section for input/output statements which can be used with KEYED files. Some are shown in the examples which follow.

Consecutive KEYED Files

For consecutive KEYED files, the keys are four-character strings internally representing record pointers for the corresponding records in a sequential file. A special key, having the value of binary zero, points to the beginning of the file. Since it is difficult to determine keys of records due to the structure of sequential files, two subroutines NEXTKEY and LASTKEY are available to provide keys of the next record and one past the last record of the file, respectively.

The PL/I (F) compiler often issues the message that the attributes KEYED and CONSECUTIVE are conflicting. They conflict in OS, but not in MTS. This message may be ignored. For example:

```
MAIN: PROCEDURE OPTIONS(MAIN);
      DECLARE (LASTKEY,NEXTKEY) ENTRY(FILE) RETURNS (CHAR(4)),
              (LAST, NEXT) CHARACTER(4), BUFF CHAR(32767) VARYING,
      ZOT FILE KEYED UPDATE ENVIRONMENT(CONSECUTIVE);
      OPEN FILE (20T) KEYED UPDATE;
      NEXT = LOW(4); /* To set the file at the beginning */
      LAST = LASTKEY(ZOT); /* To determine key after last record */
A:    READ FILE (ZOT) KEY (NEXT) INTO (BUFF); /* Read into buffer */
      /* If the record starts with a "1", change tenth and eleventh
         characters to "1" and "0", and rewrite the record */
      IF SUBSTR(BUFF,1,1) = '1' THEN DO;
         SUBSTR(BUFF,10,2) = '10';
         REWRITE FILE (ZOT) FROM (BUFF);
         END;
      NEXT = NEXTKEY (ZOT); /* Set key NEXT to the key of next record */
      IF NEXT ¬= LAST /* Are we done with last record? */
         THEN GO TO A;    /* No. */
      END;
```

Indexed KEYED Files without GENKEY Option

A key without the GENKEY option is a four-character string corresponding to a FIXED BINARY(31) internal form of the MTS line number, i.e., the line number times 1000. This type of PL/I file can be used only with MTS line files. For example:

```
         DECLARE L# FIXED BINARY (31),
                 KEY# CHARACTER(4) DEFINED L#,
                 ZE FILE KEYED DIRECT ENVIRONMENT (INDEXED),
                 BUFF CHARACTER (255) VARYING;
         L# = 1000;  /* This refers to MTS line number 1.000 */
         DELETE FILE (ZE) KEY (KEY#); /* Deletes the line */
         L# = 14000; /* Now refers to MTS line number 14.000 */
         READ FILE (ZE) INTO (BUFF) KEY (KEY#);
         PUT DATA (BUFF); /* To see that correct line was obtained */
```

The variables used with the PL/I KEY, KEYTO, and KEYFROM options must always be character-string variables, while the actual keys used to access MTS line files without the GENKEY option are fullword integers. Because of this conflict, it is necessary to associate two PL/I variables using the DEFINED attribute. These are a CHARACTER(4) variable and a FIXED BINARY(31) variable. In the above program, defining L# and KEY# to occupy the same storage location will produce a PL/I warning message that may be ignored.

Indexed KEYED Files with GENKEY Option

A key with the GENKEY option is a character string of either fixed or varying length that represents the external form of an MTS line number. This key must conform to the format:

±ddddd.ddd

where:

±	The sign, either "-" or "+", is optional but must appear if a line number is to be negative.
d	There must be at least one digit, no more than five digits to the left of the decimal point, and no more than three digits to the right. The decimal point is generally optional but must appear if any of three digits to the right of the decimal point is nonzero.
blank	There may be any number of blanks on the left or the right. No intervening blanks are allowed.

All character strings not conforming to the above rules (such as 'A' or '1.2.3') will raise the KEY conversion error. This type of PL/I file may be used only with MTS line files.

The KEYTO option may be used to return key values, typically when a KEYED indexed file is being read sequentially. When the KEYTO option is used to read indexed files with the GENKEY option, the line number returned will be formatted according to the following rules:

(1) If the KEYTO option is too short, the line number is truncated on the right without raising KEY error. For example, a key string of length 2 for the line number '-123' appears as '-1'.

(2) If the KEYTO string is of varying length, leading and trailing zeros are eliminated and the form is compressed, allowing no blanks at the left or right. For example, a key string for the line number '-001.230' appears as '-1.23'.

(3) If the KEYTO string is of fixed length less than 10 characters, the number is first compressed and then padded with any necessary blanks on the right. For example, a key string of fixed length 8 for the line number -001.230 appears as '   -1.23'.

(4) If the KEYTO string is fixed and is of length 10 or more, only leading zeros on the left of the decimal point are eliminated. The number is then right-justified with blanks. For example, a key string for the same number appears as '   -1.230'.

In the following example, an MTS line file is read beginning from line 20, and then sequentially up to and including line 30.

```
MAIN:   PROCEDURE OPTIONS(MAIN);
        DECLARE GKF FILE KEYED ENVIRONMENT(INDEXED GENKEY),
                KT CHARACTER(10) VARYING,
                BUFF CHARACTER(255) VARYING;
        READ FILE (GKF) INTO (BUFF) KEY ('20');
LOOP:   READ FILE (GKF) INTO (BUFF) KEYTO (KT);
        PUT DATA (BUFF, KT);
        IF KT < '30' THEN GO TO LOOP;
        END MAIN;
```

PL/I FILES, MTS FILES AND DEVICES, AND LOGICAL I/O UNITS

All PL/I input/output statements use PL/I files either explicitly or implicitly. These PL/I files are in turn associated with an MTS file, or more properly, and MTS file or device (FDname). With this rather indirect method of associating an I/O statement with the MTS file or device on which the actual I/O is to be done, it is possible to avoid building the MTS FDnames into the PL/I source program. Another advantage is that PL/I I/O statements may be written in a somewhat device-independent manner. This is only partially true, however. For example, KEYED I/O may only be performed using MTS line or sequential files. A disadvantage to this method of associating I/O statements with real files and devices is that documentation that refers to a "file" may not specify whether a PL/I or MTS file is meant. In the terminology of the PL/I Language Reference Manual, the term "file" always refers to a PL/I file, while the term "data set" would be known as a file/device name (FDname) in MTS.

There are several paths that may be followed to associate a PL/I file with an MTS FDname. The most direct is to choose a PL/I file name that is the same as one of the MTS logical I/O unit names, i.e., SCARDS, SPRINT, SPUNCH, SERCOM, or GUSER. In this case, the MTS logical I/O unit may be assigned to an MTS FDname on the $RUN command in the standard manner or it may be allowed to default (see MTS Volume 1 for details). The PL/I Optimizing compiler also equates SYSIN and SYSPRINT to MTS logical I/O units SCARDS and SPRINT, respectively.

Another method is almost as direct and does not require the choice of any special names. In this case, the PL/I file name is assigned to an MTS FDname in the PAR field of the $RUN command (see the section "Running a PL/I Program" in this volume). If a PL/I record format modifier is to be used in the assignment of a PL/I file to an MTS FDname, this method must be used even if the PL/I file name corresponds to one of the MTS logical I/O unit names.

Finally, a PL/I file name may be associated with a PL/I title which corresponds to an MTS logical I/O unit number (see below). The MTS logical I/O unit number may then be assigned to an MTS FDname in the standard fashion on the $RUN command. Once again, if a PL/I record format modifier is to be used in the assignment, the assignment must be made in the PAR field of the $RUN command.

The use of the TITLE option in the OPEN statement makes possible references to logical I/O units 0 to 99. For example, using the logical I/O unit 2:

```
     OPEN FILE(NUMBER) OUTPUT PRINT TITLE('2');
```

A corresponding PUT statement and $RUN command would be:

```
     PUT FILE (NUMBER) DATA (A);
      ...
     $RUN -OBJ+*PL1LIB 2=FILEA
```

A single PL/I file variable can refer to more than one logical I/O unit (but only one at a time) by closing and reopening the file with a different TITLE option.

The PL/I-callable subroutine ATTACH (described in the section "PL/I Library Subroutines" in this volume) provides the capability to perform the same type of assignment between PL/I files and MTS FDnames that is allowed in the PAR field of the $RUN command.

Standard Files

PL/I includes two standard files, SYSIN for input and SYSPRINT for output. If the PL/I Optimizer program includes a GET statement that does not include the FILE or STRING option, the compiler inserts FILE(SYSIN); if it includes a PUT statement without the FILE or STRING option, the compiler inserts FILE(SYSPRINT). The PL/I (F) compiler uses SCARDS and SPRINT instead of SYSIN and SYSPRINT as standard files.

If SYSPRINT is not declared, the compiler will give the file the attribute PRINT in addition to the normal default attributes. The complete file declaration will be:

```
     SYSPRINT FILE STREAM OUTPUT PRINT EXTERNAL
```

Since SYSPRINT is a PRINT file, the compiler also supplies a default line size of 120 (or less to fit a terminal). SYSPRINT will be associated with MTS logical I/O unit SPRINT, which defaults to *SINK*.

The attributes given to SYSPRINT by the compiler may be overridden by explicitly declaring or opening the file. The user must bear in mind that this file is also used by the error-handling routines of the compiler, and that any change made in the format of the output from SYSPRINT will also apply to the format of run-time error messages. When an error message is printed, eight blanks are inserted at the start of each line except the first. If a line of size less than 72 characters is specified, the messages will not be output to SYSPRINT.

If SYSPRINT cannot be used for error messages, run-time messages will appear on MTS logical I/O unit SERCOM. For example, if SYSPRINT is a RECORD file, messages will appear on SERCOM, instead.

The compiler does not supply any special attributes for the standard input file SYSIN; if it is not declared, it receives only the normal default attributes. SYSIN is usually associated with MTS logical I/O unit SCARDS.

Record Formats

Record formats are given in the following form:

```
U[A|M][(maximum blocksize)]
V[B][S][A|M][( maximum blocksize[,maximum recordsize])]
F[B][A|M][(maximum blocksize[,recordsize])]
D[B][A][(maximum blocksize[,recordsize])]
```

where:

U	Undefined formatted records have no fixed length and no internal count data in the record. They are probably most useful with respect to MTS line files. Trailing blanks, however, are not trimmed.
V	Variable formatted records are also of variable length but contain count fields as part of the record: one for the block length, and one for each record length. They may be blocked. If the maximum record size is not specified, it is assumed to be 4 less than the maximum blocksize.
F	Fixed formatted records are of fixed length and may be blocked. Due to the trim feature in MTS file handling, PL/I routines automatically pad the records to the right length with blanks as necessary.
D	ASCII variable-length are of different lengths. They contain 4 control bytes describing the length of a record. They also may be blocked (DB).
B	Blocked records for formats F and V (i.e., FB and VB) increase efficiency of I/O operations since the number of blocks is reduced by having every block contain two or more records instead of one. This is particularly useful for mag-

	netic tapes. B should not be specified in the ENVIRONMENT file attribute--it is inferred by the difference between the block size and the record size; the exception is VBS blocked spanned variables.
S	Spanned variable-length records (VS or VBS) permit a record to be spanned over several blocks.
A or M	This option specifies that the records contain ANS standard (A) or machine (M) carriage-control characters respectively as the first character of each data record. These will be used to effect carriage control if written to a printer or terminal. ANS standard carriage control is normally used in MTS. A and M should not be specified in the ENVIRONMENT file attribute--the equivalent are CTLASA and CTL360, respectively, for record I/O files; for stream I/O, A is assumed automatically for print files only.
Maximum Blocksize	is the length or maximum length of a block in the file, including the control fields in V, VB, D, or DB format records. It is an integral multiple of the record size in FB-formatted records.
Maximum Recordsize	is the length or maximum length of the records in the file. It includes the record length of the control word for V-formatted and D-formatted records.

Examples:

U(255) — specifies a file of records with a maximum length of 255. Since there is no count field, the user may like to determine the record length. For example:

```
DECLARE IN CHARACTER(255) VARYING;
READ FILE(INPUT) INTO (IN);
I=LENGTH(IN);
```

Here I contains the actual length of the record.

UA(133) — specifies an undefined format item with a standard carriage-control character and up to 132 positions of data to be printed.

V(88,84) specifies a file with variable-length records. Each record will contain up to 80 bytes of data, a 4-byte record-descriptor word, and a 4-byte block-descriptor word. The record and block descriptors both contain a halfword length followed by a halfword of flags. An 80-byte data file may be specified as V(88), which is same as V(88,84).

F(80) specifies a file with fixed-length records--80 bytes of data in each record. Blanks may be appended for all records shorter than 80 bytes so that their length equals 80.

FB(8000,80) specifies a blocked file with fixed-length records--one-hundred 80-byte records per block.

The following sample program can be executed giving the file OUTPUT various record format specifications.

```
$RUN *PL/1 SPUNCH=-OBJ
 P: PROCEDURE OPTIONS(MAIN);
        DECLARE
           OUTAREA CHAR(255) VARYING,
           FIELD CHAR(255);
        DO I=1 TO 5;
           OUTAREA=SUBSTR(FIELD,1,5*I);
           WRITE FILE(OUTPUT) FROM (OUTAREA);
        END;
    END P;
$ENDFILE
```

Execution with the record format on the following $RUN command:

```
$RUN -OBJ+*PL1LIB PAR=OUTPUT=-OUT@V(255)
```

will produce in the file -OUT five lines:

```
         ┌────┬────┬──────────────┐
 line 1  │ 13 │  9 │  5 data bytes│
         ├────┼────┼──────────────┴─────┐
 line 2  │ 18 │ 14 │ 10 data bytes      │
         ├────┼────┼────────────────────┴─────┐
 line 3  │ 23 │ 19 │ 15 data bytes            │
         ├────┼────┼──────────────────────────┴─────┐
 line 4  │ 28 │ 24 │ 20 data bytes                  │
         ├────┼────┼────────────────────────────────┴─────┐
 line 5  │ 33 │ 29 │ 25 data bytes                        │
         └────┴────┴──────────────────────────────────────┘
```

This file is unblocked. The same file would be produced if the file specification had been V(33,29). The specification of a blocked file for the output of the same program could be given as follows:

```
          $RUN -OBJ+*PL1LIB PAR=OUTPUT=-FILE@VB(1004,100)
```

-FILE will have several variable-length logical records written in each block. Each record can contain up to 96 bytes of data plus the 4-byte recordsize control field. Each block can contain up to 1004 bytes, including the 4-byte blocksize control field.

When the program is run, the file -FILE will contain a single line in the following format:

99	9	5 data bytes	14	10 data bytes	19	15 data bytes	•••

24	20 data bytes	29	25 data bytes

The same file would have been created for this program if the format specification had been VB(1004,29). If the record format specification were U(25 or larger), five separate lines would be written with 5, 10, 15, 20, and 25 data bytes, respectively.

Magnetic Tape I/O

When opening a PL/I file on a magnetic tape, detailed information such as record size, block size, and record format can be obtained from the current data set labels for labeled tapes (the default is U(255) in the case of unlabeled tapes). This is merged with the record format specified in either the PAR field parameter string or by the ATTACH routine. This, in turn, is merged with the environment attribute options of the PL/I file. This becomes the final record format (see examples below). If a new data set is about to be written, the new record format is passed to the MTS magnetic tape routines so that the data set characteristics and the record format do match, thus ensuring the correct record format when the data set is read in. If the tape is on a 7-track drive and is to be V-formatted, the data converter feature is enabled. Finally, because PL/I (F) routines do their own blocking, the normal MTS blocking support is disabled and not reenabled when the file is closed. To reenable the MTS blocking support, the user should issue the following command after the program is terminated:

```
     $CONTROL *T* BLK=ON
```

where *T* represents the pseudo-device name of the tape. PL/I Optimizer routines do not disable the normal MTS blocking support, and hence the command above is not necessary.

There is a slight discrepancy between PL/I routines and MTS magnetic tape routines, namely that V-formatted and D-formatted record size in PL/I must be LRECL (logical record length) plus 4.

At the CLOSE statement, all output tapes have an end-of-file written and then are backspaced to just before the end-of-file. Finally, unless the LEAVE environment option is specified, every tape, whether input or output, is rewound to the start of the first data set of the tape.

Example 1

```
     Unlabeled tape default:      U(255)
     PAR field:                   PAR=TAPE=*T*@V(500,80)
     Environment option:          PL/I (F): DECLARE TAPE FILE
                                          ENV (F(400));
                                  PL/I OPT: DECLARE TAPE FILE
                                          ENV (F BLKSIZE(400));
```

The record format in effect is FB(400,80). Note the order which gives the result.

Example 2

```
     $MOUNT XXX 9TP *XXX* VOL=ABC 'ID'
     $CONTROL *XXX* POSN=ZOO
     $RUN PGM+*PL1LIB PAR=TAPE=*XXX*
```

There is no need to specify the record format since the labels on the tape automatically specify it.

Example 3

```
     $MOUNT YYY 7TP *YYY* VOL=BEE RING=IN 'VALE'
     $CONTROL *YYY* POSN=*EOT*
     $CONTROL *YYY* DSN=SAMPLE
     $RUN ABC+*PL1LIB PAR=TAPE=*YYY*@FB(4000,80)
```

where in the program ABC there appears the statement:

```
     PL/I (F): DECLARE TAPE FILE ENVIRONMENT (V(5000));
```

or

```
     PL/I OPT: DECLARE TAPE FILE ENVIRONMENT (V BLKSIZE(5000));
```

The record format in effect will be VB(5000,80), not FB(4000, 80). Note that because this is a V-formatted tape, the labels generated by MTS will specify VB(5000,76) according to MTS since MTS itself always deals with the true record length (without the control word) whereas the PL/I record length must be specified with the 4 extra bytes for the control word. Also note that because this is a V-formatted 7-track tape, the data converter feature is enabled.

Default Record Size

If a PL/I file record size is not explicitly specified at either compile-time or execution-time, the following defaults are used:

Files with the PRINT attribute:

> The default print line size (121), the maximum output record length, or the terminal width.

Files with the OUTPUT or UPDATE attribute:

> The maximum output record length or the terminal width.

Files with the INPUT attribute:

> The maximum input record length of the MTS file or device.

For PL/I (F) output nonprint files, the record length will be the minimum of 80 or the output record length (ORL).

The record size of a file may be explicitly given at compile-time using an option of the ENVIRONMENT file description attribute or by using the LINESIZE option of the OPEN statement. For files with the PRINT attribute, LINESIZE does not include the carriage-control character, thus for these files, the record size is equal to LINESIZE plus one, e.g.,

```
OPEN FILE(DATA) LINESIZE(120);
```

The record size of a file may also be explicitly given at run-time in the PAR field of the $RUN command. Record-size information specified at compile-time takes precedence over information specified at run-time, e.g.,

```
$RUN PROGRAM PAR=DATA@U(121)
```

OTHER PL/I STATEMENTS

Below are the statements (besides stream and record PL/I statements), which have different implementations than described in the PL/I Reference Manual. Currently, these are DISPLAY, FETCH, RELEASE, and DELAY statements.

THE DISPLAY STATEMENT

The PL/I DISPLAY statement has the form

```
DISPLAY (element-expression) [REPLY(character-variable)];
```

and may be used to write varying-length character strings to the MTS logical I/O unit SERCOM and to optionally read a varying-length character string from MTS logical I/O unit GUSER. By default, SERCOM and GUSER correspond to *MSINK* and *MSOURCE*, respectively, which is usually the user's terminal in conversational mode. "element-expression" is converted as necessary before it is displayed.

Output from the DISPLAY statement is written with the @CC I/O modifier as the default for the PL/I (F) compiler and the @¬CC I/O modifier as the default for the PL/I Optimizing Compiler.

The following example illustrates the use of the DISPLAY statement.

```
TEST: PROCEDURE OPTIONS(MAIN);
      DECLARE (ANS1,ANS2) CHARACTER(30) VARYING;
      DISPLAY ('Please enter your given name')
         REPLY (ANS1);
      DISPLAY ('Please enter your surname')
         REPLY (ANS2);
      DISPLAY ('Thank you, '||ANS1||' '||ANS2);
END TEST;
```

Terminal session:

```
Please enter your given name
John Doe
Please enter your surname
MD.
Thank you, John Doe MD.
```

In addition to its normal use, the DISPLAY statement can be very useful for debugging when used to print intermediate results. For example,

```
      DECLARE A FIXED DECIMAL(3,1);
      A = 2.3;
      DISPLAY ('A='||A);
```

will output A=2.3 when executed. Note that the || operator in the above two examples indicates concatenation.

The end of file is ignored for the REPLY option.

FETCH AND RELEASE STATEMENTS

Through the use of FETCH and RELEASE statements, the PL/I Optimizing compiler allows the dynamic loading of PL/I external procedures. The PL/I Reference Manual lists restrictions on using fetched procedures in Chapter 6: Program Organization, section "Dynamic Loading of an External Procedure."

All fetchable (dynamically loaded) procedures should be placed in a loader library that is to be attached to the PL/I file LINKLIB. The LINKLIB should be specified in one of several ways: a PLIXOPT external varying character string, an ATTACH subroutine, or as a "run-time option" in the PAR field. *OBJUTIL can be used to create a link library. It is recommended that all external entries other than the actual name of the fetched procedure be deleted from the library directory. At the end of this link library, there should be the line:

```
     $CONTINUE WITH *PL1OPTLIB
```

which will resolve all needed library references.

A fetched procedure should have the same external name that the called program invokes. Various control sections of fetched procedures can be deleted such as PLISTART. Unfortunately, the *LINKEDIT program cannot be used to delete these control sections because this program still has serious problems with objects produced by the PL/I Optimizing compiler.

Below is an example that illustrates the use of dynamically loaded PL/I external procedures.

```
#$list test
>     1     MAIN: PROCEDURE OPTIONS(MAIN);
>     2           DECLARE ABC FLOAT DECIMAL,
>     3                   SUBR EXTERNAL ENTRY (FLOAT DECIMAL),
>     4                   SYSPRINT PRINT FILE;
>     5           /* Define link library to use */
>     6           DECLARE PLIXOPT CHARACTER(100) VARYING STATIC
>     7                   EXTERNAL INITIAL ('LINKLIB=MYLIB');
>     8           ABC = 0;
>     9           FETCH SUBR;
>    10           OPEN FILE(SYSPRINT);
>    11           CALL  SUBR (ABC);
>    12           PUT SKIP LIST(ABC);
>    13           RELEASE SUBR;
>    14           CLOSE FILE(SYSPRINT);
>    15           STOP;
>    16           END;
#$run *pl1opt scards=test spunch=object sprint=-p
 PL/I OPTIMIZER V1 R3.1        TIME: 09.41.20  DATE: 18 AUG 82

 NO MESSAGES PRODUCED FOR THIS COMPILATION

 COMPILE TIME    0.00 MINS        SPILL FILE:     0 RECORDS, SIZE  4051
#09:41:38  T=0.261  $0.13
#$list subr
>     1     SUBR: PROCEDURE (ABC);
>     2           DECLARE ABC FLOAT DECIMAL;
>     3           PUT SKIP LIST (ABC);
>     4           ABC = 1;
>     5           PUT SKIP DATA (ABC);
>     6           ABC = 2;
>     7           RETURN;
>     8           END;
#$run *pl1opt scards=subr spunch=-obj sprint=-pp
 PL/I OPTIMIZER V1 R3.1        TIME: 09.42.26  DATE: 18 AUG 82

 COMPILER DIAGNOSTIC MESSAGES

 ERROR ID L   STMT    MESSAGE DESCRIPTION

 COMPILER INFORMATORY MESSAGES

 IEL0533I I           NO 'DECLARE' STATEMENT(S) FOR 'SYSPRINT'.
 IEL0430I I   1       NO 'MAIN' OPTION ON EXTERNAL PROCEDURE.

 END OF COMPILER DIAGNOSTIC MESSAGES
 COMPILE TIME    0.00 MINS        SPILL FILE:     0 RECORDS, SIZE  4051
#09:42:36  T=0.26  $0.13
#$create mylib
#$run *objutil
*set library=on
*EDIT MYLIB
*ADD -OBJ
 ADDED:    ***SUBR1
*LIST ENTRYS
  ***SUBR1- PLISTART ***SUBR1 ***SUBR2 PLICALLA PLICALLB SUBR
            SYSPINT
*COMMENT - DELETE UNNECESSARY ENTRY POINTS.
*DELETE@DIRECTORY ***SUBR1 ***SUBR2 PLISTART PLICALLA PLICALLB SYSPINT
 DELETED:  ***SUBR1 ***SUBR2 PLISTART PLICALLA PLICALLB SYSPINT
*LIST OMS
  ***SUBR1      3.000
*STOP
```

```
#09:44:15  T=0.056  $0.05
#$edit mylib
:i *L '$CONTINUE WITH *PL1OPTLIB'
:    15     $CONTINUE WITH *PL1OPTLIB
:stop
#$run object+*pl1optlib

  0.00000E+00
 ABC= 1.00000E+00;
  2.00000E+00
#09:44:49  T=0.053  RC=1000  $0.02
```

THE DELAY STATEMENT

The DELAY statement is implemented only by the PL/I Optimizing compiler, not by the PL/I (F) compiler. The DELAY statement can suspend a program for a specified period of time in terms of real-time milliseconds. For example,

```
     DELAY (50);
```

The program is then suspended for 50 milliseconds.

PL/I DATA REPRESENTATIONS

Data in PL/I fall into two categories: problem data or program control data. Problem data contains values that are either arithmetic or string. Program-control data controls program execution and includes labels, events, tasks, locators, and areas. Since tasks and events are not supported in MTS, they are not discussed in this section.

ARITHMETIC DATA

Arithmetic data represent numbers such as 2, -1000, 2.71828, or 4.15E-9. They have four basic characteristics: base, scale, precision, and mode.

Any number can be written in PL/I in DECIMAL or BINARY notation, that is, base 10 or 2, respectively. For example, 33.75 can be written in decimal notation as $3 \cdot 10^1 + 3 \cdot 10^0 + 7 \cdot 10^{-1} + 5 \cdot 10^{-2} = 30 + 3 + 0.7 + 0.05 = 33.75$. In binary notation, there are only two digits, 0 and 1; the same number is written as $10011.11 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 32 + 2 + 1 + 0.5 + 0.25 = 33.75$. People are inclined to use decimal numbers while their computers prefer binary numbers.

The scale of a number is either FIXED or FLOAT. Fixed numbers are only a series of digits with an optional decimal or binary point. Floating numbers consists of two fixed numbers, one representing the mantissa, which consists only of significant digits, and the scale factor, which increases or decreases the magnitude of the number. A floating number is generally written as 4.15E-9, where 4.15 is the mantissa and -9 is the exponent scale factor. It is same as $4.15 \cdot 10^{-9}$.

The precision of a number is generally written in PL/I in either one of the two forms (p) or (p,q) where "p" and "q" are integers. "p" is the number of significant digits, and "q", used only in fixed numbers, specifies the position of a binary or decimal point relative to the rightmost digit. The number 17.76 has the precision (4,2) because there are four significant digits and the decimal point is two digits from the right.

The mode of a number is either REAL or COMPLEX. Real numbers are only single fixed-point or floating-point numbers. Complex numbers are written in the format a + bI, where "a" and "b" are real numbers and "I" is the square root of -1. "a" is the real part of the complex number, and "b" the imaginary part. In PL/I, both parts of a complex variable should have identical base, scale, and precision. FORTRAN users use (a,

b) instead of a + bI, and FORTRAN allows only floating-point parts. In PL/I, real and imaginary parts occupy adjacent fields, with the real part first. Examples of complex numbers are: 1+5I, -3.14I.

Variables not declared explicitly (such as DECLARE statements) or contextually (such as FILE option in a PUT statement) are arithmetic. If the first letter of the variable name is one of I through N, then the variable is real binary fixed with precision (15,0). Otherwise, the variable is real decimal float with precision 6. This is the same as FORTRAN except that variables starting with I through N are of length 2

The DEFAULT statement, implemented only by the PL/I Optimizing compiler, can be used to override the standard default rules or to specify a complete set of programmer-defined default rules.

Fixed-Decimal Data

Many computers such as the Amdahl 470 still operate on decimal numbers with decimal arithmetic operations such as addition, subtraction, multiplication, and division. Fixed-decimal data usually have the default precision (5,0), and the maximum precision is 15. Fixed-decimal data is represented in the computer as packed-decimal numbers. For example, 1234 appears as '01234C'. A leading zero is applied to the left as necessary, and the last hexadecimal digit is interpreted as the sign digit ("C" is positive and "D" is negative). The rest of hexadecimal digits represent the actual decimal digits 0 through 9. Thus, a fixed-decimal number with precision "p" occupies the ceiling of ((p+1)/2) bytes, since every two digits except the last occupy one byte. Fixed-decimal operations are generally slower than fixed-binary operations in the computer. There is no FORTRAN equivalent for fixed-decimal data. Examples of fixed-decimal numbers are 3, 3.14, -40. These constants are automatically converted into fixed-binary numbers by the PL/I compiler if they are used in conjunction with fixed-binary variables.

Fixed-Binary Data

Fixed-binary numbers are written in binary notation followed by the letter "B", e.g., 101B is same as fixed decimal 5 ($101B = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 1 = 5$). These numbers are internally represented either by signed 16-bit binary numbers with precision equal to or less than 15 or by signed 32-bit binary numbers with precision over 15. The equivalent in FORTRAN is INTEGER*2 and INTEGER*4, respectively. The default precision is (15,0) with the maximum being 31. Positive numbers in the computer are represented as binary numbers with the leftmost bit (the sign bit) zero. Negative numbers are put in two's-complement binary notation with the sign bit equal to 1. Thus, -1 is represented as a number with all bits 1, i.e., hexadecimal FFFF. Note that it is impossible to represent a negative zero as a fixed-binary number since the hexadecimal 8000 is interpreted as -32768 (not -0), the minimum value of the fixed-binary numbers with precision less than 16. Examples of fixed-binary numbers are 11B, 11.001B, -10101B. The value range is -32768 to 32767, inclusive, in short precision and -2147483648 to 2147483647 in long precision.

Float-Binary and Decimal Data

Both float-binary and decimal numbers are represented internally as hexadecimal floating-point numbers. These numbers have a scale factor appended such as float binary 11.01E-27B or float decimal 314.16E-2. Note that these numbers are written as fixed binary or fixed decimal, and the scale factors, here represented as the letter "E" followed by an optionally signed decimal integer exponent. For binary numbers, the mantissa is written in binary notation, the exponent in decimal, and the letter "B" is added at the right. The AMDAHL 470 computer implements only three precisions of floating-point numbers: short (float binary with precision $\leq$ 21 or float decimal with precision $\leq$ 6), long (float binary with precision > 21 or float decimal with precision > 6), and extended (float binary with precision > 53 or float decimal with precision > 16). The maximum precision is 109 for binary and 33 for decimal. The extended precision is not supported by the PL/I (F) compiler. The value range is approximately from $5.4 \cdot 10^{-79}$ (or 16^{-65}) to $7.2 \cdot 10^{75}$ (or 16^{63}).

Complex Data

Complex numbers consist of two parts: real and imaginary. The imaginary part is written in PL/I as a signed real number immediately followed by the letter "I". Thus, 0I, -1.45I, 28-0.09I are complex numbers. Unlike FORTRAN, PL/I allows fixed numbers as complex numbers. Complex numbers have their two real and imaginary parts occupy adjacent fields, with the real part first. Both real and imaginary parts of a complex variable must have the same base, same scale, and same precision.

STRING DATA

PL/I recognizes two types of string data: character and bit. Both are contiguous sequences of characters or bits with length from 0 to 32767. These strings can be treated by a program as data. Their length can be either fixed or varying.

Character-String Data

Character strings are sequences of EBCDIC characters delimited by a pair of primes (’). If a prime is to be a part of the character string, it must be written as two adjacent primes with no intervening blanks. Examples are:

```
          ’ABC’       ’HOLY COW’        ’’          ’It’’s A-OK.’
```

These examples represent "ABC", "HOLY COW", a null string, and "It’s A-OK.". Note that lengths of these are 3, 8, 0, and 9, respectively. Null strings have no characters between the primes, and their length is zero, e.g., ’’.

The character string consists of the string value. In addition, the PL/I Optimizing compiler represents varying strings with their current lengths followed by the string values. For example, the varying string ’ABC’ with maximum length of 5 occupies 7 storage bytes and is shown as:

3	’ABC’	Unused

If varying strings have the ALIGNED attribute, the PL/I Optimizing compiler will align them on halfword boundaries.

Bit-String Data

Bit strings are the same as character strings with the following differences. Each character in the sequence can only be a binary digit, i.e., either ’0’ or ’1’. The second prime in the bit strings must be followed immediately by the letter "B". Examples of bit strings are:

```
               ’010’B    ’1’B       ’0’B        ’’B
```

Lengths of these examples are 3, 1, 1, and 0, respectively. The last example is a null bit string. The length can be up to 32767 bits. In the computer, each byte has 8 bits; hence the maximum byte length is 4096 (4095 bytes plus 7 bits). In addition, the PL/I Optimizing compiler prefixes each varying bit string with a halfword current length (in bits).

PROGRAM-CONTROL DATA

Program-control data controls the program. Examples are labels, pointers, areas, and offsets.

Labels

Labels are identifiers that have either a colon prefixed to a statement or declared as LABEL. Those having colons are called label constants. An example shows the label constant NEXT in a statement assigning the value 3.141592 to PI:

```
     NEXT:      PI = 3.141592;
```

The other type of labels are label variables, which should be declared with the LABEL attribute:

```
     DECLARE TOGO LABEL;
```

Label variables can have statement labels assigned to them. The exceptions are ENTRY and PROCEDURE labels, which cannot be assigned to label variables. The example below shows how label variables are used:

```
            DECLARE TOGO LABEL (FIRST, LAST);
             ...
            TOGO = FIRST;
            GO TO PROCESS;
     FIRST: TOGO = LAST;
             ...
            GO TO PROCESS;
     LAST:  STOP;
             ...
     PROCESS:
             ...
            GO TO TOGO;
```

The DECLARE statement indicates that the label variables can have one of two labels, FIRST or LAST, as values.

Pointers

Pointers are used as locator variables to point at any data. These pointers occupy 4-byte words in the computer. A pointer, say P, pointing to the string variable XYZ will have as a value the address of XYZ. In assembly language code, it is A(XYZ). Note that the dope vector of XYZ is not pointed to. A pointer can be assigned a NULL value, as a value that cannot have an address in the computer. It is the hexadecimal FF000000. The pointer P is assigned the value of XYZ by the use of the built-in function ADDR:

```
     P = ADDR (XYZ);
```

Based variables can be allocated by the ALLOCATE statement, which obtains storage of the length rounded up to the nearest multiple of 8. For full information on the pointers, consult Chapter 8, "Storage Control," section "Based Storage," of the OS PL/I Checkout and Optimizing Compilers: Language Reference Manual.

Another kind of locator variable is used in conjunction with areas. These are called offsets, which point to a based variable relative to the start of a based area. The contents of an offset is similar as that of a pointer, except that the address in the offset is not an absolute machine address, but the address relative to the start of an area.

Areas

Areas are used in PL/I to allocate based variables. The length of an area may be specified; if not, it is assumed to be 1000. Area variables can be declared thus:

```
     DECLARE A AREA (2000);
```

Here, the area variable A has length 2000. The actual size of the area should include 16 control bytes, controlling the allocation of based variables. Thus, the area size of A is 2016. Based variables are allocated in the area by the following statement:

```
     DECLARE B BASED (P);
     ALLOCATE B IN (A);
```

When B has been allocated, the pointer P contains the actual location of B. Each based variable is allocated on a doubleword boundary. Actually, it is not necessary to use the defined pointer P of the variable B, since another pointer, say Q, can be used:

```
     DECLARE Q POINTER;
     ALLOCATE B IN (A) SET (Q);
```

Here Q will point to the based variable B; the pointer P remains unchanged.

The two allocations of the variable B can be freed, respectively:

```
     FREE B IN (A);
     FREE B IN (A) SET(Q);
```

In both cases, free lists replace the allocations and then are chained to all previous free lists of the area A. The user should be careful not to free an "unallocated" based variable. In that case, he may obtain unexpected results such as an addressing interrupt.

Areas can be emptied, that is, all allocated elements may be freed. This is accomplished by a single assignment statement with the EMPTY built-in function:

```
     A = EMPTY();
```

When areas are allocated, they are always emptied.

Offsets

Offset variables are always associated with area variables. Unlike the PL/I Optimizing Compiler, the PL/I (F) compiler restricts the area variables to be an unsubscripted level 1 based area variable. Offsets are similar to pointers except that they point to the location relative to the start of the area. An advantage of offsets becomes evident when an area is assigned or transmitted to another area, since the offsets remain the same while the pointers of the based variables in the area will have to be changed. The offsets are always declared in the following manner:

```
     DECLARE O1 OFFSET (A);
```

where "A" is a based area variable. The actual contents of an offset is a fullword pointing to the real location minus the actual start of the area. 16 control bytes are added to the area and are counted for the offsets. Thus, the first allocated variable in the area will have the offset 16 (not 0).

The offsets are set in various ways:

```
   (1)  ALLOCATE B IN (A) SET (O1);

   (2)  ALLOCATE B IN (A);
        O1 = OFFSET (P, A);

   (3)  ALLOCATE B IN (A);
        O1 = P;
```

where P is the pointer for the based variable. Example (2) is only possible with the PL/I Optimizing compiler, since both OFFSET and POINTER are built-in functions that are not available with the PL/I (F) compiler. For more information, consult Chapter 8, "Storage Control," of the _OS PL/I Checkout and Optimizing Compilers: Language Reference Manual_, form GC33-0009.

ARRAYS

Arrays in PL/I are arranged in row-major order, upward in storage. The main difference between PL/I and other languages is that arrays are in column-major order but always contiguous in other languages. Arrays in PL/I are not necessarily contiguous, especially for array structures. Besides, the subscripts in PL/I are limited to the range of -32768 to +32767. In FORTRAN, lower bounds are 1, but upper bounds may exceed +32767 provided that the total amount of the array storage in bytes does not exceed one segment (1,048,576 bytes). For arrays declared without lower bounds, the PL/I compiler will set the lower bounds to 1. Thus, if the user inserts:

```
     DECLARE VAR (5,6);
```

The bounds are 1 through 5 for the first dimension, and 1 through 6 for the second dimension. Arrays can be declared with lower and upper bounds thus:

```
     DECLARE TABLE(2:5);
```

2 is the lower bound, and 5 the upper bound. The lower bound is separated from the upper bound by a colon, and each pair of bounds is separated from the others by a comma.

STRUCTURES

A structure is an aggregate of data that can be subdivided into data items. It can be thought of as a hierarchical collection of variables, the top of which is the structure itself, and at bottom are the individual variables. Every variable in a structure should be preceded by a level number from 1 to 255 indicating its level. In general, the variable at the top is the entire structure with level 1 and is called a major structure. This structure in turn can have some other structures (called minor structures) or any other variables. Although level numbers cannot exceed 255, the maximum number of levels that may be used is 63.

A page, for example, can be thought of as a structure, thus:

```
     DECLARE 1 PAGE,
               2 TITLE CHARACTER(120),
               2 SUBTITLE CHARACTER(120),
               2 LINES (52),
                 3 PRINT_LINE CHARACTER(120),
                 3 UNDERSCORE_LINE BIT(120),
               2 FOOTNOTES (5),
                 3 PRINT_LINE CHARACTER(120),
                 3 UNDERSCORE_LINE BIT(120),
               2 BOTTOM CHARACTER(120);
```

Here a page is laid out with a title, a subtitle, 52 lines, 5 footnote lines, and a bottom line. The structure variable is PAGE, a major structure. This in turn consists of five subdivisions: TITLE, SUBTITLE, LINES, FOOTNOTES, BOTTOM. Two of these, LINES and FOOTNOTES, are called minor structures since they can be divided into two lines: PRINT_LINE which holds the text line and UNDERSCORE_LINE which indicates what position of the PRINT_LINE is underscored. Note that PRINT_LINE is ambiguous because it is a part of two minor structures, LINES and FOOTNOTES. So the PL/I compiler allows qualified identifiers, e.g., LINES.PRINT_LINE refers to PRINT_LINE of the structure LINES. Similarly, FOOTNOTES.PRINT_LINE is that of FOOTNOTES.

Before assigning the addresses to the members of a structure, the PL/I compiler first considers the alignment of each structure member, its length, and its relative position from the start of the major structure. This process is called the "structure mapping." It attempts, for instance, to minimize padding while ensuring the proper alignment of each member. Data, such as bit strings, may be either aligned or unaligned, while some other data (especially labels, pointers, offsets, and areas) must be aligned. Fixed-length bit strings, for example, are aligned on byte boundaries if declared with attribute ALIGNED; otherwise, they are aligned on the first available bit. Complete details of the structure mapping can be found in Section K: "Data Mapping" of the PL/I Language Reference Manual, GC28-8201 or GC33-0009.

EXTERNAL AND INTERNAL ATTRIBUTES

A variable is either INTERNAL or EXTERNAL. An internal variable is known only in the declaring block and its containing blocks. An external variable, on the other hand, is known by any two external procedures provided that the variable is declared with the same attributes as well as EXTERNAL. An internal variable in a procedure is not known by any external procedure unless it is passed as a parameter to the procedure. If not declared explicitly, a variable is assumed to be internal, unless it is a file, an entry name of the external procedure, or a user-defined CONDITION. There is a PL/I restriction that the name of the external variable should not be more than 7 letters long. Otherwise, the compiler will issue the message IEM2867I or

IEL0966I and form a 7-letter name from the first four and the last three letters of the original name.

STORAGE ALLOCATION

There are four different kinds of storage allocations for PL/I variables: STATIC, AUTOMATIC, CONTROLLED, and BASED. A _static_ variable is allocated at the start of a program and remains allocated throughout the execution. An _automatic_ variable is automatically allocated when a PROCEDURE or BEGIN block defining the variable is entered. Automatic variables are automatically freed when the block is no longer active. A _controlled_ variable is directly controlled by the user through the means of the ALLOCATE and FREE statements. In addition, a controlled variable can have two or more allocations at a time. Allocations of controlled variables are stacked, only the latest allocation can be referenced until the FREE statement pops the allocation out of the stack. The PL/I compiler provides the built-in function ALLOCATION so that the programmer can test whether the controlled variable is allocated or not. A _based_ variable is similar to a controlled variable in that it is allocated by the ALLOCATE statement and freed by the FREE statement. By use of pointer variables, the programmer can address a particular allocation of the based variable, e.g., P->B or Q->B.

Static variables are allocated depending whether they are internal or external. Internal static variables are always stored in the static internal control section of the external procedure. The name of this control section is formed from the external procedure name, extended on the right with the letter "A" for PL/I (F) or "2" for PL/I Optimizer and padded with asterisks to eight characters. For example, for the external procedure BLOCK, the name of the internal static control section is **BLOCKA or **BLOCK2. An external static variable is allocated in a control section with same name as that of the variable.

Automatic variables are allocated only at the start of a PROCEDURE or BEGIN block that declares them. There are two kinds of automatic variables. If the total length of an automatic variable is known at the compile time, the PL/I compiler will allocate it in the _dynamic save area_ (DSA) of the declaring block. All other automatic variables, such as strings with adjustable lengths and arrays with adjustable bounds, are allocated in the _variable data areas_ (VDA). The VDA is chained to the previous VDA or DSA.

Controlled variables are allocated by means of their pseudo-registers. If the variable is external, the name of the pseudo-register is the same as that of the variable. If internal, then the PL/I compiler will automatically generate a unique name based on the external procedure name. When a controlled variable is allocated, the address is placed in the contents of its pseudo-register, and the new allocation has the chain-back address set to the previous allocation. If there is no previous allocation, the address will be zero in PL/I (F) or the address of a dummy FCB in PL/I Optimizer.

Based variables are allocated either within the area or without. If allocated within the area, space in the area is set for the variable. Otherwise, a storage core is obtained via the GETSPACE call. Each time a based variable is allocated, its pointer is set to the first byte of the variable. Allocations of a based variable are not stacked, as each allocation is referenced by an appropriate pointer value.

PL/I PUBLIC FILE DESCRIPTIONS

The following public file descriptions are taken from MTS Volume 2, Public File Descriptions. These are public files which may be of use to PL/I programmers.

*PL1SCAN

Contents:	The IBM PL/I (F) source program scanner with a modified interface for operation under MTS.
Purpose:	To provide a *rough* prescan (or syntax check) of programs written in PL/I (F).
Use:	The program is invoked by the $RUN command.
Alt. Name:	*PL/1SCAN

Logical I/O Units Referenced:

SCARDS - source program to be scanned.
SERCOM - error messages from the scanner.
SPRINT - listing of the input source program.
SPUNCH - file to which the source is written.

Parameters: The following parameters may be specified in the PAR field of the $RUN command or in the %PROCESS batch option statement. The parameters must be separated by a comma or by one or more blanks.

SOURCE NOSOURCE	prints the source on SPRINT (this defaults to SOURCE in batch only).
CHAR48	specifies the 48-character set.
CHAR60	specifies the 60-character set (the default).
SORMGIN=(m,n)	specifies the left-hand (m) and right-hand (n) source margins. These default to 1 and 72, respectively. The right-hand margin must be equal to or less than 100.

The parameters SOURCE, NOSOURCE, CHAR48, CHAR60, and SORMGIN may be abbreviated to S, NS, C48, C60, and SM, respectively.

Description: *PL1SCAN scans a PL/I (F) source program for syntax errors. However, it *cannot* detect all errors, because it only scans one statement at a time and does not check interstatement dependencies (i.e., DECLARE statements, undefined labels, etc.).

If SPUNCH is specified, the PL/I scanner will write the source program to the file specified. This is especially useful if SCARDS is assigned to the terminal (*SOURCE*).

Comments: It is often advantageous to run *PL1SCAN since it uses considerably less CPU time and virtual memory than the PL/I compiler.

When a listing of the source code is obtained, each source line is preceded by its line number (in a style similar to MTS).

The error messages are of the form

```
IKMxxx line# error-message-text
```

where "line#" is the line number of the offending statement.

<u>*PL1TIDY</u>

Contents:	The object file of the PL/I "tidy" program.
Purpose:	To edit PL/I source programs into an easily readable format.
Use:	The program is invoked by the $RUN command.
Alt. Name:	*PL/1TIDY

Logical I/O Units Referenced:

SCARDS -	the source program to be edited.
SPRINT -	a listing of the edited source program and diagnostics.
SPUNCH -	the edited source program.
SERCOM -	error messages plus a message for each external procedure.

Parameters: The following parameters may be specified in the PAR field of the $RUN command or inserted in the PL/I comment /*TIDYPAR=.. .*/. The parameters must be separated by a comma or by one or more blanks. In case of conflicting parameters, the rightmost parameter takes precedence. The default case is underlined.

<u>LIST</u>	produces a listing of the edited source program. This is the default. If, however, SPRINT is assigned to a terminal by default, then the default becomes NOLIST.
NOLIST	suppresses the listing of the edited source program.
<u>DECK</u>	produces the edited source program.
NODECK	suppresses the edited source program.
<u>LOGICAL</u>	produces an easily readable form of the source program in which each statement is indented by an amount corresponding to its nesting depth.
COMPRESS	produces a compressed form of the source program in which all non-significant blanks are removed.
INSET=n	specifies that statements are to be indented "n" additional columns for

	each level of nesting depth. The default is 3 columns.
MARGIN=m	A statement at level one starts at the margin "m". The default is 10.
COMMENT	Blanks within comment brackets will be retained.
<u>NOCOMMENT</u>	Two or more adjacent blanks within comment brackets will be replaced by one blank.
DECLARE	Each DECLARE statement will be broken into parts separated by level-one commas. Each part will then be properly indented.
<u>NODECLARE</u>	Declare statements will be entirely compressed.
PAGE	The next or current statement will be on the next page. If the output carriage control is specified by OSORMGIN, its location will contain a "1".
DEFAULT	All defaults are restored at this point.
RESET	The logical level is set to zero. A skip to the top of a new page is done.
STMT=n	The current statement number is set to "n".
PAR	PL1TIDY parameters in /*TIDYPAR=...*/ are retained in the output.
<u>NOPAR</u>	Comments of the form /*TIDYPAR=...*/ are removed.
ISORMGIN=(a,b,c)	specifies the left margin "a", the right margin "b", and the optional carriage-control location "c" for the input source text (from SCARDS). The default is (1,72). The left and right margin specifications must be in the range (1,255). The carriage-control location, if specified, must be in the interval $1 \le c < a$ or in the interval $b < c \le 255$.

OSORMGIN=(a,b,c)	specifies the left margin "a", the right margin "b", and the optional carriage-control location "c" for the edited source program (onto SPUNCH). The default is (1,72). The left- and right-margin specifications must be in the range (1,255). The carriage-control location, if specified, must be in the interval 1≤c<a or in the interval b<c≤255.
LC	specifies that the input is to be converted to lower case. Comments and character strings are not converted.
<u>MC</u>	specifies that no conversion to upper case or lower case takes place.
UC	specifies that the input is to be converted to upper case. Comments and character strings are not converted.
CHAR48	specifies that the 48-character set is used for the input source. See the Language Reference Manuals for the PL/I (F) and Optimizing compilers for the character set.
<u>CHAR60</u>	specifies that the input source is written in the 60-character set. See the Language Reference Manuals for the PL/I (F) and Optimizing compilers for the character set.
BCD	specifies that the input source is written in BCD (Binary Coded Decimal). See the Language Reference Manuals for the PL/I (F) and Optimizing compilers for the BCD code.
<u>EBCDIC</u>	specifies that the input source is written in EBCDIC (Extended Binary Coded Decimal Interchange Code). See the Language Reference Manuals for the PL/I (F) and Optimizing compilers for the EBCDIC code.
<u>SEQ</u>	specifies that the edited source program shall have a sequence-ID field which consists of the nesting level of the statement and the sequence number of the statement. This sequence field is placed in the eight columns immedi-

	ately to the right of the right margin of the edited source program.
SEQ=xxxx	specifies that the edited source program shall have a sequence-ID field of xxxx0001 on the first output record, xxxx0002 on the second, etc. This sequence-ID field is placed in the eight columns immediately to the right of the right margin of the edited source program.
NOSEQ	specifies that the edited source program shall have no sequence-ID field.

Description: *PL1TIDY edits PL/I source programs into an easily readable form. This feature is especially desirable to clean up files containing modifications made from terminals. The program will indent each statement by an amount corresponding to its nesting depth. This program can also be used to edit programs into an acceptable form which cannot be compiled because of the rather stringent SORMGIN constraints of *PL1 and *PLC. Programs also can be edited into "compressed" form with all nonsignificant blanks removed, thus minimizing the size requirements of the file containing the source program.

*PL1TIDY processes the following control statements, which are implemented by the PL/I Optimizing compiler: %PRINT, %NOPRINT, %PAGE and %SKIP(n). These statements are produced on the output SPUNCH, and the SPRINT listing, if the LIST option, is in effect is controlled by the statements. %PRINT resumes the printing of the output; and %NOPRINT suppresses it. %PAGE causes the skip to the top of the next page. %SKIP or %SKIP(n) prints "n" blank lines; if "n" is not specified, one blank line is printed. All these control statements should be terminated by a semicolon.

When the program detects an error, it places the comment "*ERROR*" is the sequence-ID field (even if NOSEQ is specified).

For each external procedure, a message is produced on SERCOM:

```
Procedure ABC:  4 statements, 2 errors.
```

Examples:

```
$RUN *PL1TIDY SCARDS=IN SPUNCH=OUT
```

In the above example, the input source program is read from the file IN. The edited output is written to the file OUT.

```
$RUN *PL1TIDY SCARDS=IN SPUNCH=OUT PAR=NOSEQ,INSET=2
```

The above example is the same as the previous example except that sequence-ID fields are not produced for each output record and that INSET is 2 instead of the default of 3.

```
$RUN *PL1TIDY SCARDS=IN SPUNCH=OUT PAR=ISORMGIN=(1,255)
```

The example converts free-format source input with SORMGIN=(1,255) to an edited format with SORMGIN=(1,72).

PL/I LIBRARY SUBROUTINES

This section contains descriptions of the subroutines that are a part of the PL/I library *PL1LIB. Only one of these subroutines, ATTACH, resides in *PL1OPTLIB.

For PL/I (F) programs, each of these subroutines may be called directly. Many other subroutines that require an S-type calling sequence may be called by using the PLCALL subroutine which is described in the section "Interlanguage Communication Facilities."

<u>ATTACH</u>

SUBROUTINE DESCRIPTION

Purpose: To associate a PL/I file variable name with an appropriate MTS file or device name.

Location: *PL1LIB and *PL1OPTLIB

Calling Sequences:

PL/I: CALL ATTACH(string);

Parameters:

<u>string</u> is a character string of either fixed or variable length which must follow these restrictions:

(1) the string must not be a null string,
(2) the length of the string must not be more than 255 characters, and
(3) the string must conform to that of PAR=<u>string</u>.

Description: The subroutine passes <u>string</u> to an internal routine which processes the PAR=<u>string</u> format (see "PL/I File Specifications" in the section "Running a PL/I Program" in this volume).

Example:

```
          CALL ATTACH('A=X B=Y@F(80)');
```

This example associates PL/I files A and B with X (an MTS file) and with Y (another MTS file with fixed format of length 80).

BATCH

SUBROUTINE DESCRIPTION

Purpose: To determine whether the user is in batch or conversational mode.

Location: *PL1LIB

Calling Sequences:

PL/I:

```
DECLARE BATCH ENTRY
RETURNS (BIT(1));
```

Description: The subroutine returns '1'B if the user is in batch mode; otherwise, it returns '0'B.

Example:

```
IF BATCH THEN STOP;
ELSE GOTO RETRY;
```

In this example, if the program is running in batch mode, it stops; otherwise, it transfers to the label RETRY.

CNTL

SUBROUTINE DESCRIPTION

Purpose: To provide an interface between the PL/I user and the CONTROL entry in the device support routines (DSRs). This subroutine allows the PL/I user to execute control operations on files and devices. See the CONTROL subroutine description in MTS Volume 3.

Location: *PL1LIB

Calling Sequence:

PL/I: CALL CNTL(fdname,info)

Parameters:

fdname	is a CHARACTER variable or constant giving an MTS file or device name.
info	is a CHARACTER variable or constant giving the control information to be passed to the device support routines.

Return Codes:

0	Successful return from CONTROL.
>0	Unsuccessful return from CONTROL. The PL1RC subroutine may be used to interrogate the return code.

Note: The user should exercise care when using the CNTL subroutine if the PL/I file to which fdname refers is open since the PL/I library routines do not search PL/I files for an fdname that would match.

Examples:

```
CALL CNTL('*T*','REW');
IF PL1RC¬=0 THEN GOTO NOREW;
```

This example calls CONTROL to rewind the tape *T*, and then checks to see if the rewind operation was successful.

```
CALL CNTL('*SINK*','DON''T');
```

This example calls CONTROL with the Data Concentrator or Memorex device support command DON'T.

CPUTIME

SUBROUTINE DESCRIPTION

Purpose: To obtain the CPU time (in seconds) used since the beginning of execution of the current program.

Location: *PL1LIB

Calling Sequences:

PL/I:
```
DECLARE CPUTIME ENTRY
RETURNS (FLOAT BINARY);
```

Description: The subroutine returns the floating-point value of the CPU time (in seconds) used since the beginning of program execution.

Example:
```
START_TIME: PROC;
   DCL (TIME1, TIME2) STATIC FLOAT BIN,
       CPUTIME ENTRY RETURNS (FLOAT BIN);
   TIME2 = CPUTIME;
   RETURN;
TIME:  ENTRY FLOAT BIN;
   TIME1 = TIME2;
   TIME2 = CPUTIME;
   RETURN (TIME2 - TIME1);
END;
```

This example determines the amount of CPU time taken in executing a loop. It first calls START_TIME to initialize the variable TIME2; then, on every call, the procedure TIME returns the CPU time in seconds since the previous call.

ELAPSED

SUBROUTINE DESCRIPTION

Purpose: To obtain the elapsed time (in seconds) used since the beginning of execution of the current program.

Location: *PL1LIB

Calling Sequences:

PL/I:
```
DECLARE ELAPSED ENTRY
RETURNS (FLOAT BINARY);
```

Description: The subroutine returns the floating-point value (in seconds) used since the beginning of program execution.

Example:
```
PUT EDIT ('ELAPSED TIME - ',ELAPSED,'SECS')
         (A,F(15,3),A);
```

This example prints out the elapsed time in seconds since the beginning of the program.

FINFO, TFINFO, RFINFO

SUBROUTINE DESCRIPTION

Purpose: To obtain information on a file or device attached to a PL/I file.

Location: *PL1LIB

Calling Sequence:

```
PL/I:     DECLARE FINFO ENTRY(FILE) RETURNS(POINTER);
          DECLARE RFINFO ENTRY(POINTER);
            infob=FINFO(pl1file);
            CALL RFINFO(infob);

          DECLARE TFINFO ENTRY(FILE,CHARACTER(*))
            RETURNS(POINTER);
          DECLARE RFINFO ENTRY(POINTER);
            infob=TFINFO(pl1file,title);
            CALL RFINFO(infob);
```

Description: Given the PL/I file as an argument, the FINFO subroutine returns the pointer value as the address pointing to the GDINFO buffer _infob_. If the buffer is not available, it returns the null pointer. This buffer is exactly as described in the GDINFO subroutine description in MTS Volume 3.

The TFINFO subroutine does exactly the same as the FINFO subroutine except that it associates the PL/I file name with the second argument declared as a character string. If a PL/I user wants to open a PL/I file with the TITLE option and wants to inquire for the information on the file, then he should use the TFINFO subroutine with the second argument equal to the expression in the TITLE option.

The RFINFO subroutine should be called to release the information buffer _infob_ when it is no longer needed.

Example:

```
          DECLARE FINFO ENTRY(FILE) RETURNS(POINTER),
            1 INFO BASED(INFOB),
                2 FDUB POINTER,
                2 TYPE CHARACTER(4),
                2 INP_MAX FIXED(15) BINARY,
                2 OUT_MAX FIXED(15) BINARY,
                2 FDUBTYPE BIT(8),
                2 TYPEINDX BIT(8),
                2 SWITCHES BIT(8),
                2 RESERVED BIT(8),
                2 IOMODIFIER BIT(32),
                2 START_L# FIXED(31) BINARY,
                2 LAST_L# FIXED(31) BINARY,
                2 END_L# FIXED(31) BINARY,
                2 L#_INCR FIXED(31) BINARY,
                2 FDNAME_PTR POINTER,
                2 ERROR_PTR POINTER,
            1 FDNAME BASED(FDNAME_EQU_PTR),
                2 LTH FIXED(15) BINARY,
                2 NAME CHARACTER(I REFER (LTH));
          DECLARE RFINFO ENTRY(POINTER);
                .
                .
          INFOB=FINFO(SPRINT);
          FDNAME_EQU_PTR=INFO.FDNAME_PTR
                .
                .
          CALL RFINFO(INFOB);
```

The FINFO subroutine is called to obtain information about SPRINT; then, the RFINFO subroutine is called to release the information buffer after it is no longer needed.

IHEATTN

SUBROUTINE DESCRIPTION

Purpose: To allow a PL/I program to be notified of the occurrence of an attention interrupt.

Location: *PL1LIB

Calling Sequence:

PL/I: CALL IHEATTN;

Description: The IHEATTN subroutine is automatically called before the main procedure obtains control. This is to allow all attention interrupts to be controlled by the subroutine. The user may override this call to IHEATTN by calling the MTS subroutine ATTNTRP, thus effectively resetting the attention interrupt conditions as if IHEATTN was not called. These conditions can be restored by calling IHEATTN.

Once IHEATTN has been called and an attention interrupt occurs, IHEATTN scans through all active procedures (the most recent first) and tests for any statement beginning with "ON CONDITION(ATTN)". If no such statement has been executed, the condition "ON CONDITION(ATTN) SYSTEM;" is assumed.

The subroutine will take one of the following actions:

(1) If the keyword "SYSTEM;" is specified, a message such as

```
         ┌           ┐    ┌       ┐
         | STMT dddd |    | PROC  |
 ATTN AT |    or     | IN |  or   | name
         |OFFSET xxxx|    |ON-UNIT|
         └           ┘    └       ┘
```

is printed to identify the location of the interrupt. After the message is printed, the subroutine MTS is called and a return is made to MTS command mode (or debug mode). The user may use the contents of general register 1 which points to the standard 72-byte save area from ATTNTRP to obtain the PSW and registers at the time of the interrupt. The first eight bytes contain the PSW, and the remainder of the region contains the contents of the registers. A $RESTART command may be given to restart the program.

(2) If the keyword "SNAP" is specified after "ON CONDITION(ATTN)", then the above message is printed followed by a list of all active procedures at the time of the interrupt.

(3) If the keyword "SYSTEM;" is not specified, then the ON-unit is entered as a procedure with the attention conditions restored. If the ON-unit returns, IHEATTN automatically returns to the interrupted statement. Caution should be exercised to prevent infinite loops in the ON-unit. It is recommended that the user insert "ON CONDITION(ATTN) SYSTEM;" after "ON CONDITION(ATTN) BEGIN;". No I/O may be performed on a PL/I file being interrupted by an attention.

Examples: If the PL/I program that contains no statement beginning with "ON CONDITION(ATTN)" is executed, an attention interrupt will produce a message such as

```
ATTN AT STMT 0021 IN PROC PROGRAM
```

Attention interrupts may be controlled in a PL/I program by the following sequence:

```
DECLARE ATTNSW BIT(1) INIT('0'B);
ON CONDITION(ATTN) SNAP BEGIN;
   ON CONDITION(ATTN) SYSTEM;
   IF ATTNSW = '1'B THEN CALL MTS;
   ATTNSW = '1'B;
   END;
```

When an attention interrupt occurs for the first time, the attention interrupt message is printed followed by a list of the active procedures. Then the BEGIN block, which resets the ON-condition for attention interrupts and sets ATTNSW to '1', is executed; a return is then made to the statement in which the attention interrupt occurred and program execution is resumed. A subsequent attention interrupt will cause the program to print another interrupt message and then return to MTS command mode. The switch ATTNSW may be used by the program to test whether the first attention interrupt has occurred.

IHEDUMC, IHEDUMP

SUBROUTINE DESCRIPTION

Purpose: To dump the program.

Location: *PL1LIB

Calling Sequence:

```
PL/I:     CALL IHEDUMx;
```

where x is

C	dump the contents of storage and then continue processing, or
P	dump the contents of storage and then terminate processing.

Description: Storage dumps are rarely, if ever, used when programming in PL/I because of the powerful debugging aids available to help pinpoint source-program errors. If, however, it becomes necessary to get a storage dump, the user should insert the following statement in the program:

```
CALL IHEDUMP;
```

The name of the dump file used by IHEDUMP or IHEDUMC is PL1DUMP. "PL1DUMP=*SINK*" is assumed in batch mode, unless overridden by the execution parameter or by the ATTACH subroutine. In conversational mode, the user is prompted to supply a file/device name for the dump output, e.g.,

```
 PL1DUMP - SPECIFY FDNAME OR SEND END-OF-FILE
?*PRINT*
```

If the user sends an end-of-file, storage will not be dumped. Both IHEDUMP and IHEDUMC provide a good deal of information on files currently used plus chain-back of save areas and the dump output of the storage contents.

Example:

```
ON ERROR BEGIN;
   CALL ATTACH('PL1DUMP=*PRINT*');
   CALL IHEDUMP;
   STOP;
END;
```

This example automatically dumps the contents of the storage on *PRINT* when an error occurs.

IHENOTE, IHEPNT

SUBROUTINE DESCRIPTION

Purpose: To provide an interface between the PL/I user and the NOTE and POINT subroutines.

Location: *PL1LIB

Calling Sequence:

```
PL/I:     CALL IHENOTE(file,ptrs);
          CALL IHEPNT(file,ptrs,bits);
```

Parameters:

- file is a FILE variable which must first be opened either implicitly or explicitly.
- ptrs is an array of four fullword elements.
- bits is a BIT(4) variable or constant. The bit switches are:

  ’0001’B - set read pointer
  ’0010’B - set write pointer
  ’0100’B - set last pointer
  ’1000’B - set last line number

  More than one switch may be set to give the desired combination of pointers, e.g., ’1111’B sets all pointers.

Return Codes:

The subroutine PL1RC may be used to determine the return codes from NOTE and POINT.

Note: These two subroutines are intended for interaction with the two PL/I subroutines IHEREAD and IHERITE.

Example:

```
DECLARE IHEPNT ENTRY(FILE,(4) FIXED BINARY(31),
  BIT(4)), QQSV FILE,
  PTRS (4) FIXED BINARY(31);
PTRS=0;
CALL IHEPNT(QQSV,PTRS,’0001’B);
```

This example rewinds the file QQSV for input only.

<u>IHEREAD, IHERITE</u>

SUBROUTINE DESCRIPTION

Purpose: To read (IHEREAD) or write (IHERITE) a record from a PL/I file.

Location: *PL1LIB

Calling Sequences:

PL/I:
```
CALL IHEREAD(buff,[len,]mod,lnr,file);
CALL IHERITE(buff,[len,]mod,lnr,file);
```

Parameters:

<u>buff</u> is the CHARACTER variable or constant to be read or written. If IHEREAD is called, <u>buff</u> must be a varying CHARACTER variable if <u>len</u> is omitted; otherwise, <u>buff</u> must be a fixed-length CHARACTER variable if <u>len</u> is supplied.

<u>len</u> (optional) is the FIXED BINARY(15) variable or constant giving the length of the record to be read or written. If omitted, the length of <u>buff</u> is used as the record length.

<u>mod</u> is the BIT(32) variable or constant defining 32 modifier bits used to control the action of the I/O subroutine (see the section "I/O Modifiers" in MTS Volume 3, <u>System Subroutine Descriptions</u>). First eight bits should be set to zero.

<u>lnr</u> is the FIXED DECIMAL(9,3) variable or constant giving the line number to be read or written. Notice that this declaration restricts the line-number range to (-999999.999, +999999.999).

<u>file</u> is the PL/I FILE variable to be used in the I/O operation. This must be a record file with undefined format or unblocked fixed format. It cannot be an output file for IHEREAD, nor an input file for IHERITE. An update file can be used for both IHEREAD and IHERITE.

Description: The PL/I user should note the following restrictions:

(1) Care must be taken if these subroutines are to be mixed with READ, WRITE, or REWRITE statements.

(2) If the indexed bit of a modifier is on, a line number must be provided. Otherwise, a data exception (program interrupt) may occur, or unpredictable results will occur. In addition, in case of IHEREAD, the character string will become a null

string or length set to zero when there is no line associated with the line number.

(3) With IHEREAD, a line number must be within the range (-999999.999,+999999.999). If not, a fixed-overflow exception (program interrupt) will occur.

IHEREAD and IHERITE can raise seven error conditions:

(1) An ENDFILE condition is raised on an input operation with return code=4 and @INDEXED modifier bit off. The message is "IHE140I - END OF FILE ENCOUNTERED."
(2) A RECORD condition is raised if length of a record exceeds the maximum length of the buffer argument for IHEREAD call. The message is "IHE111I - RECORD VARIABLE SMALLER THAN RECORD SIZE."
(3) An input TRANSMIT condition is raised if the return code > 4 on IHEREAD. The message is "IHE120I - PERMANENT INPUT ERROR."
(4) An output TRANSMIT condition is raised if the return code > 0 on IHERITE. The message is "IHE121I - PERMANENT OUTPUT ERROR."
(5) An ERROR condition is raised if the file is stream I/O or is V-formatted or blocked F-formatted. The message is "IHE029I - UNSUPPORTED FILE OPERATION."
(6) An ERROR condition is raised if IHEREAD is used on an OUTPUT file. The message is "IHE020I - ATTEMPT TO READ OUTPUT FILE."
(7) An ERROR condition is raised if IHERITE is used on an INPUT file. The message is "IHE021I - ATTEMPT TO WRITE INPUT FILE."

Example:

```
MAIN:  PROCEDURE OPTIONS(MAIN);
     DCL (IHEREAD,IHERITE) ENTRY
          (,BIT(32),DEC FIXED(9,3),FILE),
          BUFFER CHAR(121)VARYING,
          MOD BIT(32) INIT((32) '0'B),
          LINENR DEC FIXED (9,3), NUTS FILE;
     ON ENDFILE (NUTS) GO TO FINISH;

OVER:  CALL IHEREAD(BUFFER,MOD,LINENR,NUTS);
     PUT SKIP LIST (LINENR,BUFFER);
     GO TO OVER;   /*THIS ACTS LIKE A "$LIST" COMMAND*/

FINISH:
     CLOSE FILE(NUTS); OPEN FILE(NUTS) UPDATE;
     SUBSTR(MOD,31) = '1'B; /* TURN INDEXED BIT ON */
     CALL IHERITE('',MOD,1.0,NUTS); /* DELETE LINE 1 */
     CALL IHERITE(' THIS IS LINE #2.5',MOD,2.5,NUTS);
               /* INSERT THE LINE #2.5 */

RETURN;
END MAIN;
```

IHESARC

Purpose: To set the return code from inside a PL/I procedure.

Location: *PL1LIB

Calling Sequence:

PL/I:
```
DECLARE IHESARC ENTRY(FIXED BINARY(31));
CALL IHESARC(x);
```

Parameter:

_x_ is the value of the return code to be set.

Description: This subroutine sets the return code to the value _x_ from inside a PL/I procedure. This return code may be interrogated by the PL1RC subroutine after the procedure has returned.

Example:
```
DECLARE IHESARC ENTRY(FIXED BINARY(31));
CALL IHESARC(8);
```

The above example sets the return code to the value 8.

IHETABS

Purpose: To set the tab positions for list-directed and data-directed output print files and to provide default page and line sizes.

Location: *PL1LIB

Description: The 360/370-assembly code below describes the IHETABS module in *PL1LIB. This module may be changed or replaced to fit the user’s own requirements.

```
TAB     TITLE 'IHETAB - PRINT FILE LAYOUT'
IHETAB  CSECT
        ENTRY IHETABS
IHETABS DC    H'60'      Default page size
        DC    H'121'     Default line size
        DS    X          Reserved for left and right
        DS    X            margin facilities
        DC    FL1'5'     Number of tab positions.
*                         If zero, then tab positions are
*                         not used. The maximum is 255.
        DC    FL1'25,49,73,97,121'
*                        Tab positions within the print
*                          file. Each tab should have its
*                          value greater than the previous
*                          tab; otherwise, it is ignored.
*                          The first data field begins at
*                          the left margin (position 1)
*                          and thereafter each field
*                          begins at the next available
*                          tab position.
        END
```

MAXLEN

SUBROUTINE DESCRIPTION

Purpose: To obtain the maximum length for a PL/I varying string.

Location: *PL1LIB

Calling Sequence:

PL/I:
```
DECLARE MAXLEN ENTRY RETURNS(FIXED BIN(15));
len = MAXLEN(string);
```

Parameters:

len	is the maximum length that the variable may assume.
string	is the CHARACTER or BIT variable in question.

Description: Each PL/I character or bit string has two halfword lengths with which it is associated: the current length of the string which is returned by the PL/I built-in function LENGTH, and the maximum length which is returned by the function MAXLEN. For a varying-length string, the value returned by LENGTH is always less than or equal to the value returned by MAXLEN, while the values returned for fixed-length character strings will always be equal. Because MAXLEN is not a GENERIC function, any arrays passed as arguments to MAXLEN must include subscripts, i.e., MAXLEN(ARRAY(1,1)) and not MAXLEN(ARRAY).

Example:
```
MAIN: PROCEDURE OPTIONS(MAIN);
DECLARE MAXLEN ENTRY  RETURNS(FIXED BIN(15));
DECLARE BUFF1 CHAR(100) VARYING,
        BUFF2 CHAR(150),
        BUFF3 BIT(32) VARYING;
BUFF1 = '12345';
BUFF3 = '11111111'B;
PUT SKIP LIST(LENGTH(BUFF1),MAXLEN(BUFF1),
    LENGTH(BUFF2),MAXLEN(BUFF2),
    LENGTH(BUFF3),MAXLEN(BUFF3));
PUT SKIP;
END MAIN;
```

This example would print the values:

```
5  100  150  150  8  32
```

NEXTKEY, LASTKEY

SUBROUTINE DESCRIPTION

Purpose: To determine the key of the next record (NEXTKEY) or the end-of-file record (LASTKEY) in a sequential file.

Location: *PL1LIB

Calling Sequences:

PL/I:
```
DECLARE (NEXTKEY,LASTKEY) ENTRY
(FILE) RETURNS (CHARACTER (4));
```

Description: NEXTKEY: FILE is opened as:
output - returns the key of the next record to be written.
input or update - returns the key of the next record to be read.

LASTKEY: returns the key of the end-of-file record.

FILE: a PL/I file variable conforming to the following:

(1) a KEYED file of the consecutive organization, i.e., referring to an actual MTS sequential file.
(2) must be already opened by either an OPEN statement or by an appropriate I/O statement.

If an error occurs, the returned key will have the value of HIGH(4).

Example:
```
POINT=NEXTKEY(KEYED_FILE);
LOCATE BASED FILE(KEYED_FILE) KEYFROM(POINT);
BASED='ABC';
```

This example writes the character string ABC on the next record. BASED is a string variable declared with a PL/I BASED attribute.

RAND

SUBROUTINE DESCRIPTION

Purpose: To compute uniformly distributed random numbers between 0.0 and 1.0.

Location: *PL1LIB

Calling Sequences:

```
PL/I:     DECLARE RAND ENTRY (FIXED BINARY(31))
                  RETURNS (FLOAT BINARY);
```

Description: The argument I as in RAND (I) must be a variable initialized within the range 0 to 2^{31}-1 (2147483647). The value returned by RAND (I) is between 0.0 and 1.0. In addition, the variable is changed so that a different random number is generated on a subsequent call. If the argument I contains zero, a random number will be generated depending upon the time of day.

The algorithm is taken from "Coding the Lehmer Pseudo-Random Number Generator," Communications of the ACM, Volume 12, Number 2 (February 1969).

Example:

```
RANDOM:  PROC FLOAT BIN;
   DCL I FIXED BIN (31) STATIC
         INIT (524287),
      RAND ENTRY (FIXED BIN (31))
         RETURNS (FLOAT BIN);
      RETURN (RAND (I));
      END;
```

This example generates a random number using the number 524287 as the initial base.

SIGNOFF

SUBROUTINE DESCRIPTION

Purpose:	To sign the user off.
Location:	*PL1LIB

Calling Sequences:

PL/I: `DECLARE SIGNOFF ENTRY;`

Description: The subroutine closes all open files, if any, and then signs the user off.

Example:

```
IF BATCH THEN CALL SIGNOFF;
```

This example signs off the user if he is running in batch mode.

USERID

SUBROUTINE DESCRIPTION

Purpose: To obtain the current four-character Computing Center signon ID.

Location: *PL1LIB

Calling Sequences:

PL/I:
```
DECLARE USERID ENTRY RETURNS (CHARACTER(4));
```

Description: The subroutine returns the user signon ID.

Example:
```
PUT LIST (USERID);
```

This example prints out the user’s signon ID.

INTERLANGUAGE COMMUNICATION FACILITIES

The PL/I Optimizing Compiler provides the interlanguage facilities for users who want to call non-PL/I routines from their PL/I programs. (PL/I (F) users should use the subroutines such as PLCALL described at the end of this section.) This section describes only what users need to know in order to call non-PL/I routines. Full details are included in Chapter 19, "Interlanguage Communication Facilities," of the IBM publication, *OS PL/I Checkout and Optimizing Compilers: Language Reference Manual*, form GC33-0009. Also included in that chapter is a description of how FORTRAN subprograms can call PL/I external procedures.

PL/I, FORTRAN, and most MTS system subroutines use the OS S-type calling sequence (see MTS Volume 3, *System Subroutine Descriptions*, for the complete description of this calling sequence). Among other things, this calling sequence requires that arguments to be passed between routines be represented by a parameter list, a vector of addresses that point to the arguments. FORTRAN and most MTS system subroutines expect the addresses in the list to point to the actual data items being passed. Parameter lists produced by PL/I follow this convention for some types of arguments, but for some other types (strings, structures, arrays and areas) the parameter-list addresses point to a "locator",[1] which in turn points to the actual data items. The format of these locators and other details of PL/I data representation is given in the IBM publication, *OS PL/I Optimizing Compiler: Execution Logic*, form SC33-0025. If a routine whose entry points are declared with the ASSEMBLER (abbreviated as ASM), COBOL, or FORTRAN attributes, the data addresses of the arguments are always passed instead of locators.

The following example tests if a command starts with a dollar sign "$", representing an MTS command. If so, the MTS subroutine CMD is called.

```
     Declare CMD external entry
                 (character(*), fixed binary(31))
                 options (assembler, inter),
             COMMAND character(80);
        ...
     Get list (COMMAND);
     If substr(COMMAND,1,1)='$' then
        call CMD (COMMAND,80);
     Else ...;
```

----

[1]This "locator" was called a "dope vector" in PL/I (F) terminology.

Note that in the example above the option INTER was specified. This means that any interrupts not handled by the assembler routine CMD will be dealt with by the PL/I interrupt handler.

Another difference between the PL/I calling sequence and the calling sequence used by FORTRAN is the method by which function subprograms return results. FORTRAN functions return integer and logical results in general register 0, real results in floating-point register 0, complex results or extended real results in floating-point registers 0 and 2, and complex extended results in floating-point registers 0, 2, 4 and 6. PL/I takes a different approach by appending an additional address to the parameter list used to call procedures that may return results. This additional address points to either a location or to a locator pointing to a location where the result is to be returned. Here, too, the PL/I Optimizing compiler will also automatically return the result if the entry point being called has both the FORTRAN option and the RETURNS attribute, e.g.:

```
     Declare ARSIN external entry (float(6))
                   returns (decimal float(6))
                   options (fortran),
             DARCOS external entry (decimal float(16))
                   returns (decimal float(16)),
             (ARCSIN, ANGLE) decimal float(6),
             (ARCCOS, DANGLE) decimal float(16);
     ARCSIN = ARSIN (ANGLE);
     ARCCOS = DARCOS (DANGLE);
```

In addition, many non-PL/I routines set the return code in general register 15. Their entry points should then be declared with OPTIONS (RETCODE). The lower half of general register 15 is stored in a halfword, which is also set by means of the PLIRETC built-in subroutine called by any PL/I Optimizer procedure. This value can be interrogated using the PLIRETV built-in function.

Example:

```
     Declare SKIP external entry
             (fixed binary(31),  /* number of files */
              fixed binary(31),  /* number of records */
              fixed binary(31))  /* unit number */
|            options (assembler, inter, retcode),
             NFILES fixed binary(31),
             NRECORDS fixed binary(31),
             FDUB fixed binary(31),
             PLIRETV builtin;
        ...
     Call SKIP (NFILES, NRECORDS, FDUB);
     If PLIRETV¬=0 then do;
        ...
        End;
```

If PLIRETV is not explicitly declared, it should be specified with a null argument list, e.g., PLIRETV(), so that the subroutine is contextually declared with BUILTIN attribute.

The following example illustrates the use of the PL/I to non-PL/I interface routines. This procedure copies a file from the logical I/O unit 0 to the logical I/O unit 1 by calling the MTS READ and WRITE subroutines. The procedure terminates when a zero-length line or a nonzero return code from READ occurs.

```
MAIN: Procedure options(main);

      Declare BUFF character(200),
              LEN  fixed binary(15), /* Halfword */
              IMOD fixed binary(31)
                   initial (16385),  /* @SEQ@¬TRIM */
              OMOD fixed binary(31)
                   initial (16384),  /* ¬TRIM      */
              LINE fixed binary(31),
              (READ, WRITE) external entry
                   (character(*),      /* buffer */
                    fixed binary(15),  /* length */
                    fixed binary(31),  /* modifiers */
                    fixed binary(31),  /* line number */
                    fixed binary(31))  /* unit number */
                   options (retcode, assembler, inter),
              PLIRETV builtin;

      Do while ('1'b);         /* FOREVER */
         Call READ (BUFF, LEN, IMOD, LINE, 0);
         If PLIRETV ¬=0 | LEN=0 then leave;
         Call WRITE (BUFF, LEN, OMOD, LINE, 1);
      End;

      Stop;
      End MAIN;
```

If a parameter to a FORTRAN subprogram is an array, the parameter is passed directly only if the array is of one dimension, is connected[2], and is aligned as an identical FORTRAN array. If an array parameter does not meet the condition or has more than one dimension, the PL/I Optimizing compiler automatically creates a dummy array, into which it will map the array in the FORTRAN manner. Upon the termination of the

----

[2]"Connected" attribute means that all of the elements in the array are contiguous. If a structure is declared as:

```
1 ABC(5), 2 A, 2 B, 2 C
```

Three arrays ABC.A, ABC.B, ABC.C are not connected.

FORTRAN subprogram, it then remaps the dummy array back to the real array.

This example calls SORTIT with a two-dimensional array VECTOR. Here, VECTOR is first mapped into a dummy array before the call and then is remapped back from the dummy array after call.

```
     Declare SORTIT external entry
                    ((3,5) fixed binary(31))
                    options (fortran),
             VECTOR (3,5) fixed binary(31);
        ...
     Call SORTIT (VECTOR);
```

These two languages differ in the method used to map arrays into memory. In PL/I, the rightmost subscript varies most rapidly, while in FORTRAN, the leftmost subscript varies most rapidly. For example, the two-dimensional array in the above example is mapped in PL/I as follows:

```
     VECTOR: (1,1), (1,2), (1,3),
             (2,1), (2,2), (2,3),
             (3,1), (3,2), (3,3),
             (4,1), (4,2), (4,3),
             (5,1), (5,2), (5,3)
```

Before calling the routine SORTIT, a dummy array is created, and mapped from the VECTOR as follows:

```
     DUMMY:  (1,1), (2,1), (3,1), (4,1), (5,1),
             (1,2), (2,2), (3,2), (4,2), (5,2),
             (1,3), (2,3), (3,3), (4,3), (5,3)
```

And after the call to SORTIT, the contents of the dummy array are moved to the actual array VECTOR.

There are three options NOMAP, NOMAPIN, NOMAPOUT that will avoid mapping from/to a dummy array.

NOMAP will pass the array directly to a FORTRAN subroutine. To use this array, one will have to declare an array in FORTRAN as follows:

```
     INTEGER*4 VECTOR (3,5)
```

And the subscripts of the VECTOR should be <u>reversed</u> in the entire FORTRAN subroutine.

NOMAPIN option indicates the dummy array if created is not initialized from the argument. NOMAPOUT indicates the dummy array if created is not to be assigned back into the argument. These options can be used to save CPU time.

Generally, if NOMAP, NOMAPIN, or NOMAPOUT is specified, they will apply to all arguments of a FORTRAN routine being called. Users can also specify to which arguments these three options should apply. For example, an option NOMAP(ARG1,ARG3) applies NOMAP to the first and third arguments.

The following example illustrates the use of the NOMAPIN and NOMAPOUT options. Suppose we have a simple FORTRAN subroutine that will convert the integer array INPUT into the real array OUTPUT.

```
     Declare INOUT external entry
                  ((3,4) fixed binary(31),
                   (3,4) float (6))
                  options (FORTRAN nomapout(ARG1)
                           nomapin(ARG2)),
             INPUT (3,4) fixed binary(31),
             OUTPUT (3,4) float (6);
       ...
     Call INOUT (INPUT,OUTPUT);
```

The efficiency of the program is improved because a dummy array does not have to be copied from the OUTPUT array before the call to INOUT and another dummy array is not copied back into the INPUT array after the call.

Logical (LOGICAL*4) arguments may be passed to FORTRAN routines as BIT(32) variables. Any nonzero bit, usually

```
     '00000000000000000000000000000001'B)
```

indicates the value "true", and all zero bits indicate "false".

Logical (LOGICAL*1) arguments may be passed to FORTRAN routines as BIT(8) variables, that is, bit strings of length 8. '00000001'B represents "true", and '00000000'B represents "false".

Many MTS system subroutines that return results in the general or floating-point registers should be declared with option FORTRAN but not ASSEMBLER. The following example calls the system subroutine COST. Note that a FORTRAN function must have at least one argument.

```
     Declare  COST external entry (fixed binary(31))
                   returns (float decimal(16))
                   options (fortran),
              DOLLARS float decimal(16);
       ...
     DOLLARS = COST (0);
       ...
     End;
```

Below is an example that illustrates the use of varying character strings in a call to a system subroutine. Data representation of varying strings have first two bytes holding their current lengths, in bytes. This is then followed by the actual contents of the strings. Character strings are not available in FORTRAN.

```
     Declare DISMNT external entry
              (character(*) varying)
              options (assembler, inter),
             PAR character(32) varying
              initial ('*T*');
     Call DISMNT (PAR);
```

Many system subroutines do not conform to the S-type linkage. Here, a subroutine RCALL should be used to provide an R-type linkage to the system subroutine GETFD. Note that F2 and F1 are used as arguments to RCALL instead of 2 and 1 because these two constants, which are to be expected to have the attributes FIXED BINARY(31), have the attributes FIXED DECIMAL(1).

```
     Declare RCALL external entry
                   options (assembler, inter),
             GETFD external entry,
             FILNAM character(18),
             DUMMY fixed binary(31),
             F1 fixed binary(31) initial (1),
             F2 fixed binary(31) initial (2),
             FDUB  pointer;
|            FILPTR pointer;
|    FILPTR = ADDR(FILNAM);
|    Call RCALL (GETFD, F2, DUMMY, FILPTR, F1, FDUB);
```

There is another way a PL/I procedure can pass information to a FORTRAN subprogram besides using the arguments. External static variables in PL/I can also be considered as FORTRAN common labeled blocks. Here is an example:

```
     PL/I:    Declare ABC fixed binary(31) external static;
     FORTRAN: COMMON /ABC/ XYZ
              INTEGER*4 XYZ
```

The FORTRAN variable XYZ is equated to the PL/I variable ABC.

If the labeled common block has more than one FORTRAN variable, then the common block should be represented as a PL/I structure.

```
     FORTRAN:   COMMON  /ABC/ A, B, C, R(100)
                REAL    A, B, C
                INTEGER R
```

can be represented as:

```
     PL/I:      Declare 1 ABC external static,
                2 (A, B, C) decimal float(6),
                2 R(100)    fixed binary(31,0);
```

This method is only possible in PL/I Optimizer programs, since the PL/I (F) compiler actually inserts dope vectors for all external static variables except scalar arithmetic variables. All external static variables in PL/I Optimizer programs do not have any locators attached to them.

Table showing the data equivalents for FORTRAN and PL/I data.

FORTRAN	PL/I
INTEGER*2	REAL FIXED BINARY(15,0)
INTEGER*4	REAL FIXED BINARY(31,0)
REAL*4	REAL FLOAT DECIMAL(6)
REAL*8	REAL FLOAT DECIMAL(16)
REAL*16	REAL FLOAT DECIMAL(33)
COMPLEX*8	COMPLEX FLOAT DECIMAL(6)
COMPLEX*16	COMPLEX FLOAT DECIMAL(16)
COMPLEX*32	COMPLEX FLOAT DECIMAL(33)
LOGICAL*1	BIT(8) ALIGNED
LOGICAL*4	BIT(32) ALIGNED

Note that FORTRAN variables of type LOGICAL*4 are aligned on fullwords. If this alignment is necessary, then they should at least be equated to a PL/I equivalent using the DEFINED or BASED attributes.

CALLING PL/I PROCEDURES FROM FORTRAN SUBPROGRAMS

FORTRAN subprograms can also call external PL/I procedures. The external PL/I procedures should then have a FORTRAN option. If the procedures are being called as FORTRAN functions, then they should specify an appropriate RETURNS attribute. The following example shows a FORTRAN subprogram calling the PL/I external procedure SQRT.

```
     FORTRAN:   REAL*4 SQRT, ARG, RESULT
                RESULT = SQRT (ARG)
     PL/I:      SQRT:  Procedure (ARG) returns (float decimal(6))
                                 options (FORTRAN);
                       Declare (ARG, RESULT) float decimal(6);
                         ...
                       Return (RESULT);
                       End SQRT;
```

The external PL/I procedure can also have other options such as NOMAP, NOMAPIN, NOMAPOUT. Dummy arguments may be created so that FORTRAN arrays are mapped into PL/I arrays according to PL/I rules and then mapped back into the FORTRAN arrays upon the return from the external procedure. NOMAPIN inhibits the first mapping, NOMAPOUT inhibits the second mapping, and NOMAP passes FORTRAN arrays directly to PL/I arrays.

Examples:

```
     (1)  ABC:  Procedure (A, B, C)
                options (FORTRAN);
     (2)  ABC:  Procedure (A, B, C)
                options (FORTRAN NOMAP);
     (3)  ABC:  Procedure (A, B, C)
                options (FORTRAN NOMAPIN(A) NOMAPOUT(B) NOMAP(C));
```

The first example will map all three arguments, in case dummy arguments are required. In the second example, all arguments are passed directly. In the third example, the argument A will not be copied into a dummy array at start; the argument B will not be copied from a dummy array at return; and the argument C is passed directly.

<u>Example 1</u>:

In the following example, the two-dimensional array A is printed first by the FORTRAN program, then by a PL/I procedure, and then again by the FORTRAN main program. Note that the procedure TSUB creates a dummy array, remapping the FORTRAN array into the dummy array. At the end of execution, this dummy array is then remapped into the FORTRAN array.

```
#$list main.s
>     1     C  A simple FORTRAN main program that calls the PL/I
>     2     C  external procedure TSUB.
>     3     C
>     4           INTEGER A(3,2)
>     5           DATA K /0/
>     6           DO 10 J=1,2
>     7              DO 10 I=1,3
>     8                 K = K + 1
>     9                 A(I,J) = K
>    10        10 CONTINUE
>    11           WRITE (6,100) (( I,J,A(I,J), I=1,3), J=1,2)
>    12       100 FORMAT (' A(',I1,',',I1,')=',I1)
>    13           CALL TSUB (A)
>    14           WRITE (6,100) (( I,J,A(I,J), I=1,3), J=1,2)
>    15           STOP
>    16           END
#$list tsub.s
>     1     TSUB:  Procedure (A) options (FORTRAN);
>     2            Declare A(3,2) fixed binary(31),
>     3                    K      fixed binary(31) init(6);
>     4            Do J=1 to 2;
>     5               Do I=1 to 3;
>     6                  Put skip edit ('A(',I,',',J,')=',A(I,J))
>     7                                (a,f(1),a,f(1),a,f(1));
>     8                  A(I,J) = K;
>     9                  K = K - 1;
>    10               End /* Do I=1 to 3 */;
>    11            End /*Do J=1 to 2 */;
>    11.5          Put skip;
>    12            Return;
>    13            End TSUB;
```

```
 #run *ftn scards=main.s sprint=-print spunch=-obj
  No errors in MAIN
 #14:35:02  T=0.178  $0.08
 #run *pl1opt scards=tsub.s sprint=-print(last+1) spunch=-obj(last+1)
  PL/I OPTIMIZER V1 R3.1        TIME: 14.35.33  DATE: 14 JULY 82

   NO MESSAGES OF SEVERITY W AND ABOVE PRODUCED FOR THIS COMPILATION

   MESSAGES SUPPRESSED BY THE FLAG OPTION:  7 I.

- COMPILE TIME    0.00 MINS     SPILL FILE:     0 RECORDS, SIZE  4051
 #14:36:07  T=0.397  $0.20
 #run -obj+*pl1optlib
  A(1,1)=1
  A(2,1)=2
  A(3,1)=3
  A(1,2)=4
  A(2,2)=5
  A(3,2)=6

  A(1,1)=1
  A(2,1)=2
  A(3,1)=3
  A(1,2)=4
  A(2,2)=5
  A(3,2)=6

  A(1,1)=6
  A(2,1)=5
  A(3,1)=4
  A(1,2)=3
  A(2,2)=2
  A(3,2)=1

 #14:36:59  T=0.02  $0.01
```

Example 2:

The following program reads lines from SCARDS and breaks them up into words, placing the word count into the variable COUNT. Since FORTRAN does not allow character variables, PL/I character variables should be equated to the arguments. Either the DEFINED attribute or BASED variables can be used to equate the variables. The example illustrates the use of BASED variables.

```
#$list ftntest(1,999)
>  1        REAL*8 WORDS(4095), SENTEN(4096)
>  2        INTEGER*2 LEN, I
>  3        INTEGER*4 COUNT, QQSV, BREAK
>  4      1 CALL SCARDS (SENTEN, LEN, 0, QQSV, &999)
>  5        COUNT = BREAK (SENTEN, LEN, WORDS)
>  6        IF (COUNT.EQ.0) GO TO 1
>  7        COUNT = 3*COUNT
>  8        WRITE (6,9) (WORDS(I),I=1,COUNT)
>  9      9 FORMAT (1X,9A8)
> 10        GO TO 1
> 11    999 STOP
> 12        END
#$list pl1test(1,999)
>  1  BREAK: Procedure (SENTEN, LEN, WORDS#)
>  2                   returns (FIXED BINARY(31))
>  3                   options (FORTRAN NOMAP);
>  4
>  5         Declare (SENTEN(4096), WORDS#(4095)) float(16) connected,
>  6                 LEN                           fixed binary(15),
>  7                 SENTENCE                      character(32767)
>  8                                               based (SENPTR),
>  9                 WORDS(10920)                  character(24)
> 10                                               based (WORDPTR),
> 11                 (SENPTR, WORDPTR)             pointer,
> 12                 (COUNT, INDX, JNDX)           fixed binary(31),
> 13                 (LENGTH, INDEX, SUBSTR, ADDR) builtin;
> 14
> 15         COUNT = 0;
> 16         SENPTR = addr (SENTEN(1));
> 17         WORDPTR = addr (WORDS#(1));
> 18         INDX = 1;
> 19  LOOP:  Do until (INDX>LEN);
> 20            JNDX = index( substr(SENTENCE,INDX,LEN-INDX+1), ' ');
> 21            If JNDX=0 then JNDX = LEN+1;
> 22                      else JNDX = INDX+JNDX-1;
> 23            If INDX=JNDX then do;
> 24               If INDX>LEN then leave LOOP;
> 25               INDX = INDX+1;
> 26               Go to LOOP;
> 27            End;
```

```
            COUNT = COUNT+1;
            WORDS(COUNT) = substr(SENTENCE,INDX,JNDX-INDX);
            INDX = JNDX+1;
         End;

         Return (COUNT);

         End BREAK;
```

```
 #run ftntest(1000)+pl1test(1000)+*pl1optlib
  My name is Michigan Terminal System,
  My                     name                   is
  Michigan               Terminal               System,
  and I come from University of Michigan at Ann Arbor, Michigan.
  and                    I                      come
  from                   University             of
  Michigan               at                     Ann
  Arbor,                 Michigan.
 #14:03:36  T=0.022  $0.02
```

PL/I (F) INTERLANGUAGE SUBROUTINES

The following descriptions on PL/I subroutines apply only to PL/I (F) programs. These subroutines do not reside in *PL1OPTLIB because the ease of the interlanguage communication facilities renders them irrelevant.

PLCALL, PLCALLD, PLCALLE, PLCALLF

Purpose: To enable PL/I (F) users to call non-PL/I (e.g., FORTRAN and assembler) procedures requiring a standard S-type linkage.

Location: *PL1LIB

Calling Sequences:

```
PL/I (F): CALL PLCALL(fn,n,pl);

          DECLARE PLCALLD RETURNS(FLOAT(16));
          PLCALLD(fnd,n,pl);

          DECLARE PLCALLE RETURNS(FLOAT(6));
          PLCALLE(fne,n,pl);

          DECLARE PLCALLF RETURNS(FIXED BINARY(31));
          PLCALLF(fnf,n,pl);
```

Parameters:

fn	is a subroutine which has been declared to have the ENTRY attribute and which does not return a value.
fnd	is a function which has been declared to have the ENTRY attribute and which returns a double-precision floating-point value (REAL*8 in FORTRAN; long floating-point register 0 in assembly code).
fne	is a function which has been declared to have the ENTRY attribute and which returns a single-precision floating-point value (REAL*4 in FORTRAN; short floating-point register 0 in assembly code).
fnf	is a function which has been declared to have the ENTRY attribute and which returns an integer value (INTEGER*4 in FORTRAN; general register 0 in assembly code).
n	is a number with attributes FIXED BINARY(31) which is equal to the number of arguments being passed to fn, fnd, fne, or fnf. n may be 0.
pl	is a parameter list of the n arguments to be passed to fn, fnd, fne, or fnf in the order required by the subprogram. The arguments are separated by commas. If the argument is a string variable, array variable, or structure

variable, the name of the argument or a pointer to the argument may be used; for example, ARG or ADDR(ARG). Note that if the argument is an array variable, the reference passed will be to the location of the element having all zeros for subscripts (e.g., A(0,0)), even if that element does not exist. Therefore, it may be preferable to use a pointer to an element of the array instead of the array itself (e.g., ADDR(A(1,1)) instead of A). If the argument is a scalar variable, a pointer to the argument must be used; for example, ADDR(ARG). If the argument is a scalar constant, a pointer to the argument, which can be produced by the subroutine PL1ADR, must be used. The high-order bit of the last word in the parameter list passed to _fn_, _fnd_, _fne_, or _fnf_ is set to 1. If _n_=0, there is no parameter list and no comma after _n_.

Return Codes:

The return code placed in general register 15 by _fn_, _fnd_, _fne_, or _fnf_ may be tested using the subroutine PL1RC.

Description: PL/I program interrupt ON conditions are disabled on entry to the subprogram and reenabled on return to the calling program. The values of PLCALLD, PLCALLE, and PLCALLF are the values returned by _fnd_, _fne_, and _fnf_, respectively.

Examples:

```
DECLARE PLCALLE RETURNS(FLOAT(6));
DECLARE PLCALLD RETURNS(FLOAT(16));
DECLARE (ARSIN, DARCOS) ENTRY;
DECLARE (ARCSIN, ANGLE) FLOAT(6);
DECLARE (ARCCOS, DANGLE) FLOAT(16);
DECLARE F1 FIXED BINARY(31) INIT(1) STATIC;
ARCSIN=PLCALLE(ARSIN, F1, ADDR(ANGLE));
ARCCOS=PLCALLD(DARCOS, F1, ADDR(DANGLE));
```

The above example calls the FORTRAN library functions ARSIN and DARCOS.

```
/* PAR IS A STRUCTURE VARIABLE */
DECLARE DISMNT ENTRY;
DECLARE 1 PAR ALIGNED STATIC,
          2 LEN INIT(3),
          2 TAPE CHAR(3) INIT('*T*');
DECLARE F1 FIXED BINARY(31) INIT(1) STATIC;
CALL PLCALL(DISMNT, F1, PAR);
```

The above example calls the system subroutine DISMNT.

```
          /* FILNAM IS A STRING. F1, F2, AND FDUB ARE SCALAR */
          DECLARE RCALL ENTRY;
          DECLARE F6 FIXED BINARY(31) INIT(6) STATIC;
          DECLARE PL1ADR RETURNS(POINTER);
          DECLARE GETFD ENTRY;
          DECLARE F2 FIXED BINARY(31) INIT(2) STATIC;
          DECLARE FILNAM CHARACTER(18);
          DECLARE F1 FIXED BINARY(31) INIT(1) STATIC;
          DECLARE FDUB POINTER;
          CALL PLCALL(RCALL,F6,PL1ADR(GETFD),ADDR(F2),
               ADDR(DUMMY),PL1ADR(ADDR(FILNAM)),ADDR(F1),
               ADDR(FDUB));
```

The above example produces an R-type linkage to the system subroutine GETFD.

PL1ADR

Purpose: To obtain a pointer to PL/I scalar constants and variables.

Location: *PL1LIB

Calling Sequence:

```
PL/I (F): DECLARE PL1ADR ENTRY RETURNS(POINTER);
          PL1ADR(arg);
```

Parameter:

arg is any scalar constant or variable (not strings, arrays, or structures).

Description: The value of PL1ADR is the address of the argument. The primary purpose of this subroutine is to pass pointers for scalar constants to the subroutines PLCALL, PLCALLD, PLCALLE, and PLCALLF since a constant cannot be used as an argument to the PL/I function ADDR.

PL1RC

Purpose: To interrogate the return code passed back by the last call on PLCALL, PLCALLD, PLCALLE, or PLCALLF or set by IHESARC.

Location: *PL1LIB

Calling Sequence:

```
PL/I (F): DECLARE PL1RC ENTRY RETURNS(FIXED BINARY(31));
          PL1RC;
```

Description: The value of PL1RC is the contents of general register 15 when the procedure called using PLCALL, PLCALLD, PLCALLE, or PLCALLF returns, or is the value set by IHESARC, whichever is most recent. For FORTRAN subroutines, the value returned in general register 15 is four times the value of the integer after RETURN.

Example:

```
     DECLARE PL1RC ENTRY RETURNS(FIXED BINARY(31));
     IF PL1RC=4 THEN GO TO ERROR;
```

A branch is made to ERROR if the return code from the last call on PLCALL, PLCALLD, PLCALLE, or PLCALLF is 4.

PL1BEG, PL1END

Purpose: To establish and release explicitly the PL/I (F) environment.

Calling Sequence:

```
FORTRAN:  CALL PL1BEG
          CALL PL1END

Assembly: CALL PL1BEG
          CALL PL1END
```

Comments: Calling PL1BEG explicitly establishes the PL/I (F) environment that is required if non-PL/I routines are to call PL/I (F) procedures. The PL/I environment remains established until the PL1END subroutine is called or the program is unloaded. Calling PL1BEG explicitly is not usually necessary, because it will be called implicitly by the other interface routines if the PL/I (F) environment has not been established. Calls to PL1BEG are ignored if the PL/I (F) environment has been established.

PL1END should be called to close PL/I files and to release the PL/I (F) environment after all PL/I procedures have been called. Calls to PL1END are ignored if the PL/I (F) environment has not been established.

CDFCN, CPXFCN, IPLFCN, I2FCN, LGLFCN, LG1FCN, PLDFCN, PL1FCN, PL1SUB

Purpose: To call PL/I procedures from non-PL/I routines, establishing the PL/I (F) environment if necessary.

Calling Sequence:

```
FORTRAN:  CALL PL1SUB(entry,arg1,type1,...,argn,typen)
          real  =PL1FCN(entry,arg1,type1,...,argn,typen)
          dreal =PLDFCN(entry,arg1,type1,...,argn,typen)
          int   =IPLFCN(entry,arg1,type1,...,argn,typen)
          sint  =I2FCN(entry,arg1,type1,...,argn,typen)
          cmplx =CPXFCN(entry,arg1,type1,...,argn,typen)
          dcmplx=CDFCN(entry,arg1,type1,...,argn,typen)
          logel =LGLFCN(entry,arg1,type1,...,argn,typen)
          slogel=LG1FCN(entry,arg1,type1,...,argn,typen)
```

Assembly:

The VL option of the CALL macro should be used to set the high-order bit of the final address in the variable-length parameter list.

Parameters:

entry	The entry point of the PL/I (F) external procedure that is to be called. FORTRAN users must define this entry point name using the EXTERNAL statement. If it is necessary to call a PL/I procedure with a name that is more than 6 characters long, a "dummy" name should be used and ALI loader records added to equate the names (see MTS Volume 5).
arg1...argn	The arguments to be passed, if any. Arrays are passed by omitting subscripts or by using all ones for subscripts.
type1...typen	A fullword integer (INTEGER*4) or an array of integer numbers describing the type of the previous arguments follows:

0 LOGICAL*4 or BIT(32) where .TRUE. is represented as '00000000000000000000000000000001'B and .FALSE. is represented as (32)'0'B.

1	LOGICAL*1 or BIT(8) where .TRUE. is represented as '00000001'B and .FALSE. is represented as '00000000'B.
2	INTEGER*4 or FIXED BINARY(31).
3	INTEGER*2 or FIXED BINARY(15).
4	REAL*4 or FLOAT DECIMAL(6) REAL.
5	REAL*8 or FLOAT DECIMAL(16) REAL.
6	COMPLEX*8 or FLOAT DECIMAL(6) COMPLEX.
7	COMPLEX*16 or FLOAT DECIMAL(16) COMPLEX.
8	Single character or CHARACTER(1) FIXED. In standard FORTRAN, there is no true character data type, but Hollerith literals and characters stored left-justified in other data types can be passed using this type.
9	Character string or CHARACTER(n) FIXED. This data type, like type 8, does not actually exist in FORTRAN, but can be passed as a Hollerith literal or LOGICAL*1 array. In this case, "type" must be a two-element array with the first element set to 9 (TYPE(1)=9) and the second element set to the string length "n", (TYPE(2)=n).
128	A LOGICAL*4 array.
129	A LOGICAL*1 array.
130	An INTEGER*4 array.
131	An INTEGER*2 array.
132	A REAL*4 array.
133	A REAL*8 array.
134	A COMPLEX*8 array.
135	A COMPLEX*16 array.
136	An array of single characters.
137	An array of character strings.

For types 128 through 136, the variable type must be specified as an array with the first element set to the type code (128 to 136), the second element set to a number between 1 and 7 representing the number of dimensions in the array being passed, and the third through the tenth elements set to the upper bound associated with each dimension of the array. The lower bound of each dimension is assumed to be 1.

For type 137, the variable type is also specified as an array with the first element set to 137 (TYPE(1)=137), the second element set to the length of the strings, the third element is set to the number of dimensions in the array, and the fourth through possibly the eleventh elements set to the upper bound of each dimension of the array.

Values Returned:

<u>real</u>	A REAL*4 or FLOAT DECIMAL(6) REAL value.
<u>dreal</u>	A DOUBLE PRECISION (REAL*8) or FLOAT DECIMAL(16) REAL value.
<u>int</u>	A INTEGER (INTEGER*4) or FIXED BINARY(31) value.
<u>sint</u>	An INTEGER*2 or FIXED BINARY(15) value.
<u>cmplx</u>	A COMPLEX (COMPLEX*8) or FLOAT DECIMAL(6) COMPLEX value.
<u>dcmplx</u>	A DOUBLE PRECISION COMPLEX (COMPLEX(16)) or FLOAT DECIMAL(16) COMPLEX value.
<u>logel</u>	A LOGICAL (LOGICAL*4) or BIT(32) value.
<u>slogel</u>	A LOGICAL*1 or BIT(8) value.

Comments: Assembler users should note that INTEGER*4, INTEGER*2, and LOGICAL*4 values are returned in general register 0. LOGICAL*1 values are returned right-justified in general register 0. REAL*4, REAL*8, and the real part of COMPLEX*16 values are returned in floating-point register 0. The imaginary parts of COMPLEX*8 and COMPLEX*16 values are returned in floating-point register 2.

As arrays are passed to a PL/I (F) procedure, the necessary transformations on the PL/I (F) dope vector are accomplished so that the array will be referenced in column-major order rather than the row-major order normally standard in PL/I.

IPL1RC

Purpose: To return the return code from a PL/I (F) procedure which was called while using the non-PL/I (F) to PL/I interface routines.

Calling Sequence:

FORTRAN: int=IPL1RC(0)

Assembly: CALL IPL1RC

Value Returned:

int	The fullword integer (INTEGER*4) return code for the last PL/I (F) procedure called using the interface. Assembler users will find this value in general register 0.

CALLING SYSTEM SUBROUTINES FROM PL/I (F)

PL/I(F) programs may call both S-type and R-type system subroutines by using the special interface routines PLCALL, PLCALLD, PLCALLE, and PLCALLF.

System subroutines that are called using the standard S-type conventions are called by using the PLCALL routine:

```
     DECLARE subr   EXTERNAL ENTRY,
             PLCALL EXTERNAL ENTRY;
     CALL PLCALL(subr,fn,p1,p2,...,pn);
```

where "subr" is the name of the system subroutine being called, "fn" is the number of parameters to the subroutine, and "pn" are the individual parameters. If the subroutine returns a double-precision (8-byte) value in floating-point register 0, it is called as a function by using PLCALLD:

```
     DECLARE subr    EXTERNAL ENTRY,
             PLCALLD EXTERNAL ENTRY RETURNS (REAL FLOAT(16)),
             value   REAL FLOAT(16);
     value = PLCALLD(subr,fn,p1,p2,...,pn);
```

where "value" is the double-precision value returned. If the subroutine returns a single-precision (4-byte) value in floating-point register 0, it is called as a function by using PLCALLE:

```
     DECLARE subr    EXTERNAL ENTRY,
             PLCALLE EXTERNAL ENTRY RETURNS (REAL FLOAT(6)),
             value   REAL FLOAT(6);
     value = PLCALLE(subr,fn,p1,p2,...,pn);
```

where "value" is the single-precision value returned. If the subroutine returns an integer (4-byte) value in general register 0, it is called as a function by using PLCALLF:

```
     DECLARE subr    EXTERNAL ENTRY,
             PLCALLF EXTERNAL ENTRY RETURNS (FIXED BINARY(31)),
             value   FIXED BINARY(31);
     value = PLCALLF(subr,fn,p1,p2,...,pn);
```

where "value" is the integer value returned.

The equivalences of data types for PL/I(F) are given below:

Data Type	PL/I(F) Declaration
Fullword integer	FIXED BINARY(31)
Halfword integer	FIXED BINARY(15)
One-Byte integer	BIT(8)
8-Byte integer	FIXED BINARY(31) array of two elements
Fullword real	REAL FLOAT(16)
Doubleword real	REAL FLOAT(6)
Fullword logical	FIXED BINARY(31) or BIT(32) (0 is FALSE, 1 is TRUE)
One-Byte logical	BIT(8)
Character string	CHARACTER(n) ("n" is the length of the string)
Array	Array of appropriate data type
Region	Structure of appropriate data types

Normally, when calling PL/I(F) procedures from within PL/I(F), the value of the parameter is passed instead of the address of the parameter (except for character strings, arrays, and structures, in which case the address is passed). All S-type system subroutines and all R-type subroutines that are called using the RCALL subroutine expect the address to be passed in the parameter list instead of the value. Therefore, the PL/I ADDR routine (a built-in function) or the PL1ADR routine (a PL/I(F) library function) must be used to generate the address. ADDR (which is the simplest and most direct) may be used only for variables. It returns a pointer value and is called in the form

```
     ptr = ADDR(pn);
```

PL1ADR must be used for scalar constants such as subroutine names and pointers to addresses. It returns a pointer value and is called in the form

```
     DECLARE PL1ADR EXTERNAL ENTRY RETURNS (POINTER);
     ptr = PL1ADR(pn);
```

The following programs illustrate PL/I(F) calls to system subroutines.

```
     TEST: PROCEDURE OPTIONS(MAIN);

       DECLARE MTS    EXTERNAL ENTRY,
               PLCALL EXTERNAL ENTRY,
               F0     FIXED BINARY(31) INITIAL(0);

       CALL PLCALL(MTS,F0);
       RETURN;

     END TEST;
```

The above example calls the MTS subroutine, which requires no parameters. Since it is a standard S-type subroutine, PLCALL is used.

```
     TEST: PROCEDURE OPTIONS(MAIN);

       DECLARE CMD    EXTERNAL ENTRY,
               PLCALL EXTERNAL ENTRY,
               STRING CHARACTER(255),
               LENGTH FIXED BINARY(15),
               F2     FIXED BINARY(31) INITIAL(2);

       STRING = '$Display Timespelledout';
       LENGTH = 23;
       CALL PLCALL(CMD,F2,STRING,ADDR(LENGTH));
       RETURN;

     END TEST;
```

The above example calls the CMD subroutine to execute a $DISPLAY command. The subroutine requires two parameters, the first being a character string giving the command string and the second being a halfword integer command length (CMD also allows a fullword integer to be used). The ADDR function is used to generate the address of the LENGTH parameter in the parameter list. Since STRING is a character string, the address is automatically generated in the parameter list by PL/I(F).

```
     TEST: PROCEDURE OPTIONS(MAIN);

       DECLARE GUINFO EXTERNAL ENTRY,
               PLCALL EXTERNAL ENTRY,
               ITEMNO FIXED BINARY(31) INITIAL(2),
               USERID CHARACTER(4),
               F2     FIXED BINARY(31) INITIAL(2);

       CALL PLCALL(GUINFO,F2,ADDR(ITEMNO),USERID);
       PUT EDIT('User ID = ',USERID) (A,A(4));
       RETURN;

     END TEST;
```

The above example calls the GUINFO subroutine to obtain the current user ID. The subroutine also requires two parameters, the first of which is a fullword integer and the second a character string. Again, the ADDR function must be used to generate the address of the ITEMNO parameter.

```
     TEST: PROCEDURE OPTIONS(MAIN);

       DECLARE CHKFILE EXTERNAL ENTRY,
               PLCALLF EXTERNAL ENTRY RETURNS (FIXED BINARY(31)),
               PL1RC   EXTERNAL ENTRY RETURNS (FIXED BINARY(31)),
               FILENAM CHARACTER(18),
               ACCESS  FIXED BINARY(31),
               F1      FIXED BINARY(31) INITIAL(1);

       FILENAM = 'WABC:DATA1 ';
       ACCESS  = PLCALLF(CHKFILE,F1,FILENAM);
       IF PL1RC > 0 THEN
          PUT EDIT('File does not exist') (A);
       ELSE
          PUT EDIT('Access = ',ACCESS) (A,F(2));
       RETURN;

     END TEST;
```

The above example calls the CHKFILE subroutine to determine the program's access to the file WABC:DATA. Since the access is returned in general register 0, the subroutine must be called using the PLCALLF routine.

Return codes from system subroutines are obtained by calling the PL1RC routine (a PL/I(F) library function). PL1RC returns the return code from the subroutine and is called in the form

```
     DECLARE PL1RC EXTERNAL ENTRY RETURNS (FIXED BINARY(31));
     rcode = PL1RC;
```

PL1RC may be used to obtain the return code from subroutines called by PLCALL, PLCALLD, PLCALLE, and PLCALLF.

```
     TEST: PROCEDURE OPTIONS(MAIN);

       DECLARE GFINFO  EXTERNAL ENTRY,
               PLCALL  EXTERNAL ENTRY,
               PL1RC   EXTERNAL ENTRY RETURNS (FIXED BINARY(31)),
               UNIT    CHARACTER(8),
               FLAG    FIXED BINARY(31) INITIAL(2),
               ERCODE  FIXED BINARY(31),
               ERRMSG  CHARACTER(80),
               F8      FIXED BINARY(31) INITIAL(8);

       DECLARE 1 RTN,
                 2 CHR    CHARACTER(20),
                 2 INT    FIXED BINARY(31);
       DECLARE 1 CINFO,
                 2 CIAL   FIXED BINARY(31),
                 2 CIRL   FIXED BINARY(31),
                 2 CIONID CHARACTER(4),
                 2 CIVOL  CHARACTER(8),
                 2 CIUC   FIXED BINARY(31),
                 2 CILRD  FIXED BINARY(31),
                 2 CICD   FIXED BINARY(31),
                 2 CIFO   FIXED BINARY(31),
                 2 CIDT   FIXED BINARY(31),
                 2 CIFLG  FIXED BINARY(31),
                 2 CILCD  FIXED BINARY(31),
                 2 CIPKEY CHARACTER(16),
                 2 CILCCT (2) FIXED BINARY(31),
                 2 CILNCD FIXED BINARY(31),
                 2 CILNCT (2) FIXED BINARY(31),
                 2 CICDT  (2) FIXED BINARY(31),
                 2 CILRDT (2) FIXED BINARY(31);
       DECLARE 1 FINFO,
                 2 FIAL   FIXED BINARY(31) INITIAL(0);
       DECLARE 1 SINFO,
                 2 SIAL   FIXED BINARY(31) INITIAL(0);

       UNIT    = 'SCARDS  ';
       RTN.INT = 0;
       CINFO.CIAL = 25;
       CALL PLCALL(GFINFO,F8,UNIT,RTN,ADDR(FLAG),CINFO,FINFO,SINFO,
                   ADDR(ERCODE),ERRMSG);
       IF PL1RC = 4 THEN
          PUT EDIT(ERCODE,ERRMSG) (F(2),X(2),A(80));
       ELSE IF PL1RC > 4 THEN
          PUT EDIT('Error return from GFINFO subroutine') (A);
       ELSE
          PUT EDIT(RTN.CHR,'  Owner = ',CINFO.CIONID) (A(20),A,A(4));
       RETURN;

     END TEST;
```

The above example calls the GFINFO subroutine to obtain catalog information about the file attached to the logical I/O unit SCARDS. The structure CINFO is passed to the subroutine; upon return, the subroutine will insert the catalog information into this region. By using a structure, a packed region of varying data types can be defined. Again, the PL1RC routine is called to obtain the return code from GFINFO.

```
     TEST: PROCEDURE OPTIONS(MAIN);

       DECLARE CHKACC  EXTERNAL ENTRY,
               RENAME  EXTERNAL ENTRY,
               PLCALL  EXTERNAL ENTRY,
               PLCALLF EXTERNAL ENTRY RETURNS (FIXED BINARY(31)),
               PL1RC   EXTERNAL ENTRY RETURNS (FIXED BINARY(31)),
               UNSPEC  BUILTIN,
               ACCESS  FIXED BINARY(31),
               OLDNAM  CHARACTER(18),
               NEWNAM  CHARACTER(18),
               MASK    FIXED BINARY(31) INITIAL(16),
               F2      FIXED BINARY(31) INITIAL(2);

       DECLARE 1 TRIPLE,
                 2 CCID CHARACTER(4),
                 2 PROJ CHARACTER(4),
                 2 PKEY CHARACTER(18);

       OLDNAM = 'DATA1 ';
       NEWNAM = 'NEWDATA1 ';
       TRIPLE.CCID = 'WABC';
       TRIPLE.PROJ = 'WXYZ';
       TRIPLE.PKEY = '*EXEC ';
       ACCESS = PLCALLF(CHKACC,F2,OLDNAM,TRIPLE);
       IF PL1RC > 0 THEN
          PUT EDIT('File does not exist') (A);
       ELSE IF (UNSPEC(ACCESS) & UNSPEC(MASK)) ¬= UNSPEC(MASK) THEN
          PUT EDIT('Rename access not allowed') (A);
       ELSE DO;
          CALL PLCALL(RENAME,F2,OLDNAM,NEWNAM);
          IF PL1RC > 0 THEN
             PUT EDIT('Error return from RENAME subroutine') (A);
          ELSE
             PUT EDIT('File successfully renamed') (A);
          END;
       RETURN;

     END TEST;
```

In the above example, the CHKACC subroutine is called via PLCALLF to obtain the access of the file WABC:DATA1. If rename access is allowed, then the RENAME subroutine is called using PLCALL to rename the file to NEWDATA.

In PL/I(F), there are no special problems connected with calling subroutines that have a variable number of parameters, since PLCALL and friends can readily accept a variable-length parameter list. In the example below, the COMMAND subroutine is called once with three parameters and again with five parameters.

```
     TEST: PROCEDURE OPTIONS(MAIN);

        DECLARE COMMAND EXTERNAL ENTRY,
                PLCALL  EXTERNAL ENTRY,
                CMDTEXT CHARACTER(255),
                CMDLEN  FIXED BINARY(31),
                SWS     FIXED BINARY(31) INITIAL(0),
                SUMMARY FIXED BINARY(31),
                ERCODE  FIXED BINARY(31),
                F3      FIXED BINARY(31) INITIAL(3),
                F5      FIXED BINARY(31) INITIAL(5);

        CMDTEXT = '$Display Timespelledout ';
        CMDLEN  = LENGTH(CMDTEXT);
        CALL PLCALL(COMMAND,F3,CMDTEXT,ADDR(CMDLEN),ADDR(SWS));
        CMDTEXT = '$Display Timemisspelledout ';
        CMDLEN  = LENGTH(CMDTEXT);
        CALL PLCALL(COMMAND,F5,CMDTEXT,ADDR(CMDLEN),ADDR(SWS),
                    ADDR(SUMMARY),ADDR(ERCODE));
        IF SUMMARY > 0 THEN
           PUT EDIT('Command Error Code = ',ERCODE) (A,F(3));
        RETURN;

      END TEST;
```

R-Type Subroutines

R-type subroutines can be called from PL/I(F) by using the the PLCALL routine to call the RCALL subroutine. The RCALL subroutine sets up a call to an R-type subroutine by inserting the parameters into the proper registers for the actual call to the system subroutine. The PL1ADR routine must be used to obtain the address of system subroutine name as required for the RCALL subroutine.

The call to the RCALL subroutine is made in the following manner:

```
     DECLARE RCALL  EXTERNAL ENTRY,
             subr   EXTERNAL ENTRY,
             PLCALL EXTERNAL ENTRY,
             PL1ADR EXTERNAL ENTRY RETURNS (POINTER);
     CALL PLCALL(RCALL,fn,PL1ADR(subr),r1,p1,...,r2,p2,...);
```

where "r1" is the number of registers to be set up on the call to "subr" and "p1,..." are the values to be inserted into the registers beginning with general register 0; "r2" is the number of registers to contain return values from "subr" and "p2,..." are the variables that will contain the returned values starting with general register 0.

```
     TEST: PROCEDURE OPTIONS(MAIN);

       DECLARE RCALL   EXTERNAL ENTRY,
               GETFD   EXTERNAL ENTRY,
               RENUMB  EXTERNAL ENTRY,
               PL1RC   EXTERNAL ENTRY RETURNS (FIXED BINARY(31)),
               PL1ADR  EXTERNAL ENTRY RETURNS (POINTER),
               FIRST   FIXED BINARY(31) INITIAL(1000),
               LAST    FIXED BINARY(31) INITIAL(100000000),
               BEG     FIXED BINARY(31) INITIAL(1000),
               INC     FIXED BINARY(31) INITIAL(1000),
               FILENAM CHARACTER(18),
               FDUB    POINTER,
               DUMY    FIXED BINARY(31),
               F1      FIXED BINARY(31) INITIAL(1),
               F2      FIXED BINARY(31) INITIAL(2),
               F5      FIXED BINARY(31) INITIAL(5),
               F6      FIXED BINARY(31) INITIAL(6);

       FILENAM = 'DATA1 ';                  /* SET FILE NAME */
       CALL PLCALL(RCALL,F6,PL1ADR(GETFD),ADDR(F2),ADDR(DUMY),
                   PL1ADR(ADDR(FILENAM)),ADDR(F1),ADDR(FDUB));
       CALL PLCALL(RENUMB,F5,ADDR(FDUB),ADDR(FIRST),ADDR(LAST),
                   ADDR(BEG),ADDR(INC));   /* RENUMBER FILE */
       IF PL1RC > 0 THEN                    /* TEST RETURN CODE */
          PUT EDIT('Error return from RENUMB subroutine') (A);
       ELSE
          PUT EDIT('File successfully renumbered') (A);
       RETURN;

     END TEST;
```

In the above example, the GETFD subroutine is called to obtain a FDUB-pointer for the file DATA1; the FDUB-pointer is then passed on to the RENUMB subroutine to renumber the file. The GETFD subroutine requires that register 1 contain the address of the name of the subroutine as returned by the PL1ADR routine. The register count is 2, since RCALL initializes registers beginning with register 0 (in this case, register 0 is called with a dummy argument of zero). Upon return, GETFD returns the FDUB-pointer in register 0. Hence the register count is 1 and the FDUB-pointer is inserted in the variable FDUB.

```
  TEST: PROCEDURE OPTIONS(MAIN);

    DECLARE RCALL    EXTERNAL ENTRY,
            GDINFO   EXTERNAL ENTRY,
            FREESPAC EXTERNAL ENTRY,
            PLCALL   EXTERNAL ENTRY,
            PL1RC    EXTERNAL ENTRY RETURNS (FIXED BINARY(31)),
            PL1ADR   EXTERNAL ENTRY RETURNS (POINTER),
            UNIT1    CHARACTER(4),
            UNIT2    CHARACTER(4),
            DUMY     FIXED BINARY(31),
            F0       FIXED BINARY(31) INITIAL(0),
            F2       FIXED BINARY(31) INITIAL(2),
            F5       FIXED BINARY(31) INITIAL(5),
            F7       FIXED BINARY(31) INITIAL(7),
            INFOPTR  POINTER;

    DECLARE 1 INFO BASED (INFOPTR),
              2 FDUB   FIXED BINARY(31),
              2 DEVTYP CHARACTER(4),
              2 INLEN  FIXED BINARY(15),
              2 OUTLEN FIXED BINARY(15),
              2 USE    CHARACTER(1),
                 2 DEVICE CHARACTER(1),
                 2 SWS1   CHARACTER(1),
                 2 SWS2   CHARACTER(1),
                 2 MODS   FIXED BINARY(31),
                 2 BEGLNR FIXED BINARY(31),
                 2 PRVLNR FIXED BINARY(31),
                 2 ENDLNR FIXED BINARY(31),
                 2 INCLNR FIXED BINARY(31),
                 2 NAMPTR FIXED BINARY(31),
                 2 MSGPTR FIXED BINARY(31),
                 2 IOSAVE FIXED BINARY(31),
                 2 LASTRC FIXED BINARY(31),
                 2 REGLEN FIXED BINARY(15),
                 2 WIDTH  FIXED BINARY(15),
                 2 MACID  FIXED BINARY(31);

       UNIT1 = 'SCAR';                /* SET I/O UNIT NAME */
       UNIT2 = 'DS  ';
       CALL PLCALL(RCALL,F7,PL1ADR(GDINFO),ADDR(F2),UNIT1,UNIT2,
                   ADDR(F2),ADDR(DUMY),ADDR(INFOPTR));
       IF PL1RC > 0 THEN
          PUT EDIT('Error return from GDINFO subroutine') (A);
       ELSE DO;
          PUT EDIT('Type = ',INFO.DEVTYP) (A,A(4));
          CALL PLCALL(RCALL,F5,PL1ADR(FREESPAC),ADDR(F2),ADDR(F0),
                      ADDR(INFOPTR),ADDR(F0);  /* RELEASE STORAGE */
          END;                                 /* FROM GDINFO     */
       RETURN;

     END TEST;
```

In this example, the GDINFO subroutine is called by the RCALL subroutine. Two registers (general registers 0 and 1) are set up on the call to contain the eight-character logical I/O unit name. Two registers are also set up for the return. Register 1 will contain the address of the GDINFO information region; register 0 is not used, hence a dummy argument must be inserted into the RCALL parameter list. This example also illustrates a case where a system subroutine returns a pointer to an area of storage acquired by the subroutine itself. Hence the variable INFOPTR, which upon return will contain the address of the acquired storage, must be declared as a pointer variable. The statement following the subroutine call is then used to copy the contents of that storage into the BASED structure INFO so that the individual items of GDINFO information can be accessed by the program. At the end of the program, the FREESPAC subroutine is called by the RCALL subroutine to release the storage acquired by the GDINFO subroutine. Note that the GDINF alternative entry to the GDINFO subroutine also could have been called; this would circumvent the problem of using pointer variables and based structures in the PL/I program and having to call FREESPAC to release the GDINFO area.

Special Cases

Several system subroutines cannot be directly called by PL/I programs because they require nonstandard calls for exit routines (e.g., ATTNTRP or TIMNTRP). However, most of these subroutines have S-type alternatives that perform similar functions. Some of the more common alternative entries (or subroutines) are given in the table below.

System Subroutine	Alternative Entry
ATTNTRP	ATNTRP
LINK	LINKF
LOAD	LOADF
REWIND#	REWIND
TIMNTRP	TICALL
UNLOAD	UNLDF
XCTL	XCTLF

Further information about calling external subroutines from PL/I(F) is given in the section "Interlanguage Communication Facilities" in this volume.

CALLING SYSTEM SUBROUTINES FROM *PL1OPT

PL/I programs compiled by the *PL1OPT Optimizing Compiler may call both S-type and R-type system subroutines. The calling conventions are more direct than with PL/I (F), since the special PL/I (F) interface routines PLCALL, PLCALLD, PLCALLE, and PLCALLF are not used.

System subroutines that are called using the standard S-type conventions are simply declared as external entry points:

```
     DECLARE subr EXTERNAL ENTRY (par1,par2,...,parn)
                  OPTIONS (ASSEMBLER,INTER,RETCODE);
     CALL subr(p1,p2,...,pn);
```

where "subr" is the name of the system subroutine being called, "parn" are the data type declarations for the parameters, and "pn" are the individual parameters. If the subroutine returns a value in general register 0 or in floating-point register 0, it is called as a function and declared as follows:

```
     DECLARE subr EXTERNAL ENTRY (par1,par2,...,parn)
                  RETURNS (type)
                  OPTIONS (FORTRAN,INTER,RETCODE);
     value = subr(p1,p2,...,pn);
```

where "type" is the data type of the value to be returned (FIXED BINARY(31), REAL FLOAT(6), or REAL FLOAT(16)) and "value" is the function value returned.

The equivalences of data types for PL/I are given below:

Data Type	PL/I Declaration
Fullword integer	FIXED BINARY(31)
Halfword integer	FIXED BINARY(15)
One-Byte integer	BIT(8)
8-Byte integer	FIXED BINARY(31) array of two elements
Fullword real	REAL FLOAT(16)
Doubleword real	REAL FLOAT(6)
Fullword logical	FIXED BINARY(31) or BIT(32) (0 is FALSE, 1 is TRUE)

One-Byte logical	BIT(8)
Character string	CHARACTER(n) ("n" is the length of the string)
Character string[1]	CHARACTER(n) VARYING ("n" is the length of the string)
Array	Array of appropriate data type
Region	Structure of appropriate data types

[1]Halfword length followed by character string.

Normally, when calling PL/I procedures from within PL/I, the address of the parameter is passed instead of the value of the parameter as opposed to PL/I (F) which passes the value of the parameter instead; hence the ADDR and PL1ADR routines are not used.

The following programs illustrate PL/I calls to system subroutines.

```
     Test: Procedure Options(Main);

       Declare MTS External Entry
                   Options (Assembler,Inter);

       Call MTS;
       Return;

     End Test;
```

The above example calls the MTS subroutine which requires no parameters.

```
     Test: Procedure Options(Main);

       Declare CMD External Entry (Character(*), Fixed Binary(15))
                   Options (Assembler,Inter),
               String Character(255),
               Length Fixed Binary(15);

       String = '$Display Timespelledout';
       Length = 23;
       Call CMD(String,Length);
       Return;

     End Test;
```

The above example calls the CMD subroutine to execute a $DISPLAY command. The subroutine requires two parameters: the first is a character string giving the command string and the second is a halfword integer command length (CMD also allows a fullword integer to be used).

```
     Test: Procedure Options(Main);

       Declare GUINFO External Entry (Fixed Binary(31), Character(4))
                      Options (Assembler,Inter,Retcode),
               Itemno Fixed Binary(31) Initial(2),
               Userid Character(4);

       Call GUINFO(Itemno,Userid);
       Put Edit('User ID = ',Userid) (A,A(4));
       Return;

     End Test;
```

The above example calls the GUINFO subroutine to obtain the current user ID. The subroutine also requires two parameters, the first of which is a fullword integer and the second a character string.

```
     Test: Procedure Options(Main);

       Declare CHKFILE External Entry (Character(*))
                       Returns (Fixed Binary(31))
                       Options (Fortran,Inter,Retcode),
               PLIRETV Builtin,
               Filenam Character(18),
               Access  Fixed Binary(31);

       Filenam = 'WABC:DATA1 ';
       Access  = CHKFILE(Filenam);
       If PLIRETV > 0 Then
          Put Edit('File does not exist') (A);
       Else
          Put Edit('Access = ',Access) (A,F(2));
       Return;

     End Test;
```

The above example calls the CHKFILE subroutine to determine the program's access to the file WABC:DATA. Since the access is returned in general register 0, the subroutine must be called as a Fixed Binary function.

Return codes from system subroutines are obtained by calling the PLIRETV routine (a built-in function). PLIRETV returns the return code from the subroutine and is called in the form

```
     DECLARE PLIRETV BUILTIN;
     rcode = PLIRETV;
```

PLIRETV is similar to the PLIRETC built-in function which is used to obtain return codes from PL/I procedures.

```
Test: Procedure Options(Main);

  Declare GFINFO  External Entry
                    (Character(8), *, Fixed Binary(31), *, *, *,
                     Fixed Binary(31), Character(80))
                  Options (Assembler,Inter,Retcode),
          PLIRETV Builtin,
          STORAGE Builtin,
          Unit    Character(8),
          Flag    Fixed Binary(31) Initial(2),
          Ercode  Fixed Binary(31),
          Errmsg  Character(80);

  Declare 1 Rtn,
            2 Chr    Character(20),
            2 Int    Fixed Binary(31);
  Declare 1 Cinfo,
            2 Cial   Fixed Binary(31),
            2 Cirl   Fixed Binary(31),
            2 Cionid Character(4),
            2 Civol  Character(8),
            2 Ciuc   Fixed Binary(31),
            2 Cilrd  Fixed Binary(31),
            2 Cicd   Fixed Binary(31),
            2 Cifo   Fixed Binary(31),
            2 Cidt   Fixed Binary(31),
            2 Ciflg  Fixed Binary(31),
            2 Cilcd  Fixed Binary(31),
            2 Cipkey Character(16),
            2 Cilcct (2) Fixed Binary(31),
            2 Cilncd Fixed Binary(31),
            2 Cilnct (2) Fixed Binary(31),
            2 Cicdt  (2) Fixed Binary(31),
            2 Cilrdt (2) Fixed Binary(31);
  Declare 1 Finfo,
            2 Fial   Fixed Binary(31) Initial(0);
  Declare 1 Sinfo,
            2 Sial   Fixed Binary(31) Initial(0);

  Unit    = 'SCARDS  ';
  Rtn.Int = 0;
  Cinfo.Cial = Storage(Cinfo)/4;
  Call GFINFO(Unit,Rtn,Flag,Cinfo,Finfo,Sinfo,Ercode,Errmsg);
  If PLIRETV = 4 Then
     Put Edit(Ercode,Errmsg) (F(2),X(2),A(80));
  Else If PLIRETV > 4 Then
     Put Edit('Error return from GFINFO subroutine') (A);
  Else
     Edit(Rtn.Chr,'  Owner = ',Cinfo.Cionid) (A(20),A,A(4));
  Return;

End Test;
```

The above example calls the GFINFO subroutine to obtain catalog information about the file attached to the logical I/O unit SCARDS. The structure CINFO is passed to the subroutine; the subroutine upon return will insert the catalog information into this region. By using a structure, a packed region of varying data types can be defined. Again, the PLIRETV routine is called to obtain the return code from GFINFO.

```
     Test: Procedure Options(Main);

       Declare CHKACC  External Entry
                         (Character(*), *)
                       Returns (Fixed Binary(31))
                       Options (Fortran,Inter,Retcode),
               RENAME  External Entry
                         (Character(*),Character(*))
                       Options (Assembler,Inter,Retcode),
               PLIRETV Builtin,
               UNSPEC  Builtin;
               Access  Fixed Binary(31),
               Oldnam  Character(18),
               Newnam  Character(18),
               Mask    Fixed Binary(31) Initial(16);

       Declare 1 Triple,
                 2 Ccid Character(4),
                 2 Proj Character(4),
                 2 Pkey Character(18);

       Oldnam = 'DATA1 ';
       Newnam = 'NEWDATA1 ';
       Triple.Ccid = 'WABC';
       Triple.Proj = 'WXYZ';
       Triple.Pkey = '*EXEC ';
       Access = CHKACC(Oldnam,Triple);
       If PLIRETV > 0 Then
          Put Edit('File does not exist') (A);
       Else If (UNSPEC(Access) & UNSPEC(Mask)) ¬= UNSPEC(Mask) Then
          Put Edit('Rename access not allowed') (A);
       Else Do;
          Call RENAME(Oldnam,Newnam);
          If PLIRETV > 0 Then
             Put Edit('Error return from RENAME subroutine') (A);
          Else
             Put Edit('File successfully renamed') (A);
          End;
       Return;

     End Test;
```

In the above example, the CHKACC subroutine is called as a Fixed Binary function to obtain the access of the file WABC:DATA1. If rename access is allowed, then the RENAME subroutine is called to rename the file to NEWDATA.

In PL/I, there are no special problems connected with calling subroutines that have a variable number of parameters if the subroutine is declared without a parameter list (if the parameter is specified, the compiler will generate a warning message if the number of parameters given in the actual call does not agree with the declaration). In the example below, the COMMAND subroutine is called once with three parameters and again with five parameters.

```
     Test: Procedure Options(Main);

       Declare COMMAND External Entry
                       Options (Assembler,Inter,Retcode),
               Cmdtext Character(255),
               Cmdlen  Fixed Binary(31),
               Sws     Fixed Binary(31) Initial(0),
               Summary Fixed Binary(31),
               Ercode  Fixed Binary(31);

       Cmdtext = '$Display Timespelledout ';
       Cmdlen  = LENGTH(Cmdtext);
       Call COMMAND(Cmdtext,Cmdlen,Sws);
       Cmdtext = '$Display Timemisspelledout ';
       Cmdlen  = LENGTH(Cmdtext);
       Call COMMAND(Cmdtext,Cmdlen,Sws,Summary,Ercode);
       If Summary > 0 Then
          Put Edit('Command Error Code = ',Ercode) (A,F(3));
       Return;

     End Test;
```

R-Type Subroutines

R-type subroutines can be called from PL/I by using the RCALL subroutine. The RCALL subroutine sets up a call to an R-type subroutine by inserting the parameters into the proper registers for the call to the system subroutine.

The call to the RCALL subroutine is made in the following manner:

```
     DECLARE RCALL EXTERNAL ENTRY
                   OPTIONS (ASSEMBLER,INTER,RETCODE),
             subr  EXTERNAL ENTRY;
     CALL RCALL(subr,r1,p1,...,r2,p2,...);
```

where "r1" is the number of registers to be set up on the call to "subr" and "p1,..." are the values to be inserted into the registers beginning with general register 0; "r2" is the number of registers to contain return values from "subr" and "p2,..." are the variables that will contain the returned values starting with general register 0.

```
     Test: Procedure Options(Main);

       Declare RCALL   External Entry
                       Options (Assembler,Inter,Retcode),
               GETFD   External Entry,
               RENUMB  External Entry
                         (Pointer,Fixed Binary(31),Fixed Binary(31),
                          Fixed Binary(31),Fixed Binary(31))
                       Options (Assembler,Inter,Retcode),
               PLIRETV Builtin,
               First   Fixed Binary(31) Initial(1000),
               Last    Fixed Binary(31) Initial(100000000),
               Beg     Fixed Binary(31) Initial(1000),
               Inc     Fixed Binary(31) Initial(1000),
               Filenam Character(18),
               Fdub    Pointer,
               Dumy    Fixed Binary(31),
               F1      Fixed Binary(31) Initial(1),
               F2      Fixed Binary(31) Initial(2);

       Filenam = 'DATA1 ';                    /* Set file name */
       Call RCALL(GETFD,F2,Dumy,ADDR(Filenam),F1,Fdub);  /* Get FDUB */
       Call RENUMB(Fdub,First,Last,Beg,Inc);  /* Renumber file */
       If PLIRETV > 0 Then                    /* Test return code */
          Put Edit('Error return from RENUMB subroutine') (A);
       Else
          Put Edit('File successfully renumbered') (A);
       Return;

     End Test;
```

In the above example, the GETFD subroutine is called to obtain a FDUB-pointer for the file DATA1; the FDUB-pointer is then passed on to the RENUMB subroutine to renumber the file. The GETFD subroutine requires that register 1 contain the address of the name of the subroutine as returned by the ADDR built-in function. The register count is 2, since RCALL initializes registers beginning with register 0 (in this case, register 0 is called with a dummy argument of zero). Upon return, GETFD returns the FDUB-pointer in register 0. Hence the register count is 1 and the FDUB-pointer is inserted in the variable FDUB.

```
     Test: Procedure Options(Main);

       Declare RCALL    External Entry
                        Options (Assembler,Inter,Retcode),
               GDINFO   External Entry,
               FREESPAC External Entry,
               PLIRETV  Builtin,
               Unit1    Character(4),
               Unit2    Character(4),
               Dumy     Fixed Binary(31),
               F0       Fixed Binary(31) Initial(0),
               F2       Fixed Binary(31) Initial(2),
               Infoptr  Pointer;

        Declare 1 Info Based (Infoptr),
                 2 Fdub   Fixed Binary(31),
                 2 Devtyp Character(4),
                 2 Inlen  Fixed Binary(15),
                 2 Outlen Fixed Binary(15),
                 2 Use    Character(1),
                 2 Device Character(1),
                 2 Sws1   Character(1),
                 2 Sws2   Character(1),
                 2 Mods   Fixed Binary(31),
                 2 Beglnr Fixed Binary(31),
                 2 Prvlnr Fixed Binary(31),
                 2 Endlnr Fixed Binary(31),
                 2 Inclnr Fixed Binary(31),
                 2 Namptr Fixed Binary(31),
                 2 Msgptr Fixed Binary(31),
                 2 Iosave Fixed Binary(31),
                 2 Lastrc Fixed Binary(31),
                 2 Reglen Fixed Binary(15),
                 2 Width  Fixed Binary(15),
                 2 Macid  Fixed Binary(31);

        Unit1 = 'SCAR';               /* Set I/O unit name */
        Unit2 = 'DS  ';
        Call RCALL(GDINFO,F2,Unit1,Unit2,F2,Dumy,Infoptr);
        If PLIRETV > 0 Then
           Put Edit('Error return from GDINFO subroutine') (A);
        Else Do;
           Put Edit('Type = ',Info.Devtyp) (A,A(4));
           Call RCALL(FREESPAC,F2,F0,Infoptr,F0); /* Release storage */
           End;                                /* from GDINFO     */
        Return;

      End Test;
```

In this example, the GDINFO subroutine is called by the RCALL subroutine. Two registers (general registers 0 and 1) are set up on the call to contain the eight-character logical I/O unit name. Two registers are also set up for the return. Register 1 will contain the address of the GDINFO information region; register 0 is not used, hence a dummy argument must be inserted into the RCALL parameter list. This example also illustrates the case where a system subroutine returns a pointer to an area of storage acquired by the subroutine itself. Hence the variable INFOPTR, which upon return will contain the address of the acquired storage, must be declared as a pointer variable. The statement following the subroutine call is then used to copy the contents of that storage into the BASED structure INFO so that the individual items of GDINFO information can be accessed by the program. At the end of the program, the FREESPAC subroutine is call by the RCALL subroutine to release the storage acquired by the GDINFO subroutine. Note that the

GDINF alternative entry to the GDINFO subroutine also could have been called; this would circumvent the problem of using pointer variables and based structures in the PL/I program and having to call FREESPAC to release the GDINFO area.

Special Cases

Several system subroutines cannot be directly called by PL/I programs because they require nonstandard calls for exit routines (e.g., ATTNTRP or TIMNTRP). However, most of these subroutines have S-type alternatives that perform similar functions. Some of the more common alternative entries (or subroutines) are given in the table below.

System Subroutine	Alternative Entry
ATTNTRP	ATNTRP
LINK	LINKF
LOAD	LOADF
REWIND#	REWIND
TIMNTRP	TICALL
UNLOAD	UNLDF
XCTL	XCTLF

Further information about calling external subroutines from PL/I is given in the section "Interlanguage Communication Facilities" in this volume.

PL/I BUILT-IN FUNCTIONS AND PSEUDO-VARIABLES

The following lists briefly the PL/I built-in functions and pseudo-variables. A few of them, such as POINTER built-in function, are not available in the PL/I (F) language. Several others such as DATE, TIME, UNSPEC have different meanings depending on whether they are used with the PL/I (F) or Optimizing compilers.

String Handling Functions

BIT(expr[,size])	Converts "expr" to a bit string of the given length "size".
BOOL(x,y,w)	Performs a Boolean operation "w" on two given bit strings "x" and "y".
CHAR(expr[,size])	Converts "expr" to a character string of the given length "size".
HIGH(i)	Forms a character string of the given length "i", with each character being hexadecimal FF.
INDEX(str,config)	Searches for a bit or character string "config" in the string "str".
LENGTH(string)	Finds the current length of "string".
LOW(i)	Forms a character string of the given length "i", with each character being hexadecimal 00.
REPEAT(string,i)	Forms a string with "string" repeated "i" times.
STRING(x)	Forms a string by concatenating all elements of an aggregate variable "x".
SUBSTR(string,i[,j])	Extracts a substring of length "j" from a "string" starting at position "i". If "j" is omitted, it is equivalent to LENGTH(string) -i +1.
TRANSLATE(s,r[,p])	Translates from the source string "s" by replacing characters in the positional string "p" with the corresponding characters in the replacement string "r". If "p" is omitted, then "p" represents 256 EBCDIC characters arranged in ascending order, from 00 to FF in hexadecimal.
UNSPEC(x)	Returns a bit string that internally represents "x".
VERIFY(str1,str2)	Returns 0 if all characters or bits in "str1" can be found in "str2", otherwise returns the index of the first character or bit in "str1" that cannot be found in "str2".

Arithmetic Functions

ABS(x)	Absolute value of "x".
ADD(x,y,p[,q])	x+y with precision (p,q).
BINARY(x[,p[,q]])	Converts "x" to binary value of precision (p,q).
CEIL(x)	Smallest integer ≥ real "x".
COMPLEX(x,y)	Forms a complex number x+yI, x and y being both real.
CONJG(x)	Returns a complex conjugate of "x", i.e., REAL(x)-IMAG(x).
DECIMAL(x[,p[,q]])	Converts "x" to a decimal value of precision (p,q).
DIVIDE(x,y,p[,q])	Divides "x" by "y" with result precision (p,q).
FIXED(x[,p[,q]])	Converts "x" to a fixed value of precision (p,q).
FLOAT(x[,p])	Converts "x" to a floating-point value of precision (p,q).
FLOOR(x)	Largest integer < "x".
IMAG(x)	Imaginary part of complex number "x".
MAX(x1,x2,...,xn)	Highest value of x1, x2, ..., xn.
MIN(x1,x2,...,xn)	Lowest value of x1, x2, ..., xn.
MOD(x,y)	Remainder from x/y, i.e., the lowest positive value "z" such that (x-z)/y=n where "n" is an integer.
MULTIPLY(x,y,p[,q])	Multiplies "x" and "y" with result precision (p,q).
PRECISION(x,p[,q])	Converts "x" to a value of precision (p,q).
REAL(x)	Real part of complex number "x".
ROUND(expr,n)	"expr" is rounded to nth digit to the right of decimal (or binary) point if n > 0; at (n+1)th digit to the left of decimal (or binary) point if n ≤ 0.
SIGN(x)	Returns +1 if x > 0, 0 if x = 0, -1 if x < 0.
TRUNC(x)	Truncates value "x" to an integer by returning CEIL(x) if x < 0, FLOOR(x) if x ≥ 0.

Mathematical Functions

ACOS(x)	Inverse cosine in radians.
ASIN(x)	Inverse sine in radians.
ATAN(x[,y])	Arctangent of x or x/y, in radians.
ATAND(x[,y])	Arctangent of x or x/y, in degrees.
ATANH(x)	Inverse hyperbolic tangent of x.
COS(x)	Cosine of x, "x" in radians.
COSD(x)	Cosine of x, "x" in degrees.
COSH(x)	Hyperbolic cosine of x.
ERF(x)	Error function of x

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

ERFC(x)	Complement of the error function of x, i.e., 1-ERF(x).
EXP(x)	Returns the value of "e" raised to the power of "x". "e" is the base of the natural logarithm system, 2.718281828.
LOG(x)	Natural logarithm (i.e., base e) of "x".
LOG10(x)	Common logarithm (i.e., base 10) of "x".
LOG2(x)	Binary logarithm (i.e., base 2) of "x".
SIN(x)	Sine of x, "x" in radians.
SIND(x)	Sine of x, "x" in degrees.
SINH(x)	Hyperbolic sine of x.
SQRT(x)	Square root of x.
TAN(x)	Tangent of x, "x" in radians.
TAND(x)	Tangent of x, "x" in degrees.
TANH(x)	Hyperbolic tangent of x.

Array Manipulation Functions

ALL(x)	Returns a bit string with length of the longest element of array "x". If *i*th bits of all elements exist and are 1, then the *i*th bit is set to 1; otherwise, it is set to 0.
ANY(x)	Returns a bit string with length of the longest element of array "x". If the *i*th bit of any element in "x" exists and is 1, then the *i*th bit of the result is set to 1; otherwise, it is set to 0.
DIM(x,n)	Current extent for the *n*th dimension of array "x". It is equivalent to HBOUND(x,n) - LBOUND (x,n) + 1.
HBOUND(x,n)	Current upper bound of *n*th dimension of array "x".
LBOUND(x,n)	Current lower bound of *n*th dimension of array "x".
POLY(a,x)	Forms a polynomial from one-dimension arrays "a" and "x". If declared a(m:n) and x(p:q), then the result is $a(m)+\sum_{j=1}^{n-m}\left(a(m+j)\bullet\prod_{i=0}^{j-1}x(p+i)\right)$ If "x" is an element, it is interpreted as an array of one element.
PROD(x)	Product of all elements of array "x".
SUM(x)	Sum of all elements of array "x".

Condition Functions

DATAFIELD	Returns a varying-length character string of the data field that caused the NAME condition to be raised.
ONCHAR	Returns a character string of length 1, con-

	taining the character that caused the CONVERSION condition to be raised.
ONCODE	Returns a binary integer defining the interrupt that caused the entry into ON-unit. See Section H: "ON-Conditions" in _PL/I Language Reference Manual_.
ONCOUNT	Returns the number of single I/O interrupts that remain to be handled.
ONFILE	Returns a varying-length character string showing the name of the file for which an I/O or CONVERSION condition was raised.
ONKEY	Returns a varying-length character of the key for which the I/O condition was raised.
ONLOC	Returns a varying-length character string, giving the name of the entry point for which an ON-condition was raised.
ONSOURCE	Returns a varying-length character string showing the field being processed when a CONVERSION was raised.

Storage Control Functions

ADDR(x)	Returns a pointer value identifying the location at which "x" was allocated.
ALLOCATION(x)	Returns '1'B if the controlled variable "x" has been allocated; otherwise '0'B.
CURRENTSTORAGE(x)	Returns the current storage of variable "x" in bytes.
EMPTY	Returns an area of zero size with no allocations.
NULL	Returns a null pointer or offset value.
OFFSET(x1,x2)	Returns the offset value of pointer "x1" in area "x2".
POINTER(x1,x2)	Returns the pointer value of offset "x1", given area "x2".
STORAGE(x)	Returns the storage required by the variable "x" in bytes.

I/O Functions

COUNT(file)	Returns the number of data items transmitted during the last GET or PUT operation on the stream "file".
LINENO(file)	Returns the current line number for the PRINT "file".
SAMEKEY(file)	Returns '1'B if a record is followed by another record with the same key.

Preprocessor Functions

COMPILETIME	Returns a character string of 18 characters, indicating the date and the time of compilation. It is in the form "DD MMM YY HH.MM.SS",

	where "DD" is the day of the month, "MMM" the first three characters of the month’s name (e.g. JAN), "YY" the year, "HH" the current hour, "MM" the number of minutes, and "SS" the number of seconds.
COUNTER	Returns a character string of five digits, starting with 00001 and then incremented by 00001.
INDEX(x1,x2)	Returns FIXED value indicating the position of "x2" in "x1".
LENGTH(x)	Returns FIXED value of the current length of the string "x".
PARMSET(x)	Returns ’1’B if the parameter has been set.
SUBSTR(x1,x2[,x3])	Returns a substring of the string "x1" at position "x2" with length "x3".

Miscellaneous Functions

DATE	Returns a character string of six characters in the form "yymmdd", where "yy" is the current year, "mm" is the current month, and "dd" is the current day of the month.
PLIRETV	Returns the current value of the PL/I return code, which is set by calling PLIRETC or after calling non-PL/I routines with RETCODE option.
TIME	Returns a character string of nine characters in the form "hhmmssttt", where "hh" is the current hour, "mm" the number of minutes, "ss" the number of seconds, and "ttt" the number of milliseconds.

Pseudo-Variables

Pseudo-variables are functions that can be used as receiving fields. They need to appear not only on the left-hand side of an assignment statement but also in the data list of a GET statement and as the string name in the STRING option of a PUT statement. Pseudo-variables cannot be nested, however.

COMPLEX(a,b)	The real part of the value is assigned to "a", and the imaginary part to "b". Both "a" and "b" are complex variables or arrays.
IMAG(c)	The real part of the complex value is assigned to the imaginary part of the complex variable or array "c".
ONCHAR	The expression being assigned is converted to a character string of length 1 and replaces the character that caused the CONVERSION error to be raised.
ONSOURCE	The expression being assigned is converted to a character string, padded with blanks if neces- sary to match the length of the erroneous character string that caused the CONVERSION

	error to be raised.
REAL(c)	The real part of the complex value is assigned to the complex variable or array "c".
STRING(x)	"x" is an element, array, or structure variable composed entirely of character strings with numeric character data, if any, or entirely of bit strings. The expression being assigned is converted to a character string or a bit string depending on "x". It is then assigned piece by piece to "x". When none remains, the rest of "x" will have varying strings null, and fixed strings filled with either blanks or zeros.
SUBSTR(string,i[,j])	The value is converted to a bit or character string and then assigned to a substring of the variable "string".
UNSPEC(v)	The value is converted to a bit string with length as that of "v" and then assigned to "v" without any conversion. "v" may be an element or array variable that is arithmetic, string, area, pointer, or offset. If "v" is a varying-length string, its two-byte prefix will be in the returned bit-string.

PL/I (F) OBJECT-TIME ERROR MESSAGES

The error messages in this section may be generated on SPRINT or SERCOM (if SPRINT cannot be used) as the result of an exceptional or error condition occurring during the execution of a PL/I program.

Error messages are generated at execution time for two main reasons:

(1) An error condition for which no specific ON-condition exists in PL/I. A error message is printed, and the ERROR ON-condition is raised.

(2) An ON-condition is raised, by compiled code or by the library, and the action required is system action, for which the language specifies COMMENT as part of the necessary action.

The object-time messages will take one of the following forms:

```
IHEnnnI FILE name - message AT location

IHEnnnI rtname - message AT location

IHEnnnI message AT location
```

where:

nnn	is the message number,
name	is the name of the file associated with the error (given only in I/O error messages),
message	is the text of the error message,
rtname	is the name of the PL/I library routine in which the error occurred (given only for computational subroutines), and

location is either

```
OFFSET ±xxxxx FROM ENTRY POINT yyyyyy
```

or

```
OFFSET ±xxxxx FROM ENTRY POINT OF zzzz ON-UNIT
```

If the statement-number compiler option STMT has been specified, each message will also contain

IN STATEMENT nnnnn

prior to "AT location". "nnnnn" gives the number of the statement in which the condition occurred.

The error messages for other than ON-type errors are mainly self-explanatory. Explanations in the following lists are given only when the message is not self-explanatory. For brevity, the "AT location" portion of each message is omitted.

General Error Messages

IHE001I FINISH

The FINISH condition has been raised.

IHE002I ERROR

The ERROR condition has been raised and a SNAP message was requested.

IHE003I SOURCE PROGRAM ERROR IN STATEMENT nnnnn

This message will always contain a statement number whether or not the compiler STMT option is specified.

IHE004I INTERRUPT IN ERROR HANDLER

An unexpected program interrupt occurred during the handling of another error. This indicates that the program has a disastrous error in it, such as DSA (Dynamic Save Area) chain out of order or instructions overwritten. The program is terminated and a dump is produced if the MTS $SET ERRORDUMP option is ON, and, if in batch mode, the job is terminated. The user should retry the program with STRINGRANGE and SUBSCRIPTRANGE enabled.

IHE005I PRV GREATER THAN 4096 BYTES

This error arises when the sum of the number of procedures, the number of files, and the number of controlled variables exceeds 1000. It causes a return to the system; the PL/I program is not entered.

IHE006I NO MAIN PROCEDURE

No external procedure in the program has been given the option MAIN. The program is not executed.

IHE009I IHEDUMP - NO DUMP OUTPUT

The conversational user does not wish to specify the dump output to be produced on the PL/I file PL1DUMP. In batch mode, the output is produced on *SINK*.

I/O Errors

IHE018I FILE name - FILE TYPE NOT SUPPORTED

IHE020I FILE name - ATTEMPT TO READ OUTPUT FILE

IHE021I FILE name - ATTEMPT TO WRITE INPUT FILE

IHE022I FILE name - GET/PUT STRING EXCEEDS STRING SIZE

For input: programmer has requested more than exists on the input string.

For output: programmer is trying to write more than his output string will hold.

IHE023I FILE name - OUTPUT TRANSMIT ERROR NOT ACCEPTABLE

The ERROR condition is raised, (a) upon return from a TRANSMIT ON-unit, if the device in error is other than a printer, or (b) if access to a file by RECORD I/O has been attempted after the TRANSMIT condition has been raised for output.

IHE024I FILE name - PRINT/OPTION FORMAT ITEM FOR NON-PRINT FILE

Attempt to use PAGE, LINE, or SKIP ≤0 for a non-print file.

IHE025I DISPLAY - MESSAGE OR REPLY AREA LENGTH ZERO

This message appears only if the REPLY option specifies a null string or the user issues an end-of-file.

IHE026I FILE name - DATA DIRECTED INPUT - INVALID ARRAY DATUM

The number of subscripts on the external medium does not correspond to the number of declared subscripts.

IHE027I GET STRING - UNRECOGNIZABLE DATA NAME

For GET DATA: the name of the data item found in the string is not known at the time of the GET statement.

For GET DATA data list: the name of the data item found in the string is not specified in the list.

IHE029I FILE name - UNSUPPORTED FILE OPERATION

The program has executed an I/O statement with an option or verb not applicable to the specified file. For example:

<u>I/O Option or Verb</u>	<u>File Attribute</u>
READ {SET|LOCATE}	{DIRECT|SEQUENTIAL UNBUFFERED}
REWRITE (without FROM)	{SEQUENTIAL {INPUT|OUTPUT|UPDATE} | DIRECT {INPUT|OUTPUT}}
{LINESIZE|PAGESIZE}	STREAM {INPUT|OUTPUT}

IHE030I FILE name - REWRITE/DELETE NOT IMMEDIATELY PRECEDED BY READ

IHE036I FILE name - IMPLICIT OPEN FAILURE - CANNOT PROCEED

There has been a failure in an implicit OPEN operation.

IHE038I FILE name - ENDFILE FOUND UNEXPECTEDLY IN MIDDLE OF DATA ITEM

The ERROR condition is raised when an end-of-file is encountered before the delimiter when scanning list-directed or data-directed input, or if the field width in the format list of edit-directed input would take the scan beyond the end-of-file.

<u>I/O ON-Conditions</u>

All of these conditions may be raised by the SIGNAL statement.

IHE100I FILE name - UNRECOGNIZABLE DATA NAME

Initiating ON-condition: NAME

(1) GET DATA: name of data item found on external medium is not known at the time of the GET statement.
(2) GET DATA data list: name of data item found on external medium is not specified in the list.

IHE110I FILE name - RECORD CONDITION SIGNALED

IHE111I FILE name - RECORD VARIABLE SMALLER THAN RECORD SIZE

The variable specified in the READ statement INTO option allows fewer characters than exist in the record.

F-format records: a WRITE statement attempts to put a record smaller than the record size.

IHE112I FILE name - RECORD VARIABLE LARGER THAN RECORD SIZE

The variable specified in the READ statement INTO option requires more characters than exist in the record; or a WRITE statement attempts to put out a record greater than the available record size; or a REWRITE statement attempts to replace a record with one of a greater size.

IHE113I ATTEMPT TO WRITE/LOCATE ZERO LENGTH RECORD

A WRITE or REWRITE statement attempts to put out a record of zero length, or a LOCATE statement attempts to get buffer space for a record of zero length.

IHE114I FILE name - ZERO LENGTH RECORD READ

A record of zero length has been read from a KEYED file. This message normally should not occur.

IHE120I FILE name - PERMANENT INPUT ERROR

Initiating ON-condition: TRANSMIT

IHE121I FILE name - PERMANENT OUTPUT ERROR

Initiating ON-condition: TRANSMIT

IHE122I FILE name - TRANSMIT CONDITION SIGNALED

IHE130I FILE name - KEY CONDITION SIGNALED

IHE131I FILE name - KEYED RECORD NOT FOUND

A READ, REWRITE, or DELETE statement specified a record key which does not exist in a KEYED file.

IHE132I FILE name - ATTEMPT TO ADD DUPLICATE KEY

A WRITE statement specified a key value which already exists within the KEYED file.

IHE135I FILE name - KEY SPECIFICATION ERROR

Without the GENKEY option, keys must have a length of 4. With the GENKEY option, keys must conform to the form ±ddddd.ddd. For sequential files, this error appears if keys are invalid.

IHE140I FILE name - END OF FILE ENCOUNTERED

Initiating ON-condition: ENDFILE

IHE150I FILE name - IS NOT SPECIFIED

This message is generated if the file name is not specified.

IHE151I FILE name - CONFLICTING DECLARE AND OPEN ATTRIBUTES

Initiating ON-condition: UNDEFINEDFILE

There is a conflict between the declared PL/I file attributes. For example:

Attribute	Conflicting Attributes
PRINT	INPUT, UPDATE, RECORD, DIRECT, SEQUENTIAL, BUFFERED, UNBUFFERED, KEYED
STREAM	UPDATE, RECORD, DIRECT, SEQUENTIAL, BUFFERED, UNBUFFERED, KEYED
DIRECT	SEQUENTIAL, BUFFERED, UNBUFFERED
UPDATE	INPUT, OUTPUT
OUTPUT	INPUT
BUFFERED	UNBUFFERED

Some attributes may have been supplied when a file is opened implicitly. Examples of attributes implied by I/O statements are:

I/O Statement	Implied Attributes
DELETE	RECORD, DIRECT, UPDATE
GET	INPUT
LOCATE	RECORD, OUTPUT, SEQUENTIAL, BUFFERED
PUT	OUTPUT
READ	RECORD, INPUT
REWRITE	RECORD, UPDATE
WRITE	RECORD, OUTPUT

In turn, certain attributes may imply other attributes:

Attribute	Implied Attributes
BUFFERED	RECORD, SEQUENTIAL
DIRECT	RECORD, KEYED
KEYED	RECORD
PRINT	OUTPUT, STREAM
SEQUENTIAL	RECORD
UNBUFFERED	RECORD, SEQUENTIAL
UPDATE	RECORD

Finally, a group of alternate attributes has one of the group as a default. The default is implied if none of the group is specified explicitly or is implied by other attributes or by the opening I/O statement. The groups of alternates are:

Group	Default
{STREAM|RECORD}	STREAM
{INPUT|OUTPUT|UPDATE}	INPUT
{SEQUENTIAL|DIRECT}	SEQUENTIAL
{BUFFERED|UNBUFFERED}	BUFFERED

IHE152I FILE name - FILE TYPE NOT SUPPORTED

Initiating ON-condition: UNDEFINEDFILE

In MTS, the following file types are not supported: BACKWARDS, EXCLUSIVE, REGIONAL, TELEPROCESSING, and TRANSIENT.

IHE154I FILE name - UNDEFINEDFILE CONDITION SIGNALED

IHE156I FILE name - CONFLICTING ATTRIBUTE AND ENVIRONMENT PARAMETERS

Initiating ON-condition: UNDEFINEDFILE

Examples of conflicting parameters are:

ENVIRONMENT Parameter	File Attribute
INDEXED	STREAM
CONSECUTIVE	DIRECT
INDEXED	DIRECT OUTPUT
INDEXED	OUTPUT without KEYED
Blocked records	BUFFERED

IHE157I FILE name - CONFLICTING ENVIRONMENT AND/OR DDEF MODIFIERS

Initiating ON-condition: UNDEFINEDFILE

One of the following conflicts exists:

(1) The device attached to a KEYED file is not an MTS file.
(2) KEYED files are V-formatted.

IHE159I FILE name - INCORRECT BLOCKSIZE AND/OR LOGICAL RECORD SIZE

Initiating ON-condition: UNDEFINEDFILE

One of the following situations exists:

(1) F-format records:

(a) The specified block size is less than the logical record length.
(b) The specified block size is not a multiple of the logical record length.

(2) V-format records:

(a) The specified block size is less than the logical record length + 4.
(b) The logical record length is less than 14 for a RECORD file or 15 for a STREAM file.
(c) The logical record length of spanned records is greater than 32,767 bytes or less than 5 bytes.
(d) The block size of spanned records is less than 9 bytes.

(3) Logical record length is negative.

IHE160I FILE name - LINESIZE GT IMPLEMENTATION DEFINED MAX LENGTH

Initiating ON-condition: UNDEFINEDFILE

The implementation-defined maximum line size is:

F-format records:	32759
V-format records:	32751

Computational Errors

IHE200I rtname - X LT 0 IN SQRT (X)

IHE202I rtname - X LT OR = 0 IN LOG(X) OR LOG2(X) OR LOG10(X)

IHE203I rtname - ABS(X) GE (2**50)*K IN SIN(X) OR COS(X) (K=PI) OR SIND(X) OR COSD(X) (K=180)

IHE204I rtname - ABS(X) GE (2**50)*K in TAN(X) (K=PI) OR TAND(X) (K=180)

IHE206I rtname - X=Y=0 IN ATAN(Y,X) or ATAND(Y,X)

IHE208I rtname - ABS(X) GT OR = 1 IN ATANH(X)

IHE209I rtname - X=0, Y LE 0 IN X**Y

IHE210I rtname - X=0, Y NOT POSITIVE REAL IN X**Y

IHE211I rtname - Z=+I OR -I IN ATAN(Z) OR Z=+1 OR -1 IN ATANH(Z)

IHE212I rtname - ABS(X) GE (2**18)*K IN SIN(X) OR COS(X) (K=PI) OR SIND(X) OR COSD(X) (K=180)

IHE213I rtname - ABS(X) GE (2**18)*K IN TAN(X) (K=PI) OR TAND(X) (K=180)

Computational ON-Conditions

All of these conditions may be raised by the SIGNAL statement.

IHE300I OVERFLOW

This condition is raised, by PL/I library routines or by compiled code, when the exponent of a floating-point number exceeds the permitted maximum, as defined by the implementation.

IHE310I SIZE

This condition is raised, by PL/I library routines or by compiled code, when an assignment is attempted where the number to be assigned will not fit into the target field. This condition can be raised by allowing the fixed-overflow interrupt to occur on account of SIZE. If associated with I/O, the "FILE name" will be inserted between the message number and the message text.

IHE320I FIXEDOVERFLOW

This condition is raised, by PL/I library routines or by compiled code, when the result of a fixed-point binary or decimal operation exceeds the maximum field width as defined by the implementation.

IHE330I ZERODIVIDE

This condition is raised, by PL/I library routines or by compiled code, when an attempt is made to divide by zero, or when the quotient exceeds the precision allocated for the result of a division. The condition can be raised by hardware interrupt or by special coding.

IHE340I UNDERFLOW

This condition is raised, by PL/I library routines or by compiled code, when the exponent of a floating-point number is smaller than the implementation-defined minimum. The condition does not occur when equal floating-point numbers are subtracted.

IHE350I STRINGRANGE

This condition is raised by PL/I library routines when an invalid reference by the SUBSTR built-in function or pseudo-variable has been detected.

IHE360I AREA CONDITION RAISED IN ALLOCATE STATEMENT

There is not enough room in the area in which to allocate the based variable.

IHE361I AREA CONDITION RAISED IN ASSIGNMENT STATEMENT

There is not enough room in the area to which the based variable is being assigned.

IHE362I AREA SIGNALED

Structure and Array Errors

IHE380I IHESTR - STRUCTURE OR ARRAY LENGTH GE 16**6 BYTES

During the mapping of a structure or array, the length of the structure or array has been found to be greater than or equal to 16**6 bytes.

IHE381I IHESTR - VIRTUAL ORIGIN OF ARRAY GE 16**6 OR LE -16**6

During the mapping of a structure, the address of the element with zero subscripts in an array, whether it exists or not, has been computed to be outside the range (-16**6 to 16**6).

IHE382I IHESTR - UPPER BOUND LESS THAN LOWER BOUND

During the mapping of an array or structure, an upper bound of a dimension has been found to be less than the corresponding lower bound. If only an upper bound was declared, then it may currently be less than one, the implied lower bound.

Control Program Restrictions

IHE400I DELAY STATEMENT EXECUTED, NOT PERMITTED IN MTS

Condition Type ON-Conditions

IHE500I SUBSCRIPTRANGE

This condition is raised, by PL/I library routines or by compiled code, when a subscript is evaluated and found to lie outside its specified bounds, or by the SIGNAL statement.

IHE501I CONDITION

This condition is raised by execution of a SIGNAL (identifier) statement, referencing a programmer-specified EXTERNAL identifier.

Conversion ON-Conditions

Conversion errors occur most often on input, either owing to an error in the input data, or because of an error in a format list. For example, in edit-directed input, if the field width of one of the items in the data list is incorrectly specified in the format list, the input stream will get out of step with the format list and a conversion error is likely to occur.

IHE600I CONVERSION

IHE601I CONVERSION ERROR IN F-FORMAT INPUT

IHE602I CONVERSION ERROR IN E-FORMAT INPUT

IHE603I CONVERSION ERROR IN B-FORMAT INPUT

IHE604I ERROR IN CONVERSION FROM CHARACTER STRING TO ARITHMETIC

IHE605I ERROR IN CONVERSION FROM CHARACTER STRING TO BIT STRING

IHE606I ERROR IN CONVERSION FROM CHARACTER STRING TO PICTURED CHARACTER STRING

IHE607I CONVERSION ERROR IN P-FORMAT INPUT (DECIMAL)

IHE608I CONVERSION ERROR IN P-FORMAT INPUT (CHARACTER)

IHE609I CONVERSION ERROR IN P-FORMAT INPUT (STERLING)

Conversion Errors, Non-ON-Type

IHE700I INCORRECT E(W,D,S) SPECIFICATION

IHE701I F FORMAT W SPECIFICATION TOO SMALL

IHE702I A FORMAT W UNSPECIFIED AND LIST ITEM NOT TYPE STRING

IHE703I B FORMAT W UNSPECIFIED AND LIST ITEM NOT TYPE STRING

IHE704I A FORMAT W UNSPECIFIED ON INPUT

IHE705I B FORMAT W UNSPECIFIED ON INPUT

IHE706I UNABLE TO ASSIGN TO PICTURED CHARACTER STRING

A source datum which is not a character string cannot be assigned to a pictured character string because of a mismatch with the PICTURE description of the target.

IHE798I ONSOURCE TO ONCHAR PSEUDO-VARIABLE USED OUT OF CONTEXT

This message is printed and the ERROR condition raised if an ONSOURCE or ONCHAR pseudo-variable is used outside an ON-unit, or in an ON-unit other than either a CONVERSION ON-unit or an ERROR or FINISH ON-unit following from system action for CONVERSION.

IHE799I RETURN ATTEMPTED FROM CONVERSION ON-UNIT BUT SOURCE FIELD NOT MODIFIED

A CONVERSION ON-unit has been entered as a result of an invalid conversion, and an attempt has been made to return, and hence reattempt the conversion, without using one or the other of the pseudo-variables ONSOURCE or ONCHAR to change the invalid character.

Non-Computational Program Interrupt Errors

Certain program interrupts may occur in a PL/I program because the source program has an error which is severe but which cannot be detected until execution time. An example is a call to an unknown procedure, which will result in an illegal operation program interrupt. Other program interrupts, such as addressing, specification, protection, and data interrupts, may arise if PL/I control blocks have been destroyed. This can occur if an assignment is made to an array element whose subscript is out of range, since, if SUBSCRIPTRANGE has not been enabled, the compiler does not check array subscripts; a program interrupt may occur at the time of the assignment or at a later stage in the program. Similarly, an attempt to use the value of an array element whose subscript is out of range may cause an interrupt.

Care must be taken when parameters are passed to a procedure. If the data attributes of the arguments of the calling statement do not agree with those of the invoked entry point, or if an argument is not passed at all, a program interrupt may occur.

The use of the value of a variable that has not been initialized, or has had no assignment made to it, or the use of CONTROLLED variables that have not been allocated, may also cause one of these interrupts.

IHE800I INVALID OPERATION

IHE801I PRIVILEGED OPERATION

IHE802I EXECUTE INSTRUCTION EXECUTED

IHE803I PROTECTION VIOLATION

IHE804I ADDRESSING INTERRUPT

IHE805I SPECIFICATION INTERRUPT

IHE806I DATA INTERRUPT

This condition can be caused by an attempt to use the value of a FIXED DECIMAL variable when no prior assignment to, or initialization of, the variable has been performed.

Storage Management Errors

The following errors are associated with the handling of storage and transfer of control out of blocks. In some cases, these errors are a result of program error, but it is possible that the messages may be printed because the save area chain, allocation chain, or pseudo-register vector have been overwritten.

IHE902I GO TO STATEMENT REFERENCES LABEL IN AN INACTIVE BLOCK

> The label referred to cannot be found in any of the blocks currently active; blocks are not freed. The statement number and offset indicate the GO TO statement causing the error.

IHE904I TOO MANY ACTIVE ON UNITS AND ENTRY PARAMETER PROCEDURES

> There is an implementation limit to the number of ON-units and/or entry parameter procedures which can be active at any time. An entry parameter procedure is one that passes an entry name as a parameter to a procedure it calls. The total permissible number of these ON-units/entry parameter procedures is 127.

PL/I OPTIMIZER RUN-TIME ERROR MESSAGES

The error messages are listed in the IBM publication OS PL/I Optimizing Compiler: Messages, SC33-0027. The PL/I Optimizing compiler error messages are like the PL/I (F) error messages except that most of the messages are of the form: "IBMnnnI" instead of "IHEnnnI". Here is an example of an error message:

```
IBM204I 'ONCODE'=0084 'UNDEFINEDFILE' CONDITION RAISED
   FILE OR DEVICE NOT SPECIFIED ('ONFILE'= XOKZ)
   IN STATEMENT 2 AT OFFSET +000078 IN PROCEDURE WITH ENTRY ABC
```

Below are some changes and additions to the error messages printed in the IBM publication.

```
IBM013I  FILE PLIDUMP NOT SPECIFIED

IBM090I  'ONCODE'=0400  'ATTENTION' CONDITION
         RAISED BY 'SIGNAL' STATEMENT

IBM091I  'ONCODE'=0400  'ATTENTION' CONDITION RAISED

IBM926I  CHECKPOINT/RESTART NOT SUPPORTED IN MTS

IBM204I  'ONCODE'=0084  'UNDEFINEDFILE' CONDITION RAISED
         FILE OR DEVICE NOT SPECIFIED

IBM229I  'ONCODE'=0092   'UNDEFINEDFILE' CONDITION RAISED
         FILE OR DEVICE DOES NOT EXIST

IBM230I  'ONCODE'=0092   'UNDEFINEDFILE' CONDITION RAISED
         FILE OR DEVICE IS INVALID

IBM231I  'ONCODE'=0092   'UNDEFINEDFILE' CONDITION RAISED
         FILE ACCESS NOT ALLOWED

IBM232I  'ONCODE'=0092   'UNDEFINEDFILE' CONDITION RAISED
         FILE WAIT INTERRUPTED

IBM233I  'ONCODE'=0092   'UNDEFINEDFILE' CONDITION RAISED
         FILE DEADLOCK

IBM236I  'ONCODE'=0093   'UNDEFINEDFILE' CONDITION RAISED
         FILE NOT SUPPORTED BY MTS

IBM881I  'ONCODE'=9201  SORT/MERGE NOT SUPPORTED IN MTS
```

DIFFERENCES BETWEEN OS AND MTS PL/I (F)

(1) Standard system PL/I files are SCARDS for input and SPRINT for output instead of SYSIN and SYSPRINT, respectively.

(2) The TIME built-in function returns in MTS an 8-character string in form "hh:mm:ss" instead of the 9-character string "hhmmssttt". ttt is the number of milliseconds.

(3) The DATE built-in function returns in MTS an 8-character string "mm-dd-yy" instead of the OS 6-character string "yymmdd".

(4) The DISPLAY statement does not cause a message to be displayed to a machine operator but to a terminal user. Logical I/O units SERCOM and GUSER are used. The length of a message can be more than 72 characters long.

(5) The %INCLUDE compile-time statement is implemented in MTS with many differences from OS. See the description of %INCLUDE in the section "Compiling a PL/I Program."

(6) The KEYED files are different than in OS. GENKEY and DIRECT have different meanings. Also REGIONAL files are not supported.

(7) Multitasking is not supported in MTS.

(8) Following PL/I keywords should not be used:

BACKWARDS	G(size)	PRIORITY	TRANSIENT
BUFFERS(n)	INDEXAREA(size)	R(size)	TRKOFL
COMPLETION	NCP(n)	REGIONAL(1|2|3)	UNLOCK
EVENT	NOLOCK	STATUS	WAIT
EXCLUSIVE	PENDING(file)	TASK	

(9) The equivalent ENVIRONMENT options CTLASA and CTL360 in OS are CC and MCC, respectively.

(10) There are additional subroutines described in this volume.

DIFFERENCES BETWEEN OS AND MTS PL/I OPTIMIZING COMPILERS

(1) All the object modules produced by the optimizing compiler have as the first record an MTS loader record " OPT SAVESD=ON". This record should not be deleted unless the objects are to be distributed to an OS installation. Using this loader record, appropriate library routines are loaded from the PL/I Optimizer library, and the programs will successfully run without fear of branching into the location 0.

(2) If a source program declares PLIXOPT as an external static varying character string, the MTS optimizing compiler does not process it and then produce a csect IBMBPOPT as does the OS version. This will allow the optimizer library to directly process the run-time options in the string. In addition, if PLIXOPT was declared of fixed length, an error message about the invalid length may be printed.

(3) If a user declares any MTS logical I/O unit name as a PL/I file, the optimizing compiler will prefix the name with an underscore: e.g., SCARDS to _SCARDS, since the optimizing compiler does not allow the user to declare a PL/I identifier with an underscore as the first letter. This applies to external PL/I files: SCARDS, SPRINT, SERCOM, SPUNCH, and GUSER. The change will allow any subroutine to call the MTS subroutines directly; otherwise, an operation interrupt may occur, because the external file control sections do not contain any executable code.

(4) The preprocessor %INCLUDE statement has a different syntax.

(5) Either the /PROCESS or the *PROCESS statement can be used for multiple compilation. The OS version allows only *PROCESS statements.

(6) The DISPLAY statement does not cause a message to be displayed to a machine operator but to a terminal user. Logical I/O units SERCOM and GUSER are used.

(7) The KEYED files are not supported in MTS.

(8) Multitasking is not supported in MTS.

(9) Asynchronous I/O with events is not supported in MTS.

(10) The following PL/I keywords should not be used in MTS:

ADDBUFF	GENKEY	NCP(n)	STATUS(x)
BKWD	INDEXAREA	NOLOCK	TASK
BUFFERS(n)	INDEXED	PASSWORD	TP(M|R)
BUFND(n)	KEY	PENDING(file)	TRANSIENT
BUFNI(n)	KEYED	PRIORITY	TRKOFL
BUFSP(n)	KEYFROM(x)	REGIONAL(1|2|3)	UNBUFFERED
COMPLETION(x)	KEYLENGTH(n)	REUSE	UNLOCK
DIRECT	KEYLOC(n)	SAMEKEY	VSAM
EVENT	KEYTO	SIS	WAIT
EXCLUSIVE			

PL/C

OVERVIEW

PL/C is a compile-and-execute system recognizing a dialect of the PL/I language. The system is oriented toward those developing PL/I programs, and the novice PL/I programmer. With this clientele in mind, PL/C provides a degree of source-language-level diagnostic assistance unattained by compilers for other high-level languages, even those for languages less complex than PL/I.

PL/C takes the point of view that, no matter how blatantly incorrect a program is, _some_ useful information can be extracted from it to aid the programmer in the next attempt to produce a correct program. PL/C's error-correcting repertoire ranges from the correction of spelling errors in the source program to the repair of every erroneous source-language statement in such a way that the program can _always_ be executed. Furthermore, errors which it does detect, and possibly corrects, are always clearly indicated. For example, errors in a source statement cause the statement to be repaired, if at all possible, and the PL/C text of the repaired statement to be presented, unambiguously indicating the correct (though necessarily speculative) statement formulation.

By default, PL/C performs a prodigious amount of error checking on an executing program; this checking may be disabled for well-tested programs to increase execution speed. Consonant with its compilation behavior, PL/C attempts to repair execution errors in order to prolong the lifetime of a computationally unhealthy program.

PL/C also provides debugging facilities for those programs that run but produce something other than the desired result. By extension of selected PL/I language constructs, PL/C permits the flow of execution and the change in the value of any program variable to be dynamically monitored in a conceptually clean and concise way. PL/C also permits the selective displaying of the values of program variables and execution information at program termination.

Despite its behavior during compilation with respect to errors, PL/C compiles at a rate faster than that of the MTS PL/I (F) compiler. The combination of a high compilation rate and tolerance of errors makes PL/C attractive as a PL/I program development tool and as an "instructional" compiler, where recompilation of the program is the dominant event in the program's lifetime. However, in production use, that is, in the continual use of the program for problem-solving purposes, the fact that recompilation is required before every execution of the program diminishes the attractiveness of PL/C. Further detracting from production use of PL/C is the requirement that all portions of the compiler reside in the user’s virtual memory space at all times during program compilation and execution. This may contribute significantly to the cumulative cost of many uses of a well-tested program. In the balance, the positive aspects of PL/C far outweigh the negative ones, and, unless features of the PL/I language not in the PL/C lexicon are used, PL/C should always be considered as a program development tool.

The major restrictions of PL/C as opposed to MTS PL/I (F) are:

(1) none of PL/I’s list processing, controlled storage, or based storage facilities are available,

(2) the PICTURE, DEFINED, and LIKE attributes are not recognized.

PL/C is further restricted in that it does not have the PL/I compile-time preprocessor facilities; it prohibits the use of statement names as identifier names and the passing of PAR field information to the PL/C and it does not offer the full PL/I range of features on some statements and variable attributes. PL/I and PL/C also differ in the area of I/O at the level of its interface between MTS and the source language. Whereas MTS PL/I offers a general, complex facility for the detailed specification of I/O transmission modes, PL/C offers a subset, but with subsequent ease in specification from the source-language level.

PL/C has extended the PL/I language in the area of debugging facilities. However, the presence of these features does not prevent programs using them from being compiled under PL/I, since PL/C also contains a facility whereby portions of PL/C source code may be considered PL/I, but <u>not</u> PL/C, comments. If those program sections peculiar to PL/C are sheltered by the above, this area of PL/I incompatibility can be removed.

The detailed differences between PL/I and PL/C occupy a major portion of the succeeding section, and specifics may be found there. The text is written assuming one has a knowledge of PL/I or is learning it from some source; it is not intended as an instructional text for either PL/I or PL/C. The section "PL/I Bibliography" contains a bibliography of some instructional publications.

An attempt has been made to structure this description so that information immediately needed to run PL/C is presented in the front, while the more detailed reference material appears later. The novice PL/C user is urged to read "Running PL/C in MTS" and "Diagnostic Assistance." The user should also read "Differences between PL/I (F) and PL/C" in order to become familiar with general and specific differences between the two programs. The section "Error Messages" is intended for reference only. "Differences between PL/I (F) and PL/C" contains a subsection "The PL/C Macro Feature" which is provided for the sake of completeness, as the feature is little used. The section "Internal Structure of PL/C" is intended for casual reading and is not pertinent to the running of PL/C.

September 1982

INTRODUCTION

PL/C was designed to permit efficient instruction in the PL/I language. It provides high-speed compilation, reasonably efficient execution, extraordinary diagnostic assistance, and upward compatibility with the IBM PL/I F-level compiler.

PL/C does not support the full PL/I language. The major omissions are:

- list processing,
- multitasking,
- compile-time facilities (except for INCLUDE and a non-PL/I-type macro processor).

There are other minor omissions described in "Differences between PL/I (F) and PL/C."

Some features have been added to PL/C that are not part of PL/I. These features are intended to provide additional diagnostic facilities. They include special options on the PUT statement, the FLOW condition, statements to control FLOW and CHECK printing, and pseudo-comments that can optionally be converted to source text. These pseudo-comments can be used to shelter the incompatible PL/C features so that a program can still be run under the PL/I (F) compiler.

This description is not intended to teach a beginner to write programs in PL/C. It specifically describes how PL/C differs from PL/I, and gives information necessary to use PL/C and interpret output provided by PL/C. Except as noted in this guide, PL/C is consistent and compatible with PL/I as defined by the IBM PL/I F-level compiler (see IBM references in the section "PL/I Bibliography"). The MTS version of PL/C (Release 7.6) is covered in this description.

RUNNING PL/C IN MTS

The $RUN Command

PL/C is invoked by issuing the MTS $RUN command:

```
 $RUN *PLC [logical I/O unit assignments] [PAR=compiler options]
```

where the bracketed items are optional.

Logical I/O Units Referenced:

SCARDS - for PL/C source program input and the SCARDS/SYSIN input files. The default is *SOURCE*.

SPRINT - for PL/C source program listing and diagnostic output, and the SPRINT/SYSPRINT output files. The default is *SINK*.

Others - as needed in the PL/C program. The logical I/O units SPUNCH, GUSER, and SERCOM can be directly referenced as PL/I file variables. Units 0-19 can be referenced through the TITLE option of the OPEN statement (see notes on the OPEN statement in "Differences between PL/I (F) and PL/C").

Compiler Parameters:

In the PAR field of the $RUN command, PL/C recognizes the following parameters with the associated effects. They may be specified in any order in the PAR field, and must be separated by commas. Abbreviations are underlined.

<u>D</u>EFAULTS= - define the default compiler options which will be set at the beginning of each PL/C program to be compiled. Any of the compilation options defined in "Compiler Options" below may appear on the right-hand side of this keyword. If more than one compilation option is to be specified, they must be enclosed within parentheses and separated by commas. A listing of the default parameters will appear at the beginning of the compilation if the OPTLIST option is selected.

HARDSTOP - suppress the trapping of program interrupts by PL/C. This is of dubious value except to those debugging the PL/C compiler; the default action is to let PL/C process program interrupts.

<u>ST</u>AT - print compilation and execution statistics on SERCOM if PL/C is being run in conversational mode and compiler output is being diverted to a file/device other than the terminal. STAT is the default option.

NOSTAT - suppress the printing of statistics in the above situation.

SIZE=n - define the amount of virtual memory to be acquired by PL/C for (PL/C) program code and dynamic storage allocation during execution. "n" specifies region size in K (multiples of 1024 bytes), such that 32 ≤ n ≤ 1024. The default is 40.

Example $RUN Commands:

```
$RUN *PLC PAR=SIZE=35

$RUN *PLC SCARDS=PLCFILE SPRINT=*PRINT* PAR=NOSTAT,D=(ALIST,HDRPG)

$RUN *PLC SCARDS=PROGRAM SPRINT=LISTING SPUNCH=AUX1 0=AUX2
```

Control Card Descriptions

The control cards described below are those required or permitted by the PL/C compiler. There are ten PL/C control cards which are used to set up programs for PL/C compilation and execution:

```
/COMPILE
/PROCESS
/OPTIONS
/DATA
/EXECUTE
/INCLUDE
/ATTACH
/MACRO
/MEND
/STOP
```

The slash (/) as shown above is the PL/C control card identifier in MTS. The (/) is preferred in MTS, but the identifier (*) is also recognized.

Control cards are not considered part of the source program and are not affected by the use of the SORMGIN option. That is, the control cards must always have the format described below.

(1) <u>The COMPILE Control Card</u>

Each PL/C program can be <u>optionally</u> preceded by this control card to signify the start of a new PL/C program. /COMPILE must be present in columns 1-8, and the card may also contain specification of various PL/C options, to override the default specifications built into the PL/C compiler (see the compiler options below). The COMPILE card is required when it is desired to process two or more PL/C programs back-to-back. The COMPILE card is placed <u>between</u> programs to reinitialize the compiler and signify the start of the next PL/C program. A separator page may precede each program to facilitate separating the programs in a batch job. If the ID option is given on the COMPILE card, the text from this option will appear on the separator page.

For compatibility with other documentation on PL/C (see the section "PL/I Bibliography"), the COMPILE card may be synonymously written as *PL/C.

(2) <u>The PROCESS Control Card</u>

If a program consists of several external procedures, the first procedure is optionally preceded by the COMPILE card, and each external procedure following the first procedure <u>must</u> be preceded by a card with /PROCESS in columns 1-8; /PROCESS will be supplied by PL/C, if it is omitted. Any of the compiler options given below, except as noted, can appear on a PROCESS card. If the PROCESS card changes any options defined on a preceding COMPILE card, the changes affect only the one external procedure following the PROCESS card. /PROCESS can be also be written as *PROCESS.

(3) <u>The OPTIONS Control Card</u>

The OPTIONS card permits compilation options to be changed <u>within</u> the source deck of a program. Any option available on the COMPILE card can appear on an OPTIONS card, except as noted below. An OPTIONS card may appear <u>anywhere</u> in the PL/C source program, and the options mentioned on it are <u>immediately</u> applied to the source program compilation. /OPTIONS must appear in columns 1-8. Specifying an option on an OPTIONS card changes the option only for that portion of the one external procedure following the OPTIONS card.

(4) <u>The DATA Control Card</u>

If data is required for execution of the program, it must be preceded by a card with /DATA in columns 1-5. If no data is required, this card is optional. If data is present, it starts on the card following the /DATA card, and not on the /DATA card itself. The data card margins default to 1 and 100. See the subsection "Input Card Format" below for further information regarding data cards. /DATA can also be written as *DATA.

(5) The EXECUTE Control Card

The EXECUTE control can be used interchangeably with the DATA control card, described above.

(6) The INCLUDE Control Card

The INCLUDE card is used to direct PL/C to begin reading input records from a source other than the current file or device. Following the /INCLUDE, which must be present in columns 1-8 of the input record, is a list of file or device names from which input records will be read in sequence. For example,

```
/INCLUDE FILE1,FILE2,FILE3
```

will cause the compiler to read from the MTS files FILE1, FILE2, and FILE3, continuing from one file to the next after an end-of-file is reached on its predecessor. The file or device names (FDnames) may also specify MTS line-number ranges and I/O modifiers, for example,

```
/INCLUDE FILE1(1,25)@UC
```

is acceptable.

INCLUDEs may appear in any input read by PL/C, and the text to be included may contain <u>any</u> type of PL/C input, including control cards and other INCLUDE cards. If encountered in the compilation phase, INCLUDE input has the SORMGIN parameters applied to it. /INCLUDE may also be written as *INCLUDE.

(7) The ATTACH Control Card

In order to provide the capability of attaching arbitrary MTS files or devices to file variables in a PL/C program, the ATTACH command has been added in the MTS version of PL/C.

The ATTACH command has the following form:

```
/ATTACH plcfile=mtsfdname[@F(nn)]
```

where "plcfile" is the name of a file variable used in the PL/C program, and "mtsfdname" is an MTS file or device name, with optional MTS-recognized modifiers and optional line-number range. "nn" specifies a record length to which all input records (if the file is used for input) will be padded with blanks, and the length to which output records will be truncated. If no @F modifier is given, the length will depend on the various attributes and options applied to the PL/C file variable on declaration and opening. See the notes for the OPEN statement in "Differences between PL/I (F) and PL/C" below.

ATTACH cards may appear anywhere in a PL/C input stream. The mapping they define is noted immediately, but is not applied until the "plcfile" is opened (or reopened), implicitly or explicitly, by the executing PL/C program. ATTACHments are recognized, once defined, for an entire invocation of the PL/C compiler, but may be redefined at any time.

Examples of /ATTACH Commands:

```
     /ATTACH INFILE=*SOURCE*
     /ATTACH OUTFILE=*SINK*@F(132)
     /ATTACH SCRATCH=-RECYCLE(100,200)@F(100)
     /ATTACH INPUT=MYFILE(1000)@-IC@F(40)
```

For similar facilities that may be specified at the PL/C language level, see the description of the ENVIRONMENT attribute and the OPEN statement description in "Differences between PL/I (F) and PL/C."

(8) The MACRO Control Card

See "The PL/C Macro Feature." The MACRO card may also be written as *MACRO.

(9) The MEND Control Card

See "The PL/C Macro Feature." The MEND card may also be written as *MEND.

(10) The STOP Control Card

The STOP control card is used to present a logical end-of-file to PL/C. When /STOP is encountered in a PL/C input stream in columns 1-5, an end-of-file condition is generated. A STOP card may be used to signify the end of an INCLUDE section, to terminate the input data to the executing program, or to terminate the compiler.

(11) Compiler Options

Options on the control cards are separated by blanks and/or commas, and may be continued onto subsequent cards. The continuation of any of these cards has a (/) or (*) in column 1 with columns 2 and 3 blank. An individual option may not be split over a card boundary.

Options specified on the /COMPILE card, and the default values for the options not specified, are in effect throughout the program, except as overridden by specifications on subsequent /PROCESS and /OPTIONS cards. After each external procedure options are reset to the "global" /COMPILE and default values. To facilitate complete suppression of the source listing, specifying NOSOURCE on the /COMPILE card overrides any subsequent SOURCE options.

The default options vary depending on whether PL/C is being run in batch or conversational mode. If in conversational mode and SPRINT is assigned to the terminal printer, the output is generally as abbreviated as possible. In the options list below, those options which are underlined are the defaults for batch and conversational use when SPRINT output is not directed to the terminal printer. For all other cases, a note is made indicating the conversational default if it is different from the batch default.

ATR, A, <u>NOATR</u>

> Produce attribute listing for all variables declared when ATR is in effect.

ALIST, AL, <u>NOALIST</u>

> Produce assembler listing of generated object code.

AUXIO=n, AU (on /COMPILE only)

> Limit on number of auxiliary input/output operations. The default is n=10000.

CMNTS, CMNTS=(n1,n2,...), C, <u>NOCMNTS</u>

> Contents of comments beginning with a colon (:) considered source text. If parameter(s) are given (1 ≤ ni ≤ 7), comments beginning with "ni" are also considered source text.

<u>CMPRS</u>, CP, NOCMPRS

> Source listing to be given in compressed form (certain page ejects replaced by 3 line skips). The terminal default is NOCMPRS.

CTIME=(s), CT (on /COMPILE only)

> Time limit for compilation. "s" is seconds and can be an unsigned number with at most two fractional digits. The default is s=(remaining portion of global time limit) if in batch mode and no local time limit is specified, s=(remaining portion of local time limit) if a local time limit is specified, or s=(infinite) if neither of the above is applicable. The CTIME value can never be set higher than the remaining local or global time limit.

DUMP, D, NODUMP, {NO}DUMP=(d1,d2,...), {NO}DUMP=l1l2...
(on /COMPILE only)

Produce post-mortem dump. The dump options are given below. For d1,d2,..., use the single-letter or full-name form. For l1l2..., only the single-letter form may be used.

BLOCKS, B	Traceback of blocks active at termination.
SCALARS, S	Final values of scalar variables in active blocks. (Implies BLOCKS.)
ARRAYS, A	Final values of arrays in active blocks. (Implies SCALAR and BLOCKS.)
FLOW, F	History of last 18 transfers of control if the FLOW condition is enabled.
LABELS, L	List of labels with frequency of encounter.
ENTRIES, E	List of entry names with frequency of call.
REPORT, R	Statistics on run (time, virtual memory use, auxiliary I/O operations, etc.).
UNREAD, U	List of first 5 or fewer unread data cards. (Logical ends-of-file are never read past.)
depth	An integer giving limit on number of active blocks for B, S, and A dump options. If 0 is given, depth is unlimited.
DFLTS, D	Equivalent to S, F, L, E, R, U.

The batch default DUMP options are (S,F,L,E,R,U). The terminal default is NODUMP.

DUMPE, DE, NODUMPE, {NO}DUMPE=(d1,d2,...), {NO}DUMPE=l1l2...
(on /COMPILE only)

Produce post-mortem dump only if error was encountered during execution. The batch default DUMPE options are (S,F,L,E,R,U). The terminal default is NODUMPE.

DUMPT, DT, NODUMPT, {NO}DUMPT=(d1,d2,...), {NO}DUMPT=l1l2...
(on /COMPILE only)

Produce post-mortem dump only if execution was terminated by an error. The batch default DUMPT options are (S,F,L,E, R,U). The terminal default is NODUMPT.

DUMPS, DS, NODUMPS, {NO}DUMPS=(d1,d2,...), {NO}DUMPS=l1l2...
(on /COMPILE only)

Specifies DUMP, DUMPE, and DUMPT. The batch default options are (S,F,L,E,R,U). The terminal default is NODUMPS.

ERRORS=(c,r), E (on /COMPILE only)

Suppress execution if "c" or more compile errors. If c=0, suppress execution unconditionally. Terminate execution after "r" run-time errors. If r=0, there is no limit on run-time errors. The defaults are c=50 and r=50. The maximum for each is 255.

ETIME=(s), ET (on /COMPILE only)

Time limit for execution. "s" is seconds and can be an unsigned number with at most two fractional digits. The default is s=(remaining portion of global time limit) if in batch mode and no local time limit is specified, s=(remaining portion of local time limit) if a local time limit is specified, or s=(infinite) if neither of the above is applicable. The ETIME value can never be set higher than the remaining local or global time limit.

FLAGE, FE, <u>FLAGW</u>, FW

FLAGW prints both warnings and error messages. FLAGE suppresses warnings and prints only error messages.

HDRPG, H, <u>NOHDRPG</u> (on /COMPILE only)

Print header-separator page before program. This option is invalid if PL/C output is being directed to a terminal printer.

ID='name', I (on /COMPILE only)

Program identification name (20 characters maximum). The default name is 'JOB WITH NO ID'.

LINES=n, L (on /COMPILE only)

Maximum number of lines to be printed. The default is n=60*(page limit).

LINECT=n, LC

Lines per page to be printed during compilation. The default is n=60. The PAGESIZE option of the OPEN statement may be used to control the run-time page size.

<u>MCALL</u>, MC, NOMCALL

Print macro calls.

<u>MONITOR</u>, M, NOMONITOR, {NO}MONITOR=(d1,d2,...)
{NO}MONITOR=l1l2...

The MONITOR option specifies that an error message is to be given whenever a program uses a monitored feature. The error will count towards the compile- or run-time error limit, and the standard PL/C correction will be applied. MONITOR options are given below. For d1,d2,..., use the single-letter or full-name form. For l1l2..., only the single-letter form may be used.

BNDRY, B	Monitor strings and comments extending over card boundaries.
UDEF, U	Monitor use of initialized variables. (On /COMPILE only.)
SUBRG, S	Monitor subscripts (i.e., disallow the NOSUBRG condition prefix). (On /COMPILE only.)
AUTO, A	Monitor implied arithmetic/string conversion. (On /COMPILE only.)
DFLTS, D	Equivalent to specifying MONITOR=(B,U,S,A). (On /COMPILE only.)

Only the listed options are altered (i.e., enabled or disabled) <u>except</u> when MONITOR is used on the /COMPILE card. In this case, the designated options are enabled, and all others are disabled. For compatibility with earlier versions of PL/C, the BNDRY, UDEF, and FREE options will be accepted outside of a MONITOR specification.

<u>MTEXT</u>, MT, NOMTEXT

Print macro text expansion. The terminal default is NOTEXT.

M91, M9, <u>NOM91</u> (on /COMPILE only)

Generate code to run on an IBM 360 Model 91. This option is never valid in PL/C.

<u>OPLIST</u>, O, NOOPLIST

Print list of options in effect. The terminal default is NOOPLIST.

PAGES=n, P (on /COMPILE only)

Maximum number of pages to be printed. The default is n=(global page limit)-1 (if in batch and no local page limit supplied), n=(local page limit)-1 (if local page limit specified), and n=32766 (if neither of the above is applicable). The PAGES value can never be reset higher than the remainder, if any, of a local or global MTS time limit which PL/C is running under.

SORMGIN=(s,e), SORMGIN=(s,e,c), SM

Establish source card margins: "s" is first column scanned; the default is s=1. "e" is last column scanned; the default is e=100. "c" is carriage-control column; the default is c=0.

<u>SOURCE</u>, S, NOSOURCE

Print source program listing. The terminal default is NOSOURCE.

TIME=(s), T (on /COMPILE only)

Time limit for compilation and execution. "s" is seconds and can be an unsigned number with at most two fractional digits. The default is s=(remaining portion of global time limit) if in batch mode and no local time limit is specified, s=(remaining portion of local time limit) if a local time limit is specified, or s=(infinite) if neither of the above is applicable. The TIME value can never be set higher than the remaining local or global time limit.

TABSIZE=n, TS (on /COMPILE only)

Determines amount of PL/C region allocated to symbol table. "n" must be given in fullwords. The default is 1/2 of usable area up to 32767 fullwords.

XREF, <u>NOXREF</u>

Produce cross-reference listing for any variables declared and referenced while XREF is in effect.

PL/C is reasonably tolerant in its scanning of the option phrases. Commas are optional and spaces may be added except within the parentheses after ERRORS and SORMGIN; these must be exactly as shown above. The argument of the ID option must be enclosed in primes. The following examples will both yield the same results:

```
/COMPILE I='CORNELL, EZRA' X A T=1.5 P=50

/COMPILE IX='CORNELL, EZRA',,XRFE,ATTRIBUTE TIM=1.5, PAGE=50
```

PL/C is a compile-and-execute system. However, execution can be suppressed by specifying ERRORS=(0,0) on the /COMPILE card. It should also be noted that the "assembly listing" produced when the ALIST option is specified is only an approximation to a true assembly listing of the object code generated by PL/C. (This was intended as an aid in debugging the compiler rather than to show the user exactly what code is executed. The PL/C object code includes certain in-line data structures, and the "assembly listing" does not correctly interpret these in all cases.)

Example Control Card Sequences

(1) Single program without data, with an increased PL/C region size:

```
$RUN *PLC PAR=SIZE=80
     source program cards
```

(2) Single program with data, compiler output directed to other than *SINK*:

```
$RUN *PLC SPRINT=file
/COMPILE options (required only if "options" are specified)
/ATTACH ... (possibly needed for reference to an MTS file)
     source program cards
/OPTIONS options
     more source program cards
/DATA
     data cards
/STOP
```

(3) Program with two external procedures and data:

```
$RUN *PLC (plus any needed I/O device assignments)
/COMPILE options
     source program cards for 1st external procedure
/PROCESS
     source program cards for 2nd external procedure
/OPTIONS options
     2nd external procedure continued
/OPTIONS options
     2nd external procedure continued
/EXECUTE
     data cards
/INCLUDE datafile
     more data cards read from "DATAFILE"
/STOP
```

(4) Three independent programs run in batch mode (back-to-back):

```
$RUN *PLC SCARDS=*SOURCE* SPRINT=*PRINT*
     source program cards for program 1
/INCLUDE sourcefile
     more program source cards included from "sourcefile"
/DATA
```

```
          data cards for program 1
     /COMPILE options (note this card is necessary)
          source program cards for program 2
     /PROCESS
          source program cards for program 2
     /EXECUTE
          data cards for program 2
     /COMPILE options (note this card is necessary)
          source program cards for program 3
     /STOP
```

Input Card Format

The standard field for input records is columns 1 through 100. As in PL/I, an input column may be specified to contain logical carriage-control characters which affect the spacing of the program listing. Only five logical carriage-control codes are recognized for this purpose:

blank	space 1 line before printing (normal mode)
0	space 2 lines before printing
-	space 3 lines before printing
+	do not space before printing (overprint)
1	skip to next page

Carriage-control characters do not appear on the source listing.

The default source card format can be altered by specifying the SORMGIN option on the /COMPILE, /OPTIONS, or /PROCESS card. The form is

```
SORMGIN=(a,b,c)
```

where "a" is the leftmost column to be included, "b" is the rightmost column to be included, and "c" is the column for carriage control.

In MTS, the default is SORMGIN=(1,100,0). This default was chosen to facilitate input from wide-carriage terminals and to disable the sometimes troublesome carriage-control features. The maximum column specification is 100, and the carriage-control column must be outside of the "a,b" field. Control cards are not considered part of the source program and thus are not affected by the SORMGIN option. Control cards must always be formatted as described above.

When the default MONITOR=BNDRY option is in effect, PL/C does not permit any element to be split over a card boundary. That is, keywords, identifiers, constants, and comments cannot start on one card and be continued on the next. This limits the length of string constants that may be included in a PL/C program. It also means that in program documentation with long comments, each card must be a separate comment with an opening /* and a closing */. This is more restrictive than PL/I (F), which will allow very long comments.

When the NOMONITOR=BNDRY option is specified, quoted string-constants and comments may be continued over a card boundary. The maximum length of a string-constant is 256 characters; there is no maximum on the length of a comment. Note that the card boundary is as defined by the SORMGIN option and not the physical card boundary. For example, if a SORMGIN parameter of (2,72,1) were used, column 2 of an input record would be considered to directly follow column 72 of the preceding input record--no blank would be supplied. Note also that the NOMONITOR=BNDRY option applies only to string-constants and comments. One still cannot continue a keyword, an identifier, or an arithmetic constant over a card boundary.

The card field for SCARDS/SYSIN data cards defaults to 1 to 100 (any records shorter will be padded with blanks), but may be redefined by an explicit OPEN. Data cards are not affected by the SORMGIN or BOUNDARY options. Data cards are considered to be a continuous stream of characters and the card boundary is of no significance whatever. That is, the first column of a card directly follows the last of the previous card, and any element may be continued over a card boundary.

DIAGNOSTIC ASSISTANCE

PL/C is unusual, and not compatible with PL/I, in its approach to program testing. The PL/C compiler is "error-correcting" and it includes some options and statements that are not part of PL/I.

Error Correction

When errors are encountered in a PL/C program, an attempt at repair is made, and both the error condition and repair action are announced. Translation is completed for every program, and execution continues until a preset number of errors have been detected. This number may be specified on the /COMPILE card; if specified as ERRORS=(0,0) execution will be suppressed. PL/C will correctly repair some minor syntax and punctuation errors. More importantly, by briefly prolonging the life of an obviously moribund program, it often yields additional diagnostic information which helps to reduce the number of submissions required to achieve satisfactory execution.

One type of PL/C correction that is often successful is spelling correction for keywords. This is attempted only in situations where the context demands a certain type of keyword.

When errors are detected and repaired during compilation, PL/C conveys this information to the programmer by displaying the reconstructed form of the source statement. The usual explanatory messages are also provided, but the reconstruction of the statement that PL/C actually uses often provides the best information as to what was wrong. PL/C will give the error messages only for the first six errors on a particular statement. After the first few errors have been detected, further analysis is heavily dependent upon the corrections that PL/C has attempted. Reporting additional messages is more often confusing than useful. By this point PL/C has usually abandoned the statement and called it "untranslatable." In that case, a null statement is substituted for the faulty source statement.

To enhance this error-correcting capability, two special restrictions have been placed on the source language:

(1) Comments are normally limited to a single card, eliminating the catastrophic confusion of program with comments when a user omits the closing "*/" after a comment. The programmer can override this restriction and process standard long PL/I comments by specifying the NOMONITOR=BNDRY option on the /COMPILE card.

(2) Thirty-nine keywords have been reserved and cannot be used as identifiers. These are the statement keywords and six auxiliary keywords. Although this reservation is not necessary for the analysis of a correct program, the redundancy that it introduces is extremely useful in attempting to unscramble a program that contains numerous errors. These reserved words are "recovery points" for the compiler; whenever it finds a statement to be unintelligible, it can always scan for the beginning of the next statement.

Examples of PL/C Error Correction:

```
STMT 2             DECLARE  ( A1 B1 CHARACTER VARYING ;
IN   2         ERROR  SY06 MISSING COMMA
IN   2         ERROR  SY02 MISSING (
IN   2         ERROR  SY11 MISSING EXPRESSION
IN   2         ERROR  SY04 MISSING )
IN   2         ERROR  SY04 MISSING )
PL/C USES          DECLARE ( A1, B1 CHARACTER (1) VARYING );

STMT 4             P2 PROC ORDER FIXED REORDER ;
IN   4         ERROR  SY09 MISSING :
IN   4         ERROR  SY0F MISSING KEYWORD
IN   4         ERROR  SY02 MISSING (
IN   4         ERROR  SY04 MISSING )
IN   4         ERROR  SY20 IMPROPER OPTION
PL/C USES          P2: PROCEDURE ORDER RETURNS (FIXED);

STMT 9             PUT LIST ( X(I) Y(I) DO I = 1 TO N);
IN   9         ERROR  SY06 MISSING COMMA
IN   9         ERROR  SY02 MISSING (
IN   9         ERROR  SY04 MISSING )
PL/C USES          PUT LIST ((X (I),Y (I) DO I=1 TO N));
```

An interesting one-card PL/I program is the following:

```
 STMT 1             PTU FILE(OUTPUT A+1 'CORRECTION.
 IN   1         ERROR  SY00 MISSPELLED KEYWORD
 IN   1         ERROR  SY1D MISSING EXTERNAL PROC
 IN   1         ERROR  SY3B MISSING LABEL OR ENTRY NAME
 PL/C USES          $L001$: PROCEDURE;

 IN   2         ERROR  SY04 MISSING )
 IN   2         ERROR  SY22 IMPROPER I/O PHRASE
 IN   2         ERROR  SY02 MISSING (
 IN   2         ERROR  SYEB STRING CONSTANT RUNS ACROSS CARD BOUNDARY
 IN   2         ERROR  SY06 MISSING COMMA
 IN   2         ERROR  SY04 MISSING )
 PL/C USES          PUT FILE (OUTPUT) LIST (A+1,'CORRECTION.');

 IN   3         ERROR  SY0E  MISSING END
 PL/C USES          END;

 ERROR SY1C  MISSING MAIN PROC
```

Control of Printed Output

PL/C offers flexible means of controlling the volume of printed output in both compilation and execution. Like PL/I (F), PL/C permits the suppression of the printing of the source program by specifying the NOSOURCE option on the /COMPILE, /OPTIONS, or /PROCESS cards; error messages will still appear. By using SOURCE and NOSOURCE on the /OPTIONS card, it is possible to selectively suppress the printing of program segments.

Similar control of execution output is provided by the PUT ON and PUT OFF statements. These statements control only PUT to FILE(SPRINT/ SYSPRINT). This permits the programmer to suppress the output from portions of program known to be correct while attacking errors in other sections. The PUT ON and PUT OFF statements are dynamic, and take effect as they are encountered in execution.

Tracing and Snapshot Dumping

The most powerful techniques for program debugging are still the classic trace (dynamically following the progress of the program), and dump (periodically displaying the instantaneous status of relevant parts of storage). Unfortunately, most high-level programming languages do not particularly assist the user in either regard. PL/I provides some facilities for this purpose, and PL/C has extended these facilities considerably.

The principal tracing facility of PL/I is the CHECK prefix. Applied to a PROCEDURE or BEGIN block, this lists the names of identifiers to be dynamically monitored during execution of that block. Unfortunately, it is somewhat inflexible (it can only be applied to an entire block), is static rather than dynamic in control, and tends to produce prohibitively voluminous output. PL/C has adopted a different interpretation of the CHECK of an array or structure (see the subsection "Conditions" in "Differences between PL/I (F) and PL/C") in order to reduce the volume of output, and has introduced the CHECK and NOCHECK statements in order to give flexible and dynamic control over the CHECK action. The CHECK and NOCHECK statements are effective only in a block that has a CHECK prefix and affect only the identifiers listed in that prefix. The NOCHECK statement simply suppresses the printing that results from the raising of the CHECK condition, and the CHECK statement resumes the printing. Even finer control over the checking action is provided by means of parameters on the CHECK statement (see "Incompatible PL/C Diagnostic Statements" in "Differences between PL/I (F) and PL/C").

The tracing facility is further extended by the addition of FLOW and NOFLOW prefixes and the FLOW and NOFLOW statements. The prefixes and statements are quite similar to CHECK but are concerned with the flow of control of the program. When the FLOW condition is not disabled, it is raised whenever a statement is encountered that would potentially alter the normal sequential flow-of-control. The statements that raise the condition are CALL, DO, GOTO, IF, and RETURN. In-line procedure reference (function reference) also raises the condition. An exceptional condition (except FLOW for obvious reasons) which would cause an ON-unit to be entered will also raise the FLOW condition. The standard system action is to make an origin-destination entry in a queue whose contents may be displayed by the PUT FLOW statement and the post-mortem dump. Depending upon the appearance of FLOW and NOFLOW statements, the standard system action may instead be to print an origin-destination line on SPRINT/SYSPRINT. This standard action can of course be altered by a user-supplied ON FLOW unit (see "Differences between PL/I (F) and PL/C").

A dumping facility is provided by the addition of nonstandard phrases in the PUT statement:

(1) PUT FLOW; displays the recent flow history.

(2) PUT SNAP; displays the recent "calling" history.

(3) PUT ALL; displays the current values of all automatic scalar variables in the blocks which are active when the statement is encountered.

(4) PUT ARRAY; displays the values of arrays as well as simple variables.

A DEPTH(exp) phrase can be used with any of these except PUT FLOW to limit the nesting depth for which the display is to be given (see "Differences between PL/I (F) and PL/C").

Final Diagnostic Dump

When the execution of a PL/C program is terminated, a "post-mortem dump" is produced according to the options on the /COMPILE card's DUMP, DUMPE, and DUMPT parameters. See the section "PL/C Post-Mortem Dump Statistics Report" for details.

ERROR MESSAGES

In the text of the following messages, the uppercase letters are shown as they appear on the PL/C program listing, but the lowercase words are replaced by variable information as follows:

iden	- a variable or label name will be printed
string	- a character string will be printed
number	- a fixed- or floating-point number will be printed
rtn	- the name of a subroutine will be printed
line	- a statement number will be printed
attribute	- an attribute will be printed

The error message numbers in PL/C are prefixed with a two-letter code indicating the phase in which the error occurred. The codes and their meanings are:

SY or MD	- syntactic analysis phase (MD for errors during macro definitions)
SM	- semantic-analysis phase
XR	- cross-reference phase
CG	- code-generation phase
EX	- execution phase
PM	- post-mortem dump phase

One group of messages, numbered E2-EA, can appear in any phase of the compiler. Hence, these messages will be given with a prefix for the phase in which the error occurs. All of the other messages appear in only one phase and are grouped in the list below by phase and prefix.

A line beginning PROGRAM CHECK or COMPILER ERROR indicates a problem in the PL/C compiler and not a user error (although it most often occurs in response to some user error). Please bring such programs to the attention of the Computing Center staff so that the problem can be remedied.

Variable Prefix (errors that can occur in any phase)

Number	Message
E2	ERROR LIMIT EXCEEDED
E3	LINE LIMIT EXCEEDED
E4	PAGE LIMIT EXCEEDED
E5	TIME LIMIT EXCEEDED
E6	TIME LIMIT EXCEEDED - PROBABLE COMPILER LOOP (The specified time limit has expired and an additional second has passed without the completion of a source statement. This is probably a compiler error, see beginning of this section.)
E7	string (Improper ATTACH card formation. See "Running PL/C in MTS" for the proper format.)
E8	UNABLE TO PROCESS INCLUDE COMMAND (See "Running PL/C in MTS.")
E9	SYMBOL-TABLE OVERFLOW. USE LARGER REGION OR INCREASE TABSIZE. (See TABSIZE option in "Running PL/C in MTS.")
EA	STRING TOO LONG FOR LINE IN ABOVE MESSAGE (OR COMPILER ERROR) (PL/C tried to print an error message containing a string longer than an output line. The message is terminated and execution continues.)

SY or MD Prefix (errors during the syntactic analysis phase)

In some cases, particularly in declarations, errors are discovered too late in the analysis of the statement for PL/C to conveniently correct the text of the statement. The internal form of the program has been altered to a correct construction (often just a null statement), but the usual display of corrected source text is omitted. Messages that frequently are issued in such circumstances are marked with an asterisk (*) in the following list.

Number	Message
00	MISSPELLED KEYWORD (Apparent misspelling of one of the reserved keywords. See list under DECLARE in "Differences between PL/I (F) and PL/C.")
01	EXTRA ((The extra left parenthesis is deleted.)
02	MISSING ((A left parenthesis is supplied.)
03	EXTRA) (The extra right parenthesis is deleted.)
04	MISSING) (Possibly missing operator, a right parenthesis is supplied.)
05	EXTRA COMMA (The extra comma is deleted.)

```
 06       MISSING COMMA
             (A comma is supplied.)
 07       EXTRA SEMI-COLON
             (The extra semicolon is deleted.)
 08       MISSING SEMI-COLON (OR MISUSE OF RESERVED WORD)
             (A semicolon is supplied.  A frequent cause of this error
             is the use of a reserved keyword as an identifier.  This
             forces the start of a new statement so the statement prema-
             turely ended apparently lacks a semicolon.  The user may
             need to choose a nonreserved identifier rather than supply
             a semicolon.)
 09       MISSING :
             (A colon is supplied.)
 0A       MISSING =
             (An equal sign is supplied.)
 0B *     IMPROPER ATTRIBUTE ON PARAMETER
 0C       INEFFECTIVE IF
             (Warning:  A pointless IF statement has been given; the
             THEN unit is null and there is no ELSE unit.)
 0D *     IMPROPER ENTRY/RETURNS ATTRIBUTE
 0E       MISSING END
             (/PROCESS, /DATA, /STOP, or end-of-program encountered with
             a block still open; END supplied.)
 0F       MISSING KEYWORD
             (The required keyword is supplied.)
 10       INCOMPLETE EXPRESSION
 11       MISSING EXPRESSION
 12       MISSING VARIABLE
 13       MISSING ARGUMENT, ONE SUPPLIED
 14       EMPTY LIST
 15       IMPROPER NOT
             (¬ cannot be used as a binary operator; an equal sign "="
             is substituted.)
 16       IMPROPER ELEMENT
             (An element which has appeared in a syntactically
             incorrect position has been discarded.)
 17       IMPROPER SYNTAX, TRANSLATION SUSPENDED
             (The statement has been abandoned and replaced by a null
             statement.  PL/C scans ahead for a semicolon or reserved
             word to start the next statement.)
 18       INCONSISTENT OPTION, STATEMENT DELETED
             (After the statement was completed, it was found to contain
             inconsistent options.  It is deleted and replaced by a null
             statement.)
 19       NOT ENOUGH CORE, TRY LARGER REGION
             (Program will not be executed as is.  The program should be
             run with a larger SIZE parameter.  See "Running PL/C in
             MTS.")
 1A       NESTING TOO DEEP
             (Nesting depth exceeds capacity of PL/C.  See restrictions
             in "Running PL/C in MTS.")
 1B       INACCESSIBLE STATEMENT
             (Warning:  This statement cannot be reached in execution.)
```

1C		MISSING MAIN PROC (Warning: No procedure has OPTIONS(MAIN) phrase; the first procedure is assumed to be MAIN.)
1D		MISSING PROCEDURE STATEMENT (A statement is not contained in an external procedure, or an identifier declared ENTRY appears as a nonentry label. A PROC statement is supplied by PL/C. This condition is sometimes a byproduct of another error; for example, an extra END in the interior of a procedure ends it prematurely and causes subsequent statements to be apparently outside of any procedure.)
1E		MISSING /PROCESS (OR EXTRA END) (Either the required /PROCESS between external procedures has been omitted, or an extra END has prematurely ended a procedure. PL/C supplies a /PROCESS.)
1F		MISPLACED ENTRY STATEMENT (ENTRY cannot be in a BEGIN block or in an iterative DO loop.)
20		IMPROPER OPTION(S) (Improper option on statement, or invalid option on /COMPILE, /PROCESS, or /OPTIONS card. See "Running PL/C in MTS.")
21		IMPROPER FORMAT ITEM
22		IMPROPER I/O PHRASE
23		IMPROPER TO PHRASE
24		IMPROPER BY PHRASE
25		IMPROPER WHILE PHRASE
26		IMPROPER SPECIFICATION (Error in iteration specification of DO statement.)
27		MULTIPLE DECLARATION (This identifier has already been used in a way that precludes its appearance here. In most contexts, the identifier is replaced by a new identifier generated by PL/C.)
28	*	IMPROPER attribute ATTRIBUTE FOR iden (The attribute indicated cannot be applied to this identifier, usually because of previous attributes.)
29		IMPROPER FACTORING
2A	*	IMPROPER DIMENSION
2B	*	IMPROPER PRECISION
2C	*	IMPROPER SCALE
2D	*	IMPROPER VARYING ATTRIBUTE (String type not specified; VARYING deleted.)
2E		IMPROPER FILE-NAME (The identifier in a FILE phrase is not a valid file name.)
2F		EXTERNAL NAME TOO LONG (Warning: PL/I allows a maximum of 7 characters for an identifier in this context.)
30	*	IMPROPER INIT ATTRIBUTE (INITIAL is incompatible with previous attributes.)
31	*	IMPROPER STRUCTURE LEVEL (This may be improper construction of a structure declaration. It also arises from any stray integer in a faulty declaration.)

32	*	IMPROPER ATTRIBUTE IN STRUCTURE (Major and minor structure names cannot have type attributes, leaves cannot have storage class attributes.)
33		TOO MANY IDENTIFIERS (There is a PL/C limit of 88 identifiers in a single factor or structure; the program is not executed.)
34		IMPROPER THEN OR ELSE (THEN or ELSE given on a statement not following an IF.)
35		IMPROPER THEN OR ELSE UNIT (This statement is not allowed as a THEN or ELSE unit. ELSE is deleted; a null statement inserted after THEN.)
36		MISSING THEN (THEN supplied.)
37		IMPROPER CHECK OR NOCHECK (Prefix applied to a statement other than BEGIN or PROCEDURE; prefix deleted.)
38		IMPROPER PREFIX ORDER (Warning: Executes correctly under PL/C but incompatible with PL/I (F).)
39		EXTRA LABEL
3A		IMPROPER LABEL
3B		MISSING LABEL OR ENTRY NAME (A new identifier is generated by PL/C.)
3C		IMPROPER ON-CONDITION (Often triggered by an extra left parenthesis, for example around the left side of an assignment, which is taken to be the beginning of a condition prefix.)
3D		IMPROPER ON-UNIT (This statement is not allowed as a simple ON-unit. BEGIN and END are supplied.)
3E		IMPROPER SPACE ('NO' is improperly separated from rest of keyword; the space is removed. Example: NO CHECK becomes NOCHECK.)
3F		PL/I FEATURE NOT IN PL/C (The feature used is not included in current PL/C.)
40		FEATURES INCOMPATIBLE WITH PL/I (F) HAVE BEEN USED (Warning: An incompatible PL/C feature has been used and is not enclosed in a pseudo-comment. The program will not be accepted by PL/I (F).)
41		INCOMPATIBLE OPTION
42		MISSING OPTION (A "required" option is not given. For example, ENV requires CONSECUTIVE.)
44		MISSING DECIMAL INTEGER
45		NON-* BOUND/LENGTH FIELD (* and not an expression must be given for subscript bound or string length in this context.)
46		DECLARATION FOR ENTRY iden DOES NOT AGREE WITH CORRESPONDING PROC OR ENTRY POINT. DCL IGNORED
47		PROCEDURE iden IS NOT PRESENT ("iden" has been declared as an entry name, but no corresponding procedure definition appears.)
48		*-LENGTH NOT ALLOWED. 256 USED

49	TOO MANY DIGITS IN EXPONENT
4A	ILLEGAL EXPONENT (Illegal character appeared after E. A space is inserted before the E.)
4B	ILLEGAL BINARY NUMBER (The number is treated as decimal.)
4C	ILLEGAL USE OF COLUMN 1 ON CARD (If SM=(2,X,1), the value in column 1 is concatenated with the string beginning in column 2. This error should not appear in MTS-PL/C.)
4D	*/ NOT IN COMMENT (The comment is ignored.)
4E	NAME > 31 CHARACTERS (The first 256 or fewer characters are used. PL/C allows up to 256 characters while PL/I (F) allows only 31 characters and would use the first 16 and last 15 bits.)
4F	ILLEGAL CHARACTER (The character is ignored.)
50	STRING CONSTANT RUNS ACROSS CARD BOUNDARY (A prime is supplied.)
51	IMBEDDED BLANK(S) IN OPERATOR (The blanks are ignored. Example: * * becomes **.)
52	COMMENT RUNS ACROSS CARD BOUNDARY (The comment is terminated at the end of the card.)
53	2 DECIMAL POINTS IN NUMBER (The number is terminated at the second decimal point.)
54	EXPONENT RUNS ACROSS CARD BOUNDARY (The exponent is ignored.)
55	SPACE MISSING BETWEEN NUMBER AND LETTER (The required space is supplied.)
56	MISSING */ BEFORE END OF FILE OR CONTROL CARD
57	INVALID BIT STRING
58	MISSING QUOTE BEFORE END OF FILE OR CONTROL CARD
59	STRING LENGTH > 255
5A	MISPLACED /MEND CARD
5B	ERROR STACK OVERFLOW - MESSAGE(S) LOST
5C	TABSIZE TOO LARGE. DEFAULT USED
5D	OPTION(S) NOT ALLOWED AT THIS INSTALLATION (Option specified on /COMPILE, /PROCESS, or /OPTIONS card is not valid in MTS.)
5E	TOO MANY SIGNIFICANT DIGITS, 16 USED
5F	TOO MANY SIGNIFICANT DIGITS, 53 USED
60	EXPONENT TOO LARGE
60	MACROS NOT ALLOWED. COMPILATION TERMINATED (Will never appear in MTS.)
61	TOO MANY OPERANDS IN CHECK OR FLOW STATEMENT (0 or 10**75 is supplied, as appropriate.)
63	MISSING /MEND BEFORE END-OF-FILE OR CONTROL CARD
64	MISSING MACRO NAME
65	MISSING %; BEFORE END-OF-FILE OR CONTROL CARD
66	MACRO NAME ILLEGAL OR ALREADY IN USE
67	MISSING PARAMETER NAME IN MACRO DEFINITION
68	MACRO PARAMETER NAME > 31 CHARACTERS. FIRST 31 USED.

69	TOO MANY MACRO PARAMETERS. LIST TRUNCATED.
6A	MACRO PARAMETER NAME APPEARS TWICE IN LIST.
6B	SYMBOL TABLE AREA OVERFLOW. INCREASE CORE AVAILABLE. (The SIZE parameter on the $RUN command should be increased. See "Introduction.")
6C	ILLEGAL CHARACTER(S) ON CARD. BLANK(S) USED.
6D	MACRO EXPANSION CAUSES REPRINTING OF ABOVE LINE
6E	DYNAMIC CORE OVERFLOW DURING MACRO EXPANSION. INCREASE REGION. (That is, the SIZE parameter on the $RUN command. See "Introduction.")
70	COMPILER ERROR--ILLEGAL INTERNAL MACRO PARM ID. (Compiler error, see beginning of this section)
71	MISSING (IN MACRO CALL
72	MACRO ARGUMENT > 256 CHARACTERS. FIRST 256 USED.
73	TOO FEW ARGUMENTS IN MACRO CALL. NULL STRING(S) SUPPLIED.
74	MISSING COMMA IN MACRO CALL
75	MISSING) IN MACRO CALL
76	END-OF-FILE OR CONTROL CARD WITHIN MACRO CALL
77	PICTURE SPECIFICATION FOR IDEN TOO LONG; COMPLEX ATTRIBUTE DELETED
78	NUMERIC SPECIFICATION FOLLOWING SIGN IN PICTURE SPECIFICATION
79	IMPROPER NUMERIC SPECIFICATION FOLLOWING V IN PICTURE SPECIFICATION
7A	IMPROPER CHARACTER IN CHARACTER PICTURE SPECIFICATION
7B	MORE THAN ONE SIGN OR CR/CB IN PICTURE SPECIFICATION
7C	MORE THAN ONE V IN PICTURE SPECIFICATION
7D	V IN EXPONENT IN PICTURE SPECIFICATION
7E	MORE THAN ONE E OR K IN PICTURE SPECIFICATION
7F	MISSING EXPONENT FIELD IN PICTURE SPECIFICATION
80	INCOMPLETE CR/DB IN PICTURE SPECIFICATION
81	CR$B USED IN FLOATING PICTURE SPECIFICATION
82	MIXED Z AND * IN PICTURE SPECIFICATION
83	Z OR * FOLLOWS 9,I,R OR T IN PICTURE SPECIFICATION
84	Z OR * FOLLOWS DRIFTING FIELD IN PICTURE SPECIFICATION
85	INVALID Z OR * FOLLOWING V IN PICTURE SPECIFICATION
86	A OR X USED IN NUMERIC PICTURE SPECIFICATION
87	F USED IN FLOATING PICTURE SPECIFICATION
88	CHARACTER(S) FOLLOWING SCALE FACTOR IN PICTURE SPECIFICATION
89	VALUE OF REPETITION FACTOR TOO LARGE IN PICTURE SPECIFICATION
8A	CR/DB OR MISPLACED SIGN IN FLOATING PICTURE SPECIFICATION. EXPONENT DELETED
8B	MORE THAN ONE DRIFTING FIELD IN PICTURE SPECIFICATION
8C	PL/I RESTRICTS THE USE OF S + - $ IN FLOATING PICTURE SPECIFICATION
8D	TOO MANY $'S IN PICTURE SPECIFICATION
8E	ILLEGAL PICTURE SPECIFICATION. SEE STMT LINE
8F	SCALE FACTOR IS < -128 OR > 127 IN PICTURE SPECIFICATION
90	NO DIGITS SPECIFIED IN NUMERIC PICTURE SPECIFICATION
91	MORE THAN 15 DIGITS SPECIFIED IN FIXED NUMERIC PICTURE SPECIFICATION
92	MORE THAN 16 DIGITS SPECIFIED IN FLOAT NUMERIC PICTURE SPECIFICATION

93	EXPONENT MORE THAN 2 DIGITS LONG IN PICTURE SPECIFICATION
94	TOO MANY DIGITS IN SCALE OR REPETITION FACTOR IN PICTURE SPECIFICATION
95	NON-NUMERIC CHARACTER IN SCALE OR REPETITION FACTOR IN PICTURE SPECIFICATION
96	INVALID CHARACTER IN PICTURE SPECIFICATION
97	PICTURE SPECIFICATION IS TOO LONG

SM Prefix (errors during the semantic analysis phase)

When statements in error are reconstructed during the semantic analysis phase, an additional line is printed, labeled DECLARED IN BLOCK. This line specifies the block in which each variable has been declared.

Number	Message
40	VARIABLE NOT PERMITTED (There must be a constant in this context.)
41	WRONG TYPE FOR EXPRESSION (Expression types are arithmetic, string, label, or file, and the wrong one has been used here.)
42	WRONG STRUCTURE OR DIMENSIONALITY FOR EXPRESSION (A scalar needed where an array or matching structure has been used.)
43	ILLEGAL SUBSCRIPTING (Subscripts are not allowed in certain contexts, e.g., GET DATA.)
44	ILLEGAL USE OF PSEUDO-VARIABLES (E.g., CHECK prefixes, GET/PUT DATA.)
45	NAME NEEDED (A name is needed in this context, e.g., initializing label constants.)
46	ENTRY-NAME NEEDED (CALL must have entry name.)
47	NO STRUCTURE APPEARED (No structure appeared in a BY NAME assignment.)
48	STRUCTURES DO NOT MATCH (Structures do not match in a BY NAME assignment.)
49	FUNCTION ARGUMENTS MISSING
4A	OPERAND OF BINARY OPERATOR string HAS IMPROPER TYPE
4B	OPERANDS OF BINARY OPERATOR string DISAGREE IN TYPE, STRUCTURE OR DIMENSIONALITY
4C	OPERAND OF UNARY OPERATOR string HAS IMPROPER TYPE
4D	SUBSCRIPT number OF iden NOT NUMERIC
4E	iden HAS TOO MANY SUBSCRIPTS. SUBSCRIPT LIST DELETED
4F	iden HAS TOO FEW SUBSCRIPTS. SUBSCRIPT LIST DELETED
50	NAME NEVER DECLARED, OR AMBIGUOUSLY QUALIFIED (Expression replaced or CALL deleted.)
51	SUBSCRIPT number OF iden NOT SCALAR
52	iden HAS TOO MANY ARGUMENTS. FUNCTION REFERENCE DELETED
53	ARGUMENT number OF FUNCTION iden DISAGREES WITH CORRESPONDING PARAMETER
54	iden HAS TOO FEW ARGUMENTS. FUNCTION REFERENCE DELETED

55	ARGUMENT number OF FUNCTION iden WAS *. ILLEGAL ARGUMENT
56	TABLE OVERFLOW. EXPRESSION DELETED (Processing expression, try larger SIZE parameter value.)
57	TABLE OVERFLOW. EXPRESSION DELETED (Processing expression skeleton, try larger SIZE parameter value.)
58	TABLE OVERFLOW. EXPRESSION DELETED (Processing expression tree, try larger SIZE parameter value.)
59	TABLE OVERFLOW. EXPRESSION DELETED (Processing entry parameter, try larger SIZE parameter value.)
5A	iden HAS WRONG # OF SUBSCRIPTS (Wrong number of subscripts in a BY NAME assignment; structures do not match.)
5B	MISMATCHED DIMENSIONALITY (Mismatched dimensionality in a BY NAME assignment; structures do not match.)
5C	ILLEGAL LABEL VARIABLE iden (Subscripted label not declared in block.)
5D	ILLEGAL ASSIGNMENT TARGET
5E	ASSIGNMENT SOURCE INCOMPATIBLE WITH TARGET
5F	MAJOR STRUCTURE NAME NEEDED
60	DEFAULT ATTRIBUTES FOR ENTRY NAME iden CONFLICT WITH RETURNS OPTION IN STMT line
61	iden IS ASSUMED A VARIABLE, NOT A BUILT-IN FUNCTION

<u>XR</u> <u>Prefix</u> (errors during the cross-reference phase)

<u>Number</u>	<u>Message</u>
62	NOT ENOUGH CORE FOR CROSS-REFERENCE (The program should be run with a larger SIZE parameter. See "Running PL/C in MTS.")
63	CROSS REFERENCE ABBREVIATED DUE TO LACK OF SPACE (The program should be run with a larger SIZE parameter. See "Running PL/C in MTS.")
64	COMPILER ERROR IN XREF PHASE--INVALID STATEMENT CODE (A compiler error. See beginning of "Error Messages.")

<u>CG</u> <u>Prefix</u> (errors during the code-generation phase)

<u>Number</u>	<u>Message</u>
00	FORMAT WILL BE EXECUTED ONLY ONCE (The format specification of the EDIT statement does not contain any format items which would cause data to be transferred between the I/O list and the I/O buffer, i.e., no A, B, C, E, F, or R format item. If run under PL/I (F), the program would loop.)
01	CONSTANT BOUND, LENGTH, SUBSCRIPT OR ITERATION FACTOR EXCEEDS 32767 IN MAGNITUDE. 10 IS USED (PL/I language restriction.)

02	WORKSPACE OVERFLOW IN STATEMENT PROCESSING (The combined nesting of BEGIN and PROCEDURE blocks, iterative DO groups, and IF statements is too deep for the code-generation phase. The rest of the program is not scanned for code-generation errors. Increasing the SIZE parameter will not help. The nesting depth must be reduced.)
03	iden REQUIRES TOO MUCH SPACE. UPPER BOUND OF SUBSCRIPT number IS SET TO LOWER BOUND (More than 2**31 bytes would be required for the array as declared.)
04	PRIMARY DATA STORAGE AREA FOR BLOCK # number EXCEEDS SIZE LIMIT BY number BYTES (Primary data storage does not include space for arrays or strings. Try adding some more BEGIN blocks.)
05	LENGTH OF iden (number) IS NOT IN PROPER RANGE. 80 IS USED (Length is < 0 or > 256.)
06	iden REQUIRES TOO MUCH SPACE. LOWER BOUND OF SUBSCRIPT number IS SET TO ZERO (The array element with all subscripts zero must be within 2**31 bytes of the array element with all subscripts at their lower bound. The lower bounds should be moved closer to zero.)
07	ARITHMETIC FIRST ARGUMENT TO SUBSTR PSEUDO-VARIABLE. A STRING TEMPORARY IS USED (Arithmetic argument remains unchanged.)
08	SEVERE ERRORS. EXECUTION SUPPRESSED. (A previous code-generation error has made it impossible to continue into execution. All code-generation errors have been reported.)
09	CONVERSION REQUIRED TO MATCH ARGUMENT iden OF iden (Warning: PL/C has generated code to convert the argument of a procedure call so that the attributes of the value passed will match the attributes of the corresponding parameter. PL/I (F) would not do this conversion because the attributes of the parameter have not been specified in an ENTRY declaration.)
0A	SCALAR ARGUMENT SUPPLIED TO AGGREGATE PARAMETER iden OF iden. ((1:10) USED FOR ALL BOUNDS) (This is a PL/C restriction. See "Differences between PL/I (F) and PL/C." Assign the constant to an array with the proper bounds and pass that array to the procedure.)
0B	WORKSPACE OVERFLOW IN EXPRESSION PROCESSING (Either the situation which would generate error CG02 exists or the nesting of array expressions, array subscripting, function references, or parenthesized expressions is too deep. Simplify the expression. Increasing the SIZE parameter will not help.)
0C	NO FILE SPECIFIED. SYSIN/SYSPRINT ASSUMED (A warning message.)
0D	iden IS A PARAMETER IN I/O LIST OR CHECK PREFIX (Warning: PL/I (F) does not allow parameters in DATA-directed I/O lists nor in CHECK prefixes. PL/C will accept the parameter.)

0E	BOTH FORMS OF INITIALIZATION USED FOR LABEL VARIABLE iden (PL/I (F) does not permit a LABEL variable to be initialized via both the INITIAL attribute and subscripted statement label constants. Both forms are accepted by PL/C. Where there is conflict the INITIAL attribute takes precedence.)
0F	STORAGE CAPACITY IS EXCEEDED (Object code exceeds available space, rerun with a larger SIZE parameter. See "Running PL/C in MTS.")
10	ILLEGAL COMPLEX COMPARE. REAL PARTS WILL BE COMPARED
11	iden IS ILLEGAL OPERAND IN INITIAL, LENGTH OR DIMENSION ATTRIBUTE OF STATIC VARIABLE constant IS USED. (The bounds, lengths, and iteration factors used with a STATIC or EXTERNAL variable must be optionally signed decimal constants. A nonconstant has appeared in this context and has been replaced by a constant of appropriate type.)
12	NON-CONSTANT OPERAND(iden) IN INITIAL, LENGTH OR DIMENSION ATTRIBUTE OF STATIC VARIABLE (Warning: A STATIC or EXTERNAL variable, BUILT-IN function, or EXTERNAL user-defined function has been used in the bounds, length, or iteration factor for a STATIC/ EXTERNAL variable. This is not allowed in PL/I. PL/C uses the value of the operand in error.)
13	PL/C BUILT-IN FUNCTION USED (Warning: A built-in function has been used that is not included in PL/I (F).)
14	ARGUMENT TO MAX OR MIN IS COMPLEX. REAL PART IS USED
15	NO SCALE FACTOR ARGUMENT APPEARED. RESULT IS SET FLOAT (See explanation of error CG16.)
16	UNNECESSARY SCALE FACTOR ARGUMENT APPEARED. RESULT IS SET FIXED (For ADD, DIVIDE, or MULTIPLY, both a precision argument (P) and a scale factor argument (Q) must be present if the result is to have FIXED scale. Only argument P may appear if the result is to have FLOAT scale. If either requirement is violated, PL/C converts the argument to the scale implied by the number of arguments given.)
17	ARGUMENT SHOULD BE A CONSTANT. 10 IS USED (Certain arguments to the built-in functions ADD, BINARY, DECIMAL, DIVIDE, FIXED, FLOAT, MULTIPLY, PRECISION, and ROUND must be decimal constants in PL/I (F).)
18	ABS(ARGUMENT) > 32767. 10 IS USED (Constant arguments to built-in functions mentioned in explanation of error CG17 must be less than 32768.)
19	ARGUMENT SHOULD BE REAL. IMAGINARY PART IS USED (Constant of the form "nI" appeared where real constant was required. The "nI" is ignored.)
1A	ILLEGAL COMPLEX ARGUMENT. REAL PART IS USED
1B	ILLEGAL ARGUMENT TO BUILT-IN FUNCTION. SHOULD BE REAL, FIXED DECIMAL CONSTANT (Warning: PL/I (F) requires that certain arguments of the built-in functions BIT, CHAR, HIGH, LOW, and REPEAT be unsigned decimal constants. PL/C will take the argument as

written.)

1C	RESULT SCALE FACTOR = number > 127 IN MAGNITUDE. RESULT SCALED INCORRECTLY TO 127*SIGN(number) (Following the rules for PL/I expression evaluation, the scale factor (Q) of the result would be outside the permitted range -127 to 127. So that execution may be attempted, the result is scaled to the closest bound of the legal range. The value of the result will be incorrect.)
1D	PROGRAM MAY LOOP IF THIS FORMAT IS EXECUTED (The FORMAT statement does not specify a data transmission format item. See explanation of error CG00.)
1E	VARIABLE iden HAS A * BOUND OR LENGTH FIELD. 10 IS USED (Only parameters in PL/C may have * bound or length.)
1F	PARAMETER iden HAS A NON-* BOUND OR LENGTH FIELD (Parameters must have an * in this field in PL/C)
20	LOWER BOUND OF SUBSCRIPT number OF iden EXCEEDS UPPER BOUND. (0:10) IS USED
21	SPECIFIED P(number) TOO LARGE. MAX PRECISION IS USED (Hardware maximum precision, i.e., 31 for FIXED BINARY, 15 for FLOAT DECIMAL.)
22	STRING ARGUMENT TO COMPLEX PSEUDO-VARIABLE (The assignment is performed anyway.)
23	TOO MANY ERRORS DURING COMPILATION. EXECUTION SUPPRESSED
24	COMPILER ERROR DURING CODE GENERATION. PROGRAM ABORTED (A compiler error, see beginning of this section.)
25	ILLEGAL ARGUMENT TO REAL OR IMAG PSEUDO-VARIABLE (The assignment is performed anyway. Argument must be complex arithmetic.)
26	IMPLIED ARITHMETIC-TRING CONVERSION INVOKED (MONITOR message. Conversion is performed.)
27	STRING CONSTANT IN INITIAL, LENGTH OR DIMENSION ATTRIBUTE OF STATIC VARIABLE (Conversion is performed.)
28	BIT STRING IN GET OR PUT STRING. STATEMENT DELETED

EX Prefix (errors during the execution phase)

No	ON Code	Message
00	0004	PROGRAM RETURNS FROM MAIN PROCEDURE
01	0004	PROGRAM IS STOPPED (Normal termination; a STOP or EXIT has been executed.)
02	0070	END OF FILE REACHED (The ENDFILE condition is raised. System action terminates the program.)
03	0300	EXPONENT OVERFLOW. RESULT IS SET TO 1
04	0300	EXPONENT OVERFLOW. RESULT IS LEFT UNCHANGED
05	0310	FIXED-POINT OVERFLOW (Low-order digit set to 1.)
06	0310	FIXED-DECIMAL OVERFLOW
07	0310	NUMBER TOO LARGE TO CONVERT TO FIXED BINARY. 1 IS USED
08	0320	FIXED-POINT QUOTIENT TOO LARGE. PROBABLE DIVISION BY 0. RE-

```
         SULT IS SET TO 0
 09 0320  FIXED-POINT QUOTIENT TOO LARGE.  PROBABLE DIVISION BY 0.  RE-
         SULT IS LEFT UNCHANGED
 0A 0320  FLOATING-POINT DIVISION BY 0.  RESULT IS SET TO 1
 0B 0320  FLOATING-POINT DIVISION BY 0.  RESULT IS LEFT UNCHANGED
 0C 0330  EXPONENT UNDERFLOW.  RESULT IS SET TO 0
 0D 0330  EXPONENT UNDERFLOW.  RESULT IS LEFT UNCHANGED
 0E 0340  SIZE RAISED.  RESULT IS LEFT UNCHANGED
            (Occurs when the value of an expression is assigned to a
            variable whose precision is too small to hold the value.
            In PL/C, no left-truncation occurs.  Instead the computed
            value is assigned to the variable, regardless of its de-
            clared precision.)
 0F 0340  SIZE RAISED DURING CONVERSION.  RESULT IS SET TO 0
 10 0340  SIZE RAISED DURING STRING-TO-ARITHMETIC CONVERSION.  VALUE
         USED IS number
 11 0340  NUMBER TOO LARGE TO CONVERT TO SPECIFIED BIT STRING.  (SIZE
         CONDITION) NUMBER IS number STRING USED IS string
 12 0340  RESULT OF BIT-TO-ARITHMETIC CONVERSION GREATER THAN 2**56-1.
         (SIZE CONDITION) STRING IS string VALUE USED IS number
 13 0341  NUMBER TOO LARGE FOR FIELD.  TRUNCATED ON LEFT.  FULL FIELD
         WOULD BE string
            (In a PUT statement, the value is too large to fit in the
            specified field [for EDIT] or the field implied by the
            attributes of the item [for LIST].  Signs and digits are
            lost on the left as in PL/I.  The message indicates the
            full field before truncation.)
 14 0350  INDEX OF SUBSTRING < 1 (number)
            (Second argument of SUBSTR is less than one.)
 15 0350  INDEX OF SUBSTRING > STRING LENGTH (number)
            (Second argument of SUBSTR is greater than the length of
            the first argument.)
 16 0350  LENGTH OF SUBSTRING < 0 (number)
            (Value of third argument of SUBSTR is negative.  It is re-
            placed by 0.)
 17 0350  SUBSTRING REQUESTED RUNS OVER END OF STRING
 19 0520  SUBSCRIPT number OF iden IS OUT OF BOUNDS (number).  number IS
         USED
 1A 0602  TOO MANY CHARACTERS FOLLOWING CLOSING QUOTE.  ALL ARE IGNORED.
         FIELD IS string
 1B 0603  TOO MANY DIGITS IN NUMBER, PRECISION LOST.  STRING IS string
 1C 0604  TOO MANY EXPONENT DIGITS, EXTRA DIGITS IGNORED.  STRING IS
         string
 1D 0605  INVALID CHARACTER(S) IN FIELD.  0 USED FOR EACH.  ORIGINAL
         STRING IS string.  FIRST BAD CHARACTER IS string
            (The CONVERSION condition has been raised.)
 1E 0615  ILLEGAL CHARACTER(S) IN CHARACTER-TO-BIT CONVERSION.  0'S USED
 1F 0900  ATTEMPT TO USE MATH BUILT-IN FUNCTION IN "CALL" STATEMENT.
         STATEMENT IGNORED
 20 0901  iden REFERENCED RECURSIVELY.  "RECURSIVE" ATTRIBUTE HAS NOW
         BEEN APPLIED
            (Indicated PROCEDURE is being used recursively but did not
            have RECURSIVE option.)
```

21	0902	iden HAS IMPROPER LENGTH (number). 80 IS USED (Length is less than zero or greater than 256 and violates a PL/C restriction.)
22	0903	LOWER BOUND ON SUBSCRIPT number OF iden EXCEEDS UPPER BOUND. (1:10) IS USED (Expressions for array bounds are evaluated before any statements in the block in which the array is declared are executed. Variables used in these expressions must be initialized in an outer block.)
23	0904	RETURN FROM iden VIA STMT line DOESN'T RETURN A VALUE AS EXPECTED IN STMT line. 0 IS USED
24	0905	RETURN FROM iden VIA STMT line REQUIRES ILLEGAL CONVERSION. BLANKS OR 0 IS USED (PL/C restriction; PL/I (F) would convert.)
25	0906	RETURN FROM iden VIA STMT line RETURNS A VALUE TO "CALL" IN STMT line. VALUE IGNORED (Results would be unpredictable in PL/I (F).)
26	0907	CALL TO iden FROM STMT line RETURNS VIA STMT line WITH STRING LONGER THAN DECLARED LENGTH. RETURNED LENGTH IS USED
27	0908	CALL TO iden FROM STMT line RETURNS VIA STMT line WITH STRING SHORTER THAN DECLARED LENGTH. IT IS PADDED
28	0909	BOUNDS OF iden DO NOT MATCH BOUNDS IN THE REST OF THE EXPRESSION. (Execution is terminated.)
29	0910	iden HAS NOT BEEN ALLOCATED (In a procedure invoked to initialize a variable, a reference has been made to an array, structure, or string which has not been allocated space. Variables are allocated and initialized in the order in which they are declared. See "Order of Evaluation in DECLARE Statements" in "Differences between PL/I (F) and PL/C.")
2A	0911	FORMAT LABEL IN GOTO (Execution is terminated.)
2B	0912	VALUE OF LABEL VARIABLE (iden IN STMT line) IS IN A CURRENTLY INACTIVE BLOCK (Execution is terminated.)
2C	0913	iden INVOKED FOR INITIALIZATION IN STMT line TERMINATES VIA GOTO (Execution is terminated.)
2D	0914	iden IN STMT line IS IN A CURRENTLY INACTIVE ITERATIVE DO GROUP (Execution is terminated.)
2E	0915	SECOND ARGUMENT OF BIT/CHAR IS NOT POSITIVE. IMPLIED LENGTH IS USED
2F	0916	STRING > 256 CHARACTERS LONG (PL/C limitation. Only the 256 leftmost characters are retained.)
30	0917	ATTEMPT TO ASSIGN INVALID BIT STRING TO FIXED-DECIMAL DATA ITEM. 0 IS USED (May occur in UNSPEC pseudo-variable, or during a READ statement. See "Built-in Functions and Pseudo-variables" in "Differences between PL/I (F) and PL/C.")
31	0918	UNDEFINED ENTRY. STATEMENT IGNORED

```
                (The procedure $UENTRY, supplied by PL/C to repair some
                semantic error, has been referenced.)
 32 0919  DELETED STATEMENT ENCOUNTERED
                (This message is produced during execution of a program
                when a statement deleted by an earlier phase of the compil-
                er is encountered.)
 33 0920  UNDEFINED LABEL IN GOTO
                (Execution is terminated.)
 34 0921  UPPER BOUND ON SUBSCRIPT FOR iden > 32767 IN MAGNITUDE.  10 IS
          USED
 35 0922  LOWER BOUND ON SUBSCRIPT FOR iden > 32767 IN MAGNITUDE.  1 IS
          USED
 36 0923  LABEL COUNTER OVERFLOW.  IT IS RESET TO 0
                (Warning:  A labeled statement has been executed more than
                10 million times causing an internal PL/C counter to over-
                flow.  This may indicate a loop in the program.)
 39 0925  RECORD I/O STRUCTURE VARIABLE iden CONTAINS VARYING STRINGS.
          MAXIMUM LENGTHS ARE USED
                (Only fixed length strings may be members of structures
                used by RECORD I/O.)
 39 0926  INVALID PARAMETER REFERENCE (OR COMPILER ERROR)
                (A parameter has been referenced which was not in the pa-
                rameter list of the entry point used to call the
                procedure.)
 3A 0927  ATTEMPT TO USE AUTOMATIC ARITHMETIC-STRING CONVERSION
                (An arithmetic variable or expression in an I/O list has
                been associated with a string format item or string data,
                or vice-versa.  MONITOR message.  Conversion is performed.)
 3B 0928  OUTPUT STRING TOO LONG.  FIRST 32767 CHARACTERS USED
 3C 0929  INVALID BLANK FIELD IN GET EDIT.  0 IS USED
 3D 0930  DIMENSION SPECIFIED IN HBOUND, LBOUND OR DIM < 1.  1 IS USED
 3E 0931  DIMENSION SPECIFIED IN HBOUND, LBOUND OR DIM > MAXIMUM.  MAXI-
          MUM IS USED
 3F 1002  ATTEMPT TO WRITE OVER END OF STRING.  STATEMENT TERMINATED
 40 1002  ATTEMPT TO READ OVER END OF STRING.  STATEMENT TERMINATED
 41 1018  CLOSING QUOTE MISSING IN INPUT FIELD:  string QUOTE SUPPLIED
 42 3798  ONSOURCE/ONCHAR PSEUDO-VARIABLE USED OUT OF CONTEXT
                (ONCHAR or ONSOURCE may be changed by the program only when
                they have been set to point to a string in error at the
                time the CONVERSION condition arises.  At other times an
                attempt to change [assign to] either is an error.)
 43 3799  IMPROPER RETURN FROM CONVERSION ON-UNIT.  SOURCE IS string
                (The CONVERSION ON-unit did not change the character which
                was in error.)
 44 0936  FEATURE NOT AVAILABLE IN THIS RELEASE
                (PL/I (F) feature used that is not implemented in the cur-
                rent release of PL/C.)
 45 0937  iden IS AN ILLEGAL FORMAT LABEL
                (The label referenced by the R(label) format item is ille-
                gal.  This may be:
                    (a) because it is not the label of a FORMAT statement,
                        or
                    (b) because it labels a statement internal to some block
                other than the block containing the R(label).
         The remote format item is ignored.)
 46 0010  string IS AN ILLEGAL NAME
         (Something other than an identifier was read during a GET
         DATA statement, where an identifier should have appeared.
         The NAME condition is raised.)
 47 0938  INVALID FORMAT OPTION
         (An option in the format used with a GET or PUT EDIT state-
         ment appeared in an illegal context:
            (a) A or B format:  appeared without a field-width pa-
                rameter on input,
            (b) COLUMN format:  appeared without a target column pa-
                rameter, or was used in GET/PUT STRING statement,
            (c) F format:  appeared without a field-width parameter,
            (d) LINE format:  appeared without a target line
                parameter,
            (e) LINE or PAGE:  was used in a GET statement, a PUT
                STRING, or a PUT FILE(X) where X was not a PRINT
                file,
            (f) P format:  appeared without a valid PICTURE
                specification,
            (g) SKIP format:  was used in a GET/PUT STRING
                statement,
            (h) X format:  appeared without a field-width parameter.
         The format item and corresponding list item are dropped.)
 48 0939  INVALID FORMAT ITEM OPERAND
         (In formats E(W,Q), F(W,Q), or F(W,Q,P), either:
            (a) a negative Q appeared on input, or
            (b) on output, either 0>W, W>255, 0>Q, or Q>W
         The format item and corresponding list item are skipped.)
 49 0010  string IS NOT KNOWN TO PROGRAM
         (In a GET DATA statement, the name on the data card has not
         been used in the program.  The NAME condition is raised.
         The data-card assignment is skipped.)
 4A 0010  INCOMPATIBLE STRUCTURE FOR iden
         (In a GET DATA statement, a name in the input was quali-
         fied, although it was declared without substructures, or an
         unqualified name appeared in the data, although it was de-
         clared as a structure in the program.  The NAME condition
         is raised.  The data-card assignment is skipped.)
 4B 0010  iden IS NOT IN GET LIST
         (In a GET DATA statement, a name appeared in the input
         which was not in the data list.  This error can arise for a
         qualified name if its first identifier (major structure
         identifier) is not in the data list.  The NAME condition is
         raised.  The data-card assignment is skipped.)
 4C 0010  ARRAY ERROR FOR iden
         (In a GET DATA statement, subscripts appeared on a name in
         the input, but the name was not declared as an array and
         may not be subscripted.  The data-card assignment is
         ignored.)
 4D 0520  string BOUND ERROR.  number IS USED
         (In a GET DATA statement, a subscript on a name in the in-
         put is out-of-bounds.  The upper or lower bound is used, as
         indicated.)
 4E 0010  NO BOUNDS SPECIFIED FOR iden
         (In a GET DATA statement, no subscript appeared in the in-
         put following an array name.  The data-card assignment is
         ignored.)
 4F 0081  CONFLICTING FILE ATTRIBUTES SPECIFIED OR IMPLIED.  CODE=number
         (The codes are:
            0:  PUT to RECORD file (SYSPRINT will be used, if
                possible).
            1:  GET from OUTPUT or RECORD file.
            2:  More than one of INPUT, OUTPUT, UPDATE specified.
            3:  STREAM file specifying non-CONSECUTIVE organization.
            4:  Both RECORD and STREAM specified.
            5:  Both DIRECT and SEQUENTIAL specified.
            6:  Both DIRECT and CONSECUTIVE specified.
            7:  SEQUENTIAL CONSECUTIVE and KEYED specified.
            8:  DIRECT OUTPUT and INDEXED specified.)
 50 0084  FILE CANNOT BE OPENED.  CODE=number
         (The MTS-PL/C interface refuses to open a file.  The codes
         are:
            0:  PL/C workspace overflow during file open buffer al-
                location.  Try using a larger SIZE parameter value
                on the $RUN command.
            1:  BLKSIZE not a multiple of LRECL (RECFM=F or FB).
            3:  Attempt to open unassigned device or nonexistent
                file.
            5:  Unable to retrieve blocking information from a mag-
                netic tape.
            6:  No LRECL, BLKSIZE, or LINESIZE specified.
            7:  Spanned records not supported.
            8:  BLKSIZE not large enough (RECFM=V or VB).
           10:  Input file assigned to punch.
           11:  Invalid logical device name.
           12:  Invalid physical device type.
           13:  Auxiliary I/O not permitted.
           15:  File not RECFM=F or V.
           16:  Key length not specified.
           17:  Only one buffer is allowed for a DIRECT file.
           18:  RKP out of range (too high or less than 4 for V for-
                mat file).
           19:  DELETE option cannot be specified with the key occu-
                pying first byte on the record.)
 51 0932  SYSTEM DATA SET CANNOT BE RE-ALLOCATED WHILE OPEN UNDER AN-
          OTHER FILE
         (An attempt has been made to open SCARDS/SYSIN or SPRINT/
         SYSPRINT while it is open under another filename.)
 52 0933  FILE NOT OPENED IN UNDEFINEDFILE ON-UNIT
 53 0934  INVALID ARGUMENT TO LINENO.  iden NOT A PRINT FILE
 54 0935  INVALID ARGUMENT TO COUNT.  iden NOT A STREAM FILE
 55 1004  string OPTION INVALID.  FILE DOES NOT HAVE "PRINT" ATTRIBUTE
         (The LINE and PAGE options are invalid in a PUT FILE(X),
         unless X has the PRINT attribute.  The option is ignored.)
 56 1009  FILE CANNOT BE USED FOR STREAM INPUT
             (File is open as a RECORD or OUTPUT file.)
 57 1009  FILE CANNOT BE USED FOR STREAM OUTPUT
             (File is open as a RECORD or INPUT file.  SYSPRINT will be
             used if possible.)
 58 1009  FILE CANNOT BE USED FOR RECORD I/O
             (File is open for STREAM I/O.)
 59 1009  FILE CANNOT BE USED FOR INPUT
             (File is open for OUTPUT.)
 5A 1009  FILE CANNOT BE USED FOR OUTPUT
             (File is open for INPUT.)
 5B 1009  I/O STATEMENT AND/OR OPTIONS INCOMPATIBLE WITH FILE
          CODE=number
             (Given for special RECORD I/O options.  Codes are:
                0:  KEY/KEYTO/KEYFROM specified for non-KEYED file.
                1:  "KEY" not valid for this type of file.
                2:  "KEY" or "KEYFROM" required.
                3:  Other incompatibility.)
 5C 0940  COMPILER ERROR - NO NAME FOR UNINITIALIZED VARIABLE AT OFFSET
          number
             (May occur if SUBSCRIPTRANGE is disabled and an uninitia-
             lized value is referenced.  If this occurs when SUBSCRIPT-
             RANGE is enabled, it is a compiler problem.)
 5D 0941  iden HAS NOT BEEN INITIALIZED.  IT IS SET TO string
 5E 0942  FORMAT iden HAS INVALID CONDITION PREFIXES
             (The conditions in effect for a FORMAT statement must be
             the same as those in effect for the EDIT statement which
             references the FORMAT statement.  The conditions on the
             FORMAT statement are ignored and execution continues.)
```

Note: For errors EX5F through EX6D, see the IBM publication, IBM System 360 PL/I Subroutine Library, Computational Subroutines, form number GC28-6590, for the exact formulas used.

```
 5F 1509  rtn ABS(X) ≥ (2**50)*K; FOR TAN(X), K=PI.  FOR TAND(X), K=180.
          RESULT IS SET TO 1
             (Issued by TAN or TAND.  TAN(X) is called directly by TANH
             (A+BI) and TAN(A+BI).  The argument is too large in abso-
             lute value.)
 60 1513  rtn ABSOLUTE VALUE OF REAL ARGUMENT (number) IS > 175.366.
          RESULT IS SET TO 1
             (Issued by SINH or COSH.)
 61 1507  rtn ARGUMENT(number) IS GREATER THAN PI*2**50 = .3537E+16.
          RESULT IS SET TO 1
             (Issued by COS or SIN.  COS(X) and/or SIN(X)
             are called by COSD, SIND, SIN(A+BI), COS(A+BI),
             SINH(A+BI), COSH(A+BI), and EXP(A+BI).)
 62 1501  rtn ARGUMENT (number) IS NEGATIVE.  RESULT IS SET TO
          SQRT(ABS(ARG))
             (Issued by SQRT(X).  SQRT(A**2+B**2) is used to calculate
             ABS(A+BI), and various real SQRT calls are made in calcu-
             lating SQRT(A+BI).  In these indirect cases, message EX8D
             should not occur, but calculation errors might produce it.)
 63 1511  rtn BOTH ARGUMENTS ARE 0.  RESULT IS SET TO 1
             (Issued by ATAN(Y,X) or ATAND(Y,X).  ATAN(Y,X) is used in
             calculating LOG(A+BI), ATAN(A+BI), and (A+BI)**(C+DI).)
 64 1505  rtn ARGUMENT(number) ≤ 0.  RESULT IS SET TO 1
             (Issued by LOG(X).  LOG(X) is called to compute LOG2(X),
             LOG10(X), (A+BI)**(C+DI), LOG(A+BI), and ATANH(Y).  ATANH
             (Y) is in turn used in ATAN(A+BI) and ATANH(A+BI).)
 65 1559  rtn Z=+I OR -I IN ATAN(Z) OR Z=+I OR -I IN ATANH(Z).  RESULT
          IS SET TO 1+0I
             (Issued by ATAN(A+BI) or ATANH(A+BI).)
 66 1515  rtn ABSOLUTE VALUE OF ARGUMENT IS ≥ 1.  RESULT IS SET TO 1
             (Issued by ATANH(X).  ATANH(X) is used in calculating
             ATANH(A+BI) and ATAN(A+BI).)
 67 1557  rtn Z1=0 AND IMAG(Z2) = 0 OR REAL(Z2) ≤ 0.  RESULT IS SET TO 1
             (Issued by (A+BI)**(C+DI).)
 68 1556  rtn IN COMPLEX EXPONENTIAL FUNCTION REAL ARGUMENT IS > 174.673
          RESULT IS SET TO 1+0I
             (Issued by EXP(A+BI).  EXP(A+BI) is used in calculating
             Z**W, when W or Z is complex.)
 69 1556  rtn IN COMPLEX EXPONENTIAL FUNCTION IMAGINARY ARGUMENT IS >
          PI*2**50 = .3534E+16.  RESULT IS SET TO 1+0I
             (Issued by EXP(A+BI).  See explanation of message EX61.)
 6A 1555  rtn Z=0 AND N <= 0 IN Z**N.  RESULT IS SET TO 1+0I
             (Issued by X**Y.)
 6B 1505  rtn BOTH REAL AND IMAG ARGUMENTS ARE 0.  RESULT IS SET TO 1+0I
             (Issued by LOG(A+BI).  LOG(A+BI) is used in calculating
             Z**W, when W or Z is complex.)
 6C 1553  rtn ARGUMENT (number) IS > 174.673.  RESULT IS SET TO 1
             (Issued by EXP(X).  EXP(X) is called in calculating ERF,
             ERFC, TANH, SIN(A+BI), COS(A+BI), SINH(A+BI), COSH(A+BI),
             and EXP(A+BI).  EXP(A+BI) is in turn used in calculating
             Z**W, when W or Z is complex.)
 6D 1551  rtn X=0 AND Y ≤ 0 IN X**Y.  RESULT IS SET TO 1
 71       FILE CANNOT BE USED FOR UPDATE.
             (File is opened for INPUT or OUTPUT.)
 75 0943  FILE BEING CLOSED IS IN USE IN INTERRUPTED I/O STATEMENT.  IT
          IS NOT CLOSED.
 76 0944  INVALID ATTRIBUTES FOR SYSTEM FILE
 77 0023  FILE name - ATTEMPT TO READ/WRITE RECORD OF ZERO LENGTH
    0024
 78 0021  FILE name - LENGTH OF VARIABLE(number) ¬= LENGTH OF
    0022  RECORD(number)
             (The number of bytes of storage occupied by the variable
             must equal the number of bytes in the record.  See notes on
             the RECORD attribute, in "Differences between PL/I (F) and
             PL/C.")
 79       CONDITION (iden) SIGNALLED.  NO ON-UNIT PENDING
 7A       iden SIGNALLED.  "ERROR" RAISED AS STANDARD SYSTEM ACTION
 7B       KEY CONDITION RAISED.  ONCODE=number
 7C       NORMAL RETURN FROM "ERROR" ON-UNIT.  PROGRAM IS STOPPED
 7D       NORMAL RETURN FROM "FINISH" ON-UNIT.  PROGRAM IS STOPPED
 7E       ABOVE ERROR IS FATAL.  PROGRAM IS STOPPED
 7F       NOT ENOUGH CORE.  TRY LARGER REGION
```

```
            (A larger SIZE parameter should be specified on the $RUN
            command.  See "Running PL/C in MTS.")
 80       AUXILIARY I/O LIMIT EXCEEDED
            (See AUXIO option in "Running PL/C in MTS.")
 81       ATTEMPT TO SWITCH FILE TO SYSPRINT HAS FAILED
```

DIFFERENCES BETWEEN PL/I (F) AND PL/C

General Differences

PL/I (F) Features Not Included in PL/C:

(1) Regional auxiliary files.
(2) Controlled and based storage, and list processing.
(3) Multitasking.
(4) Compile-time facilities, except for INCLUDE and an incompatible MACRO facility.
(5) 48-character set option.
(6) Message DISPLAY to the operator.
(7) DEFINED and LIKE attributes.
(8) A few built-in functions and pseudo-variables.

Additional Restrictions Imposed by PL/C:

(1) 33 statement keywords and 6 auxiliary keywords are reserved and cannot be used as identifiers.
(2) The names of built-in functions and pseudo-variables are not reserved and may be used as identifiers, but if they are to be used in this way they should be explicitly declared; contextual declaration of these particular identifiers may succeed (depending upon context) but will produce a warning message.
(3) Parameters cannot be passed to the MAIN PROCEDURE of a PL/C program from the MTS $RUN command.
(4) String constants and comments must be contained in a single source card unless the PL/C NOBOUNDARY option is specified.
(5) String constants cannot have repetition factors.
(6) There are restrictions on the END, ENTRY, FORMAT, PROCEDURE, READ, and WRITE statements.
(7) There are restrictions on dimension, ENTRY, ENVIRONMENT, INITIAL, LABEL, and length attributes.
(8) Not all of the PL/I (F) condition codes are used by PL/C and the default condition states under PL/C are not exactly the same as those under PL/I (F).

Incompatible Features Added to PL/C:

(1) CHECK, NOCHECK, FLOW, and NOFLOW statements; a FLOW condition ONORIG, ONDEST, STMTNO built-in functions.
(2) Diagnostic options on the PUT statement.
(3) A built-in function to generate pseudo-random numbers.
(4) Comments that are convertible to source text depending upon the first letter of their contents.
(5) A text-replacement MACRO processor.

Differences in Internal Representation of Data:

Internally, PL/C carries out all floating-point arithmetic operations in double-precision form, adopting user-specified precision only on output. This means that computation is often somewhat more precise than would be the case under PL/I (F). The result is usually a slight difference in the least-significant figures of results, but of course it is possible for the differences to become highly significant.

PL/C assigns a fullword of storage to each FIXED BINARY variable and a doubleword of storage to each FIXED DECIMAL variable, regardless of the declared precision. This means that PL/C variables may hold values larger than their PL/I (F) counterparts. However, the default state for the SIZE condition in PL/C is "enabled" so that situations in which PL/C would give different results from PL/I (F) are detected.

Each bit in a PL/C bit-string is actually assigned an entire byte in storage. Each PL/C string variable also has an eight-byte control block called a dope vector so that an array of short strings takes a surprising amount of memory.

Decimal-base variables in PL/C are maintained internally in floating-binary form and converted on output.

This internal representation does not apply to record files, which are written in standard PL/I representation, and assumed to be in that representation when read. This means that PL/C and PL/I are compatible with respect to record files; files written by either compiler can be read by the other.

Order of Evaluation in DECLARE Statements:

PL/I (F) will reorder the evaluation of bounds and lengths and the initialization of variables so that, in the absence of circular dependencies, variables will be allocated and initialized before they are used to allocate or initialize other variables. PL/C uses a simpler strategy which depends upon the order in which DECLARE statements appear in the block, and the order in which variables are listed in a DECLARE statement:

(1) First, all scalar arithmetic and label variables are given their initial value.

(2) Then, proceeding in the order in which they are declared, strings, arrays, and structures are allocated space and initialized. Any expressions in the bounds or length fields are evaluated before space is allocated. After space has been allocated, the variable is initialized before processing the next variable in the order of declaration.

This strategy does not eliminate any allocation scheme available in PL/I (F) but does require the programmer to order his declaration of variables to avoid the use of unallocated or uninitialized variables declared in the same block.

Dimensional Limits in the Compiler:

The internal structure of the PL/C compiler is very different from that of the PL/I (F) compiler and it was not feasible to limit certain critical dimensions of the source program in exactly the same way. This means that there are probably some unusually large and complex programs that would be accepted by PL/C but would exceed some dimensional limit in PL/I (F); the opposite is certainly true. The compilation limits in PL/C are the following:

(1) Maximum nesting of IF statements is 12.
(2) Maximum static (syntactic) nesting of PROCEDURE, BEGIN, and DO statements is 11.
(3) Maximum nesting of factors in DECLARE is 6.
(4) Maximum number of label prefixes on a single statement is 87.
(5) Maximum depth of parenthesis nesting in expressions is 14.
(6) Maximum number of identifiers in a factor or structure in DECLARE is 88.
(7) No single expression can contain more than 256 symbols.

These limits are fixed by the structure of the compiler and cannot be changed by increasing the memory made available to the compiler. In most other respects the compiler's limits are related to the amount of memory available; for example, length of program and size of arrays. In these cases when the compiler indicates that a limit has been exceeded, the user can resubmit the program with a larger virtual memory region. An indication of which errors involve fixed limits and which can be alleviated with additional memory can be found in the section "Error Messages."

Comments

PL/C is a subset of PL/I. It is intended to be "upward compatible" with PL/I. A program that runs without error under the PL/C compiler should run under PL/I (F) and produce the same results.

Due to the many diagnostic features peculiar only to PL/C, if the programmer wishes to use these features for diagnostic runs under PL/C and still be able to run the same program under PL/I, he must enclose such features in the "pseudo-comments" described below.

PL/C permits sections of source to be treated either as source text or as comments, depending upon an option specified on the /COMPILE or /PROCESS card. The appropriate text is written as a normal PL/I comment, except that a colon or one of the integers from 1 to 7 is given as the first character of the comment. For example:

```
     /*5 X(I) = P(I+K); */

     /*: PUT DATA(X(I)); */
```

With normal default options, PL/C will treat these as normal comments (as will the PL/I (F) compiler). However, if the option COMMENTS is given on the /COMPILE or /PROCESS card then PL/C will scan as source text the content of all comments whose first character is a colon; hence, the PUT DATA statement will be included in the source program. If the option COMMENTS=(5) is given on the /COMPILE card, PL/C will scan as source text the content of all comments whose first character is either a 5 or a colon; hence, both of these statements would be included in the source program. Since the integers 1 to 7 may be used, one can establish seven different classes of "compilable comments" in a program and selectively include and exclude their contents from compilation just by changing the option specification on the /COMPILE, /OPTIONS, or /PROCESS card. If NOCOMMENTS is specified on a /OPTIONS or /PROCESS card, it negates the effect of a previous COMMENTS option.

Compilable comments were originally designed to permit the introduction of non-PL/I constructions in PL/C and still preserve compatibility with the PL/I (F) compiler. By simply enclosing all such constructions in what appear to PL/I (F) to be normal comments, one can run the program under either compiler without the necessity of physically removing cards. It would appear that the capability can have much wider use than this. Even a steadfast PL/I programmer who disdains the use of the special PL/C diagnostic statements might find occasions when it would be convenient to selectively include or exclude sections of source.

Note: In studying the source listing of a PL/C program, it is not always easy to keep track of what has been compiled and what is in fact a comment. The /COMPILE, /OPTIONS, or /PROCESS specification of course determines this, but a good local indication is the statement numbering; comments do not receive statement numbers.

Statements

Standard PL/I Statements:

The PL/I (F) statements that are not included in PL/C are:

ALLOCATE, DELAY, DISPLAY, FREE, LOCATE, UNLOCK, WAIT

The statements that are included in PL/C are listed below. Except as noted below these statements are exactly like their PL/I (F) counterparts as described in Section J of the IBM publication, IBM/360 Operating System PL/I (F) Language Reference Manual, form number GC28-8201.

BEGIN

- The ORDER and REORDER options are accepted by PL/C but are not effective; they do not alter the object code that is generated.

CALL

- The TASK, EVENT, and PRIORITY options are not included.
- Scalars may not be used as arguments for array or structure parameters.

CLOSE

- All files are rewound when CLOSEd by PL/C, either explicitly or implicitly. On succeeding file OPENs, the first record read from a file will be the first line of the file.
- When a tape file is CLOSEd, if data has been written onto the tape, a tape mark is written. In every case, the tape is then repositioned to the beginning of the tape file.

DECLARE, DCL

- The following words are reserved and cannot be used as identifiers:

ALLOCATE	BEGIN	CALL	CLOSE
DECLARE	DCL	DELETE	DO
END	ENTRY	EXIT	FORMAT
FREE	GET	GO	GOTO
IF	ON	OPEN	PROCEDURE
PROC	PUT	READ	RETURN
REVERT	REWRITE	SIGNAL	STOP
WRITE			
TO	BY	WHILE	THEN
ELSE			
NO	CHECK	NOCHECK	FLOW
NOFLOW			

All of the other keywords are available for use as identifiers, but note the next requirement below.

- The names of built-in functions and pseudo-variables are not reserved and may be used as identifiers, but if they are to be used in this way they should be explicitly declared; contextual declaration of these particular identifiers may succeed (depending upon context) _but will produce a warning message_. For example, HIGH can be used as a variable name, but it should be listed in a DECLARE statement. Explicit declaration as a label or entry name is also accepted.

- PL/C does not support all of the attributes of PL/I (F).

- Attributes may be factored exactly as in PL/I (F). The maximum depth of factoring is 6.

- The maximum number of identifiers that can be included in one factor or one structure is 88.

DELETE

- The FILE and KEY options are included. KEY must be used for DIRECT files.

DO

END

- If a label (entry name) follows END, it must have been the first (leftmost) label (entry name) on a preceding BEGIN, DO, or PROCEDURE statement to be effective.

ENTRY

- Scalars may not be used as arguments for array or structure parameters.

- The entry name on an ENTRY statement in PL/C cannot be identical to an identifier that has been declared earlier in the procedure that contains the ENTRY statement. PL/C will reject such an entry name as a "multiple declaration" even though this is a valid PL/I construction.

EXIT

- Since PL/C does not include multitasking, the EXIT statement is exactly equivalent to the STOP statement.

FORMAT

GET

- The files SYSIN and SCARDS are synonymous in PL/C. Their input record lengths default to 100 characters, but may be redefined.

- See notes on the OPEN statement below.

GO TO, GOTO

IF

- The maximum nesting depth for IF statements is 12.

Null

ON

- The default states for all PL/C conditions except CHECK and FLOW are enabled. This differs from PL/I (F) where the default states for SIZE, STRINGRANGE, and SUBSCRIPTRANGE are "disa- bled." If SUBSCRIPTRANGE is disabled (by a NOSUBSCRIPTRANGE prefix), the integrity of the compiler cannot be guaranteed and batch operation is threatened.

- The AREA, KEY, and PENDING conditions are not included in PL/C.

- Not all of the PL/I (F) ON codes are included in PL/C.

- An incompatible FLOW option has been added to PL/C.

- As in PL/I (F), an ON-unit cannot consist of a DO, END, RETURN, FORMAT, PROCEDURE, or DECLARE statement.

- A FLOW condition that is not part of PL/I has been added to PL/C.

OPEN

- The following options are included:

TITLE, PRINT, LINESIZE, PAGESIZE, INPUT, OUTPUT, RECORD, SEQUENTIAL, STREAM, KEYED, DIRECT, and UPDATE

- The following options are not included:

  BUFFERED, UNBUFFERED, BACKWARDS, EXCLUSIVE, and TRANSIENT

- All PL/C files have the EXTERNAL default attribute.

- The TITLE option, in PL/C, is used to specify the name of an MTS file or device, or a logical I/O unit name or number, with which all I/O operations associated with the PL/C file variable will be performed. If the option is not used and no ATTACH control card naming the file variable has been encountered, the PL/C file variable name is assumed to be the name of an MTS file or device with which I/O will be done, i.e., if a file variable MONIES was specified in an OPEN statement, the MTS file MONIES would be referred to for I/O, or if SERCOM was the file variable, all its I/O would be done on the logical I/O unit SERCOM. If the TITLE option were not available, the only MTS files which could be referred to in this way would be files or devices whose names obeyed PL/I identifier naming conventions, which are somewhat restrictive. Hence, if one wants to refer to an MTS file named ...OOPS, the option TITLE('...OOPS') should be included on the OPEN statement for the file in the PL/C program. Note that the logical I/O units 0-19 may be referred to in this way, e.g., TITLE('0').

- The PL/C compiler input (SCARDS/SYSIN) stream defaults to 100 characters in length. The length of the output (SPRINT/ SYSPRINT) stream defaults to either 133 characters, or the maximum file/device length, whichever is shorter. Both may be redefined by explicit OPENing.

- For other PL/C files, record-length defaulting is presented in the figure below. Length-definition processing proceeds from left to right, top to bottom. As soon as a value is defined by one level of choice, that is the ultimate definition. Vertical lines represent mutually exclusive paths of choice. If any file has the PRINT option/attribute, one character carriage control is added.

```
                                         ┌─────@F spec. length
                               ATTACHed │
                          ┌─────────────┤
                          │    file     │
  LINESIZE────ENVIRONMENT─┤             └─────80
   value      information │
                          │
                          │             ┌─────system input:100
                          │             │     [INCLUDEd files]
                          └─────────────┤
                               other    ├─────PRINT:120
                                        │
                                        └─────MTS max. device
                                                     length
```

Figure 1: Record-Length Defaulting

See the ENVIRONMENT option notes below.

PROCEDURE, PROC

- The TASK option is not included.
- The RECURSIVE option is included. All procedures under PL/C are recursive, but unless the RECURSIVE option is specified, recursive use of the procedure will result in a run-time diagnostic message.
- The ORDER and REORDER options are accepted, but are ineffective. They have no effect on the object code generated.
- Scalars may not be used as arguments for array or structure parameters.
- Like PL/I, but <u>unlike previous releases of PL/C</u>, an asterisk (*) may <u>not</u> be given as a length specification in the RETURNS option. CHAR(256) VAR or BIT(256) VAR may be used to declare a procedure which returns a string of arbitrary length.

PUT

- All of the PL/I (F) options are included.
- Additional diagnostic options that are not part of PL/I (F) are included in PL/C. See "Incompatible PL/C Diagnostic Statements" below.
- The files SPRINT and SYSPRINT are synonymous in PL/C. Their output record lengths are the minimum of 133 characters and the MTS maximum output record length, but may be redefined.

- See the notes on the OPEN statement, above.

READ

- Only the FILE, INTO, KEY, KEYTO, and IGNORE options of PL/I (F) are included.

- PL/C-written fixed-length records are fully compatible with PL/I (F). But note that LABEL variables written by PL/C-compiled programs can be meaningful only when read back in by the same PL/C program in the same compiler run.

- See also the notes on the RECORD attribute and the OPEN statement.

RETURN

REVERT

REWRITE

- FILE, FROM, and KEY supported. File must be opened for update. If KEY is specified, file must be DIRECT.

SIGNAL

STOP

WRITE

- Only the FILE, KEYFROM, and FROM options of PL/I (F) are included.

- FILE and FROM must be present.

- PL/C-written fixed-length records are fully compatible with PL/I (F). But note that LABEL variables written by PL/C-compiled programs can be meaningful only when read back in by the same PL/C program in the same compiler run.

- See also the notes on the RECORD attribute and the OPEN statement.

Incompatible PL/C Diagnostic Statements:

The following PL/C statements are <u>not</u> PL/I (F) statements. When these statements are used in a program, compatibility with PL/I (F) is lost. However, compatibility may be preserved by enclosing these statements in a "compilable comment."

NOCHECK

- When NOCHECK is encountered in the execution of a block that is within the scope of a CHECK prefix, the printing that would normally result from the raising of the CHECK condition is suppressed. If a CHECK prefix is not present, the NOCHECK statement is ineffective.

CHECK

- When CHECK is encountered in the execution of a block, that is within the scope of a CHECK prefix, the printing that results from the raising of the CHECK condition (which may have been suppressed by a previous NOCHECK statement) is resumed. If no CHECK prefix is present, or there has been no preceding NOCHECK statement, the CHECK statement is ineffective. Note that the normal action is to do the printing that results from the raising of the CHECK condition, so that the NOCHECK statement is provided to override this normal action. This is the opposite of the situation for the FLOW condition.

- Alternate forms have one or two control parameters:

```
            CHECK(exp1) or CHECK(exp1,exp2)
```

  "exp1" specifies the maximum number of items in the printing resulting from raising of the CHECK condition _in the current block_ that will appear. After the specified number of instances, NOCHECK is automatically applied. "exp2" gives the maximum number of times the printing of the CHECK condition will be permitted in _each block dynamically entered from the current block_. That is,

  CHECK(N,M) is equivalent to

```
            CHECK(N) in the current block, and
            CHECK(M,M) as the first statement in every
                  block entered from the current block.
```

  CHECK(N) is equivalent to

```
            CHECK(N) in the current block, and
            CHECK is the first statement in every
                  block entered from the current block.
```

  CHECK is equivalent to

```
            CHECK in the current block, and
            CHECK as the first statement in every block
                  entered from the current block.
```

  Each time that a CHECK statement is encountered, the controlling counters are reset to the new limiting values.

NOFLOW

- When NOFLOW is encountered in the scope of a FLOW prefix, the printing that results from raising the FLOW condition is suppressed. If the FLOW condition is disabled, the NOFLOW statement is ineffective.

FLOW

- When FLOW is encountered in the scope of a FLOW prefix, the printing resulting from the raising of the FLOW condition is resumed. If the FLOW condition is disabled, or there has been no preceding NOFLOW statement, the FLOW statement is ineffective. Note that the normal action is <u>not to do the printing</u> that results from the raising of the FLOW condition, so that the FLOW statement is provided to override this normal action. This is the opposite of the situation for the CHECK condition.

- Alternate forms have one or two control parameters:

  ```
  FLOW(exp1) or FLOW(exp1,exp2)
  ```

  These have exactly the same interpretation as for CHECK as described above.

- The FLOW condition is raised by any action that potentially alters the normal sequential flow-of-control. That is, by the CALL, DO, GOTO, RETURN, and IF statements, by any exceptional condition (except FLOW) which would cause an ON-unit to be entered, and by in-line procedure references.

Diagnostic Options on the PUT Statement:

OFF	- suppresses printing of execution output on SPRINT/SYSPRINT.
ON	- resumes printing of execution output on SPRINT/SYSPRINT.
FLOW	- displays the recent FLOW history of the program.
SNAP	- displays the recent calling history of the program.
ALL	- displays the current values of all automatic, scalar variables in the blocks active at the time of encounter, and the current values of all static or external scalar variables.
ARRAY	- same as ALL but includes array as well as scalar variables.

DEPTH(exp) - specifies the depth of block nesting for which the display is to be produced. Used only with SNAP, ALL, or ARRAY options.

If an ON or OFF option appears, it must be the only option on the statement. FLOW, SNAP, ALL, and ARRAY can appear in any combination with each other, and with the standard SKIP, PAGE, or LINE options of PL/I. They cannot be combined with FILE, STRING, LIST, DATA, or EDIT options.

Attributes

Standard PL/I (F) Attributes:

The PL/I (F) attributes not included in PL/C are listed below. These words are recognized by PL/C as attributes, but deleted with an appropriate message:

AREA
BACKWARDS
BASED
BUFFERED, BUF
CONTROLLED, CTL
EVENT
EXCLUSIVE, EXCL
GENERIC
IRREDUCIBLE
LIKE
OFFSET
PACKED
POINTER, PTR
POSITION, POS
REDUCIBLE
TASK
UNBUFFERED, UNBUF

The attributes that are included in PL/C are listed below. Except as noted below these attributes are exactly like their PL/I (F) counterparts as described in section I of the IBM publication, IBM System/360 Operating System PL/I (F) Language Reference Manual, form number GC28-8201.

ALIGNED

AUTOMATIC, AUTO

BINARY, BIN

BIT

BUILTIN

CHARACTER, CHAR

COMPLEX, CPLX

DECIMAL, DEC

DIMENSION

- *'s _must_ be used for the bounds in all DIMENSION attributes associated with a parameter. This includes the bounds in the DIMENSION attributes used in the ENTRY attribute for an ENTRY parameter. *'s may not be used for the bounds under any other circumstances.

DIRECT

- See "Indexed I/O in PL/C" for an explanation of the differences.

ENTRY

- Structure parameters cannot be declared in the list after ENTRY. This means an entry name cannot be passed as an argument if any of its parameters is a structure.
- In the absence of explicit attributes, PL/C supplies a scalar parameter with FLOAT, DECIMAL, and REAL attributes, whereas PL/I (F) makes no assumption as to attributes.
- The bounds in all dimension attributes and the length in all BIT or CHAR attributes in the parameter list that follows ENTRY must be *'s.

ENVIRONMENT, ENV

- The CONSECUTIVE, INDEXED, F, V, U, CTLASA, CTL360, and GENKEY options are recognized; CTL360 and GENKEY are ignored.

  When the associated PL/C file is attached to a magnetic tape, the F, V, or U specification in the ENV attribute is taken to define the blocking parameters for the tape. The record-length/blocking parameter processing procedure for magnetic tapes is as follows:

  In the following, BLKSI refers to the block size parameter specified in the ENV options list, and LRECL refers to the logical record length. Their ultimate definitions are used by the MTS tape routines.

If an F, U, or V format specification has not been given, or no ENV option has been specified, the current DSR format and blocking parameters are used. If a format has been specified, the following actions take place:

(a) If a LINESIZE value has been defined, it overrides any LRECL specification, and sets LRECL=LINESIZE (+1, if PRINT).
(b) If a LRECL has not been defined, a BLKSI <u>must</u> have been defined, or else an error occurs. LRECL is set to BLKSI (-8, if V format), if BLKSI is defined.
(c) If a LRECL has been defined (either by the LINESIZE override or explicitly), BLKSI is set to LRECL (+8, if V format), if not explicitly given.

When attached to an MTS file or device, only the F specification is meaningful. It is taken as defining a record length to which all input records will be padded and all output records to be truncated. The length is defined precisely as with magnetic tapes, described above. The resultant LRECL value is taken as the file's record length.

EXTERNAL, EXT

FILE

- Only EXTERNAL files are permitted.

- Variables declared with the FILE attribute are presumed to be the name of an MTS file, device, or logical I/O unit with which all I/O associated with the FILE variable is to be performed. For example, all I/O performed with a FILE variable named SPUNCH would be done on the MTS logical I/O unit SPUNCH. This would also be the case for a FILE variable PAYROLL and an MTS file PAYROLL. To perform I/O operations upon MTS files or devices which cannot be referred to in this way (because they do not follow PL/I variable naming conventions), see the TITLE option description notes.

FIXED

FLOAT

INITIAL, INIT

- Iteration factors, but not string repetition factors, are allowed in the INITIAL list. This means that the phrase (x) (1) '---' in PL/I (F) would have to be given as (x) ('---') in PL/C. But the phrase (x) '---' is not allowed in PL/C.

INPUT

INTERNAL, INT

KEYED

- PL/C KEYED I/O is not compatible with PL/I (F) KEYED I/O, but is almost identical to PL/I KEYED I/O on MTS. The KEYs used in PL/C correspond to the MTS line number for the I/O operation. Thus, the KEYED attribute should only be applied to PL/C files that are attached to MTS line files.

- The KEYED attribute must be specified for a PL/C file whenever KEYED I/O is desired. Unlike PL/I on MTS, the CONSECUTIVE attribute is not recognized in PL/C for KEYED files. Also unlike PL/I on MTS, the GENKEY attribute is ignored in PL/C.

- The key used in KEY, KEYTO, or KEYFROM clauses in I/O statements is the internal form of the MTS line number of the line read or written (e.g., the line number times 1000). The KEY must be convertible to FIXED BINARY(31). PL/C will automatically perform conversion to the appropriate form whenever possible. The automatic string/numeric conversion feature of PL/C allows the user to perform operations similar to KEYED I/O with the GENKEY option in PL/I on MTS. For example, the following program will read line number 123.000 from the file MYFILE.

```
MAIN:  PROCEDURE OPTIONS(MAIN);
       DECLARE BUFF CHARACTER(80) VARYING,
               LINENUM CHARACTER(10) VARYING,
               MYFILE FILE KEYED ENVIRONMENT(INDEXED);
       LINENUM = '123000';
       READ FILE (MYFILE) INTO (BUFF) KEY(LINENUM);
       END MAIN;
```

- Sequential I/O may be performed on KEYED files in PL/C. The line number read or written by the sequential I/O operation may be obtained through the KEYTO clause.

- The condition ONKEY (oncode = 69) may be used to detect end-of-file conditions resulting from KEYED I/O.

LABEL

- The optional list of statement-label constants of PL/I (F) is not permitted in PL/C and will be discarded if given.

LENGTH

- An (*) must be used for the length in all CHAR and BIT attributes associated with a parameter. This includes lengths in the ENTRY attribute. *'s may not be used for lengths under any other circumstances.

- Length cannot be greater than 256.

OUTPUT

PICTURE, PIC

- Sterling PICTURE data-types are not supported.
- PL/C will not provide for drifting signs in exponents.

PRECISION

PRINT

REAL

RECORD

- Only a limited form of RECORD I/O is implemented (see READ and WRITE above).
- PL/C expects all input records read from a RECORD file to be a fixed length, the length being the size, in bytes, of the internal representation of the variable appearing in the READ statement's INTO option. It should be noted that since MTS may trim excess blanks off of ends of input records, PL/C may complain about some records in a RECORD FILE. To prevent this occurrence, the use of the ATTACH control card and the ENVIRONMENT option should be considered.

RETURNS

SEQUENTIAL, SEQL

STATIC

STREAM

UNALIGNED, UNAL

UPDATE

VARYING, VAR

Factoring

- The maximum depth of factoring is 6.
- The maximum number of identifiers in a factor is 88.

Structures

- The maximum number of identifiers in a structure is 88.

Built-In Functions and Pseudo-Variables

The PL/I (F) built-in functions listed below are not included in PL/C. A plus sign (+) after the name indicates that a pseudo-variable of the same name is not included.

Based Storage:

ADDR, EMPTY, NULL, NULLO

Multitasking:

COMPLETION+, PRIORITY+, STATUS+

Array Generic:

POLY

String Generic:

STRING pseudo-variable

Miscellaneous:

ALLOCATION

Notes:

(1) The following built-in functions and pseudo-variables are included in PL/C:

Pseudo-Variables:

SUBSTR, REAL, IMAG, COMPLEX, ONCHAR, ONSOURCE, UNSPEC

Incompatibility with the PL/I (F): The right-hand side of the assignment

```
UNSPEC(decimal-fixed-variable) = bit-string;
```

must be a bit string that represents valid IBM 360 packed-decimal decimal data.

Arithmetic Generic:

ABS, ADD, DIVIDE, MAX, MIN, MOD, MULTIPLY, SIGN, FLOOR, CEIL, TRUNC, COMPLEX, CONJG, IMAG, REAL, ROUND

Incompatibility: FLOOR and CEIL (FLOAT scale) under PL/I (F) treat all numbers of magnitude less than 16**(-16) as zero; under PL/C only true zeros are treated as zero.

Mathematical Generic:

EXP, LOG, LOG10, LOG2, ATAND, ATAN, TAND, TAN, SIND, SIN, COSD, COS, TANH, ERF, ERFC, SQRT, COSH, SINH, ATANH

String Generic:

BOOL, HIGH, INDEX, LENGTH, LOW, REPEAT, STRING, SUBSTR, TRANSLATE, UNSPEC, VERIFY

Array Generic:

ALL, ANY, DIM, HBOUND, LBOUND, PROD, SUM

Incompatibility: In PL/C the built-in functions DIM, LBOUND, and HBOUND operate without actually evaluating an array expression, if such an expression is given as the first argument. Since PL/I (F) evaluates the expression before performing the function, there could be "side effects" under PL/I (F) that will not occur in PL/C.

Type Conversion:

BINARY, BIT, CHAR, DECIMAL, FIXED, FLOAT, PRECISION

Condition:

DATAFIELD, ONCHAR, ONCODE, ONCOUNT, ONFILE, ONKEY, ONLOC, ONSOURCE

PL/C has an additional condition: KEY condition with ONCODE= 69. This will be raised if:

(a) A REWRITE or DELETE is issued and the previous I/O operation was not a READ.
(b) A DELETE is invalid because the file is not opened for UPDATE.

PL/C does not include all of the ON codes of PL/I (F).

Miscellaneous:

COUNT, DATE, LINENO, TIME

PL/C Built-in Functions Not Included in PL/I (F):

RAND, ONORIG, ONDEST, STMTNO

See notes (5), (6), and (7) below.

(2) Except as noted otherwise, all built-in functions perform exactly as do their PL/I (F) counterparts. In other words, the values of SIN(x), for example, in PL/C and in PL/I (F) are identical floating-point numbers when the functions are applied to the same floating-point value of "x". The methods used in PL/C are those described in the IBM publication, IBM System/360 Operating System, PL/I Subroutine Library, Computational Subroutines, form number GC28-6590.

(3) The names of built-in functions and pseudo-variables are not reserved and may be used as identifiers, but if they are to be used in this way they should be explicitly declared; contextual declaration of these particular identifiers may succeed (depending upon context) but will produce a warning message. For example, HIGH can be used as a variable name, but it should be listed in a DECLARE statement. Explicit declaration as a label or entry name is also accepted.

(4) Note on UNSPEC: Although the PL/I (F) and PL/C internal representation of certain data-types differ, the PL/C and PL/I (F) UNSPEC functions produce identical results (except for the packed-decimal error case noted under note (1) above). The PL/C UNSPEC is somewhat more complicated (and time-consuming to execute) than its PL/I (F) counterpart, since the PL/C UNSPEC must convert to and from the PL/I (F) representation.

(5) A built-in function to produce a sequence of pseudo-random numbers has been included in PL/C. There is no comparable built-in function in PL/I (F). When this function is used, a PL/C program will not run under PL/I (F). Therefore, a program that includes the use of this built-in function will receive a warning message from PL/C.

Definition:

RAND finds the next number in a pseudo-random number sequence in which the argument was the last element, and returns the next number to the point of invocation.

Reference:

RAND(x)

Argument:

The argument "x" may be an element or array expression of coded-arithmetic type. It must be REAL. If it is not also

FLOAT, it will be converted to FLOAT by the RAND function. The value must be in the range 0 < x < 1.

Result:

The value returned will be FLOAT with the same base (BINARY, DECIMAL) as the argument.

Use:

RAND is normally used in an assignment statement:

```
x = RAND(x);
```

Repeated execution of such a statement will cause "x" to take on the successive values in a sequence of pseudo-random numbers uniformly distributed over the interval (0.0, 1.0), so long as the initial, or "seed", value of "x" was properly chosen. The "seed" value should be between 0 and 1, and have nine significant figures, the rightmost of which is odd. This will maximize the period of the sequence. The generation method used in the PL/C RAND function is based on the method of Coveyou and Macpherson described in _Journal of the ACM_, 14:1(1967), pp. 100-119.

(6) The ONORIG and ONDEST built-in functions have no arguments and are nonzero only within an ON FLOW unit. ONORIG returns the statement number of the statement that caused the FLOW condition to be raised, and ONDEST returns the statement number of the statement that is the target of that transfer.

(7) The STMTNO built-in function has a single argument giving the label of a source statement, and it returns the statement number of that statement. It is intended primarily for use in an ON FLOW unit, but may appear in any context.

Conditions

The PL/I (F) conditions _not_ included in PL/C are:

AREA, PENDING

All of the other PL/I (F) conditions are included, and the FLOW condition has been added. The PL/C conditions are equivalent to their PL/I (F) counterparts, which are described in section H of the IBM publication, _IBM PL/I (F) Language Reference Manual_, form number GC28-8201, except as noted below. The conditions and acceptable abbreviations are listed below, with the default state underlined:

CHECK <u>NOCHECK</u>

The default PL/C output for the CHECK condition is not identical to that of PL/I (F). When an element of a "checked" array is changed, PL/C displays only that particular element (rather than the entire array). Similarly, when a member of a "checked" structure is changed, only that member is displayed (rather than the entire structure). The timing of the PL/C display is also not exactly the same as that of PL/I (F). The PL/C display occurs immediately after the CHECK condition is raised, rather than waiting until the end of the statement. See also the description of the CHECK statement.

<u>CONVERSION</u> <u>CONV</u> NOCONVERSION NOCONV

<u>ENDFILE</u>

<u>ENDPAGE</u>

<u>ERROR</u>

The PL/C ERROR condition is not entirely compatible with PL/I (F). The standard system action in PL/I (F) is to raise the FINISH condition and stop. In PL/C, the standard system action is to apply the automatic PL/C error correction and then continue execution; the FINISH condition is not raised (unless that particular PL/C "correction" terminates execution). Normal return (if there is a pending ON unit) is compatible--both PL/C and PL/I (F) signal FINISH and terminate execution.

<u>FINISH</u>

<u>FIXEDOVERFLOW</u> <u>FOFL</u> NOFIXEDOVERFLOW NOFOFL

FLOW <u>NOFLOW</u>

The FLOW condition has been added to PL/C to permit the dynamic monitoring of the flow-of-control. When the FLOW condition is enabled by a FLOW prefix, the condition is raised by any statement that potentially will alter the normal sequential flow of execution. These are the CALL, DO, GOTO, IF, and RETURN statements, any exceptional condition (except FLOW) which causes an ON-unit to be entered, and references to user-defined functions.

The standard system action depends on the FLOW and NOFLOW statements. If the condition is raised within the scope of a NOFLOW statement, the origin and destination of the transfer are saved in a first-in-first-out queue which will hold up to 18 entries. A PUT FLOW statement will cause the contents of this queue to be printed in the form:

```
oooo -> dddd   or   nnn*(ooo -> ddd)
```

where "oooo" is the origin statement number, "dddd" is the destination statement number, and "nnn" is a count of the number of times the transfer was made in succession. If the condition is raised within the scope of a FLOW statement, the origin and destination of the transfer are immediately printed on SPRINT in the same format as used by the PUT FLOW statement. This may, of course, be replaced with a user-supplied ON FLOW unit. The ONORIG and ONDEST built-in functions are useful in such an ON-unit. The NOFLOW and FLOW statements will still serve to dynamically suppress and resume the printing that may result from the raising of the FLOW condition. The user should disable the FLOW condition in any FLOW ON-unit supplied (by using a NOFLOW prefix) to avoid an infinite loop.

If the FLOW or NOFLOW prefix is applied to a PROCEDURE or BEGIN statement, its scope is the entire block. If applied to a DO statement, its scope is the DO group. Applied to other statements, its scope is the single statement.

KEY

See "Indexed I/O in PL/C" for a description of KEY condition 69.

<u>NAME</u>

<u>OVERFLOW</u> <u>OFL</u> NOOVERFLOW NOOFL

<u>RECORD</u>

<u>SIZE</u> NOSIZE (Note default is different from PL/I (F).)

When the SIZE condition is raised, the arithmetic results in PL/C will differ from those produced by PL/I (F). In particular, PL/I (F) may truncate results on the left to the user-specified precision, while PL/C always retains the implementation-defined maximum precision.

<u>STRINGRANGE</u> <u>STRG</u> NOSTRINGRANGE NOSTRG
(Note default is different from PL/I (F).)

<u>SUBSCRIPTRANGE</u> <u>SUBRG</u> NOSUBSCRIPTRANGE NOSUBRG
(Note default is different from PL/I (F).)

The NOSUBSCRIPTRANGE prefix will not be effective unless the NOMONITOR=(SUBRG) suboption is specified. Since out-of-range subscripts can damage the compiler (if PL/C is not in the link pack area) and interfere with batch operation, many installations elect to inhibit (override) this option.

<u>TRANSMIT</u>

<u>UNDEFINEDFILE</u> <u>UNDF</u>

<u>UNDERFLOW</u> <u>UFL</u> NOUNDERFLOW NOUFL

User-Defined Condition

<u>ZERODIVIDE</u> <u>ZDIV</u> NOZERODIVIDE NOZDIV

Notes:

(1) All of the PL/I (F) condition prefixes are included in PL/C.

(2) All of the conditions listed above can be given in ON, REVERT, and SIGNAL statements. These include conditions beginning with "NO", e.g., NOCHECK statements.

(3) As noted in the ERROR condition, PL/C will attempt to correct run-time errors and continue. In the case of arithmetic condition, if the condition is disabled (i.e., a NO-prefix is in effect), the correction will be suppressed. If a user-supplied ON-unit is pending, the correction will be applied, but no error message will be given.

<u>Prefixes</u>

(1) Prefixes can be given in PL/C in any order. For the sake of compatibility, a warning message will be issued if they are not in the order required by PL/I (F):

```
THEN or ELSE

   condition prefixes

      labels or entry names
```

They will, however, perform correctly, regardless of the order of appearance.

(2) Multiple labels and entry names may be given (with a maximum of 87 on one statement), but only the first (leftmost) label or entry name on a statement can be referenced by a subsequent END statement.

(3) All of the PL/I (F) condition prefixes are included in PL/C, and FLOW and NOFLOW have been added. These are listed below with the default state underlined:

CHECK <u>NOCHECK</u>

As in PL/I (F) the CHECK and NOCHECK prefixes may be given only on PROCEDURE and BEGIN statements. Unlike PL/I (F), the raising of the CHECK condition may be suppressed and restored <u>dynamically</u> within the block by means of the PL/C CHECK and NOCHECK statements.

<u>CONVERSION</u> <u>CONV</u> NOCONVERSION NOCONV

<u>FIXEDOVERFLOW</u> <u>FOFL</u> NOFIXEDOVERFLOW NOFOFL

<u>FLOW</u> NOFLOW (not a PL/I prefix)

If the FLOW or NOFLOW prefix is applied to a PROCEDURE or BEGIN statement, its scope is the entire block. If applied to a DO statement, its scope is the DO group. Applied to other statements, its scope is the single statement.

<u>OVERFLOW</u> <u>OFL</u> NOOVERFLOW NOOFL

<u>SIZE</u> NOSIZE

<u>STRINGRANGE</u> <u>STRG</u> NOSTRINGRANGE NOSTRG

<u>SUBSCRIPTRANGE</u> <u>SUBRG</u> NOSUBSCRIPTRANGE NOSUBRG

If SUBSCRIPTRANGE is disabled, the integrity of the compiler cannot be guaranteed and batch operation is threatened.

<u>UNDERFLOW</u> <u>UFL</u> NOUNDERFLOW NOUFL

<u>ZERODIVIDE</u> <u>ZDIV</u> NOZERODIVIDE NOZDIV

Note that the default state for all PL/C conditions except CHECK is "enabled." This differs from PL/I (F) where the default for SIZE, STRINGRANGE, and SUBSCRIPTRANGE is "disabled."

Multiple condition prefixes may be given on a statement.

If a condition prefix is otherwise correct, PL/C will accept a space after NO in the condition name. That is, NO CHECK will be accepted for NOCHECK. A warning message will be given.

(4) The entry name on an ENTRY statement in PL/C cannot be identical to an identifier that has been declared earlier in the PROCEDURE that contains the ENTRY statement. PL/C will reject such an entry name as a "multiple declaration" even though this is a valid PL/I construction.

The PL/C Macro Feature

Macro Definition:

A macro definition must be given within a "macro packet." A packet begins with a /MACRO card (/ in column 1) and ends with a /MEND card (/ in column 1). One or more macro definitions comprise a packet. There is no limit to the length of a packet, or to the number of packets that may be used, although the memory available will restrict the total amount of macro text in a program. Packets may be inserted anywhere in the source program, but each macro _must be defined before it is referenced_. A macro definition has the form:

```
     <macro name> = <macro body> %;
```

or

```
     <macro name>(formal-param-1,formal-param-2,...,
          formal-param-n) = <macro body> %;
```

The macro body may contain any text except the sequence "%;". No requirement is made regarding balanced quotes or partial comments. Care should be taken about card boundaries, since macro text is interpreted as a series of _lines_, each with its own indentation. Two separate lines will never be put on the same line during expansion, although it may be necessary to split a line to fit within source margins.

The macro name may consist of up to 255 characters, starting with an alphabetic character. Like all symbols in PL/C, macro names cannot be split over card boundaries. (This is independent of the BNDRY option.) The name must be distinct from all other identifiers used in the program. PL/C keywords may be used as macro names, but such names will not be recognized as macros in any text which is syntactically scanned before the definitions are processed.

The formal parameters are recognized only within the macro being defined. A maximum of 10 formal parameters is allowed in a macro. They may contain up to 31 characters, again starting with an alphabetic character. Formal parameters are PL/C symbols, and therefore cannot be split over card boundaries. Formal parameters are local to a macro definition, and will override within that definition any other use of the symbol (identifier, keyword, or macro name). They are recognized only when surrounded by any of the PL/C delimiters (including blanks and source-text boundaries). Parameters appearing within comments and strings are also recognized, although, again only when properly delimited. Thus, if E is a formal parameter in the text

```
     E = 12;    /* SET E TO UPPER BOUND */
```

the E's in SET and UPPER will <u>not</u> be considered parameter instances.

Macro Expansion:

Within PL/C source text, macro calls are expanded when encountered during syntactic analysis. This means that macro calls inside of strings and comments will <u>not</u> be recognized. Macro text may contain further macro references, either explicit or parameter substitutions, which are expanded when encountered during the scan of the original expansion. A macro call is of the form:

```
<macro name>
```

or

```
<macro name>(actual-param-1, actual-param-2,...,actual-param-n)
```

The second form may be split over a card boundary, but the macro name itself must not be split. The number of actual parameters must equal the number of formal parameters in the macro definition. During expansion, each actual parameter is substituted for the corresponding formal parameter in the macro body before the syntactic-analysis scan of the macro text is performed.

Actual parameters are treated as pure text during substitution; that is, embedded parameters are not recognized. Macro references are not detected until the macro text containing the parameter is syntactically scanned. Blanks surrounding actual parameters in the call are ignored. As a convenience, there are two additional rules governing the text of actual parameters:

(1) If a parameter begins with a single quote, all text up to and including the next single quote will be considered part of the parameter. (Two consecutive quotes within such text do not terminate the parameter and remain unchanged.)

(2) If an actual parameter begins with a left parenthesis, all text up to the matching right parenthesis is considered part of the parameter. The outer parentheses, however, are discarded from the text of the parameter before substitution.

For example, given the macro packet

```
/MACRO
 ASGN(P,Q,R) = SUBSTR(P,Q) = R; %;
/MEND
```

and the macro call

```
ASGN(X,(10,3),'ABC''XYZ')
```

the expansion would be

```
SUBSTR(X,10,3) = 'ABC''XYZ';
```

Macro Display:

Unless NOSOURCE is specified, macro definitions are always printed. Macro calls are printed by default; NOMCALL may be used to suppress this printing. Macro expansions are normally printed; the NOMTEXT option will cause them not to appear. If NOMTEXT and NOMCALL are <u>both</u> used, the default options, MTEXT and MCALL, are applied.

PL/C POST-MORTEM DUMP STATISTICS REPORT

The PL/C post-mortem dump statistics report is generated in response to the R suboption on the DUMP, DUMPE, or DUMPT options. Much of the report is self-explanatory. However, note the following:

(1) PAGES and LINES refer to output to the system output file, SPRINT. CARDS refers to input from the system input file, SCARDS. INCL'S refers to the total number of cards read from all /INCLUDEd files. AUXIO refers to the total number of records read from, written to, or updated in auxiliary files. This includes auxiliary input file cards that are /INCLUDEd.

(2) BYTES USED figures are given exactly and then rounded up to the nearest K (=1024 bytes). BYTES UNUSED figures are given exactly and then rounded down to the nearest K.

```
For example:   USED 1025 (2K)
             UNUSED 1025 (1K)
```

Since some internal PL/C tables grow and shrink, BYTES USED means the maximum amount used at any time, and BYTES UNUSED means the minimum amount unused at any time.

(3) SYMBOL TABLE refers to the PL/C internal table that records information about identifiers, variables, constants, entries, files, and blocks used in the program. It exists through compilation and execution. Symbol table space is also used during compilation for storing macro definitions. (See diagram below.)

(4) INTERMEDIATE CODE refers to the PL/C internal representation of the program during compilation. It exists only during compilation. (See diagram below.)

(5) OBJECT CODE refers to the machine code generated by PL/C for the program. It is created by compilation, and exists through execution. (See diagram below.)

(6) STATIC CORE refers to memory for storing STATIC (and EXTERNAL) variables. It exists during execution time. Even if no STATIC or EXTERNAL variables are declared, STATIC CORE USED will be approximately 350 bytes for fixed overhead. (See diagram below.)

(7) AUTOMATIC CORE refers to memory for storing AUTOMATIC variables. It exists during execution time. Even if no 2582 AUTOMATIC variables exist in the program, AUTOMATIC CORE USED will be approximately 200 bytes. (See diagram below.)

(8) DYNAMIC CORE refers to memory for auxiliary and /INCLUDE file DCBs, buffers, macro expansions, and several other miscellaneous 2586 functions. It is used dynamically through compilation and execution time. (See diagram below.)

(9) TOTAL STORAGE refers to the total through both compilation and execution. As noted above, TOTAL STORAGE USED is the maximum amount used at any time, and TOTAL STORAGE UNUSED is the minimum amount unused at any time. (See diagram below.)

(10) In the line "THIS PROGRAM MAY BE RERUN WITHOUT CHANGE IN A REGION rK BYTES SMALLER USING TABLESIZE=t", "r" is simply taken from TOTAL STORAGE UNUSED and "t" is CEIL(SYMBOL TABLE USED/4). This statement means that if you:

(a) Decrease the region available to PL/C by rK
(b) Specify TABSIZE=t on the /COMPILE card
(c) Change nothing else

and then rerun the program, you will get the same results. Specifically, you will not run out of core (assuming the original program did not).

PL/C CORE USAGE DIAGRAM

```
               |     COMPILATION TIME      |         |  EXECUTION |
               |            | |            |         |    TIME    |
               | SYNTACTIC  | |            |         |            |
               | & SEMANTIC | |    CODE    |         |            |
               |  ANALYSIS  | | GENERATION |         |            |
        -------+------------+-+------------+---------+------------+
         A     |   SYMBOL  || |            |    A    |            |
         |     |   TABLE   || |   SYMBOL   |    |    |   SYMBOL   |
    TABSIZE*4  |           || |   TABLE    |    |    |   TABLE    |
   BYTES    (OR|          V| |            |    |    |            |
   1/2 OF TOTAL|            | |            |    |    |            |
   IF  TABSIZE |            | +------------+    |    +------------+
   IS NOT SPEC-|            | |   OBJECT | |    |    |            |
   IFIED) - MAX|           A| |    CODE  | |    |    |   OBJECT   |
   128K BYTES. |           || |          | |    |    |    CODE    |
         |     |   MACRO   || |          V |    |    |            |
         V     |DEFINITIONS|| |            |    |    |            |
        -------+------------+ |            |  TOTAL  +------------+
               |INTERMEDIATE| |            | STORAGE |   STATIC   |
               |    CODE   || +------------+    |    |    CORE    |
               |           || |            |    |    |            |
               |           || |INTERMEDIATE|    |    +------------+
               |          V| |    CODE    |    |    | AUTOMATIC ||
               |            | |            |    |    |    CORE   ||
               |            | |            |    |    |          V|
               |          A| +------------+    |    |            |
               |           || |            |    |    |          A|
               |  DYNAMIC  || |  DYNAMIC   |    |    |  DYNAMIC  ||
               |    CORE   || |    CORE    |    V    |    CORE   ||
               +------------+-+------------+---------+------------+
```

Note: This diagram is slightly simplified

EFFICIENT PROGRAMMING IN PL/C

PL/C was designed to emphasize speed of compilation rather than execution. If execution is substantial it may be worthwhile running under the PL/I (F) compiler once the program has been thoroughly checked out. However, if a program is to be run under PL/C and the execution time is significant, there are a number of options and devices that can be employed to improve execution speed. These all have the effect of suppressing or disabling some of the diagnostic provisions that are normally compiled into PL/C programs. While this will improve execution speed it obviously reduces the degree of protection and the amount of information provided. Execution speed will be increased by each of the following:

(1) Disabling the FLOW condition. This is done by giving a (NOFLOW): prefix for each external procedure. Even if the FLOW events are not being printed (as a result of execution of a FLOW statement) the tracing code is present and active if the FLOW condition is enabled. (Similarly the CHECK condition must be disabled--by removing CHECK prefixes. Suppressing printing with the NOCHECK statement does not eliminate the checking code.)

(2) Disabling the SUBSCRIPTRANGE condition. This is done by giving a (NOSUBSCRIPTRANGE): prefix on each external procedure. This is only possible if the NOMONITOR=(SUBRG) suboption is specified. Note that some installations inhibit (override) this suboption since subscript testing is vital to ensure the integrity of the compiler. If run in NOMONITOR=(SUBRG) mode, the elimination of subscript testing will make a substantial improvement in the execution of a program with frequent references to subscripted variables.

(3) Disabling the SIZE condition. This is done by giving a (NOSIZE): prefix on each external procedure.

(4) Specifying the NOMONITOR=(UDEF) suboption. This will eliminate the code required to test for use of uninitialized variables. All variables (including strings) will be initialized to (hex) zero.

In order to limit the amount of printing from a PL/C program, consider the use of the CMPRS, NODUMP, FLAGE, NOHDRPG, NOOPLIST, and NOSOURCE options. To reduce the amount of printing during execution of the program, consider using the PUT OFF and PUT ON statements in sections that are not of current interest.

Because of the differences in internal representation in PL/C, there are certain operations that are relatively inefficient (compared to PL/I (F)). These are RECORD I/O, bit-string operations, and the UNSPEC and TRANSLATE built-in functions. To the extent that a program can avoid use of these features, its execution speed relative to PL/I (F) will be improved.

INTERNAL STRUCTURE OF PL/C

The material in this section has been included to offer a very brief idea of how the PL/C compiler is organized and how it operates. More complete descriptions of compilers in general, and PL/C in particular, are listed in the section "PL/I Bibliography."

A compiler is a computer program that translates a problem description from one language to another. The initial language is called the _source language_, in this case PL/I or PL/C, and the final language is called the _object language_, in this case the internal machine language of the Amdahl 470 computer. The compiler is a program that reads the source-language statements in as data and produces the object-language instructions as output. Most compilers will actually deliver this output to the user, e.g., on punched cards. The PL/C compiler, on the other hand, retains this output in computer storage, and once the compilation process is completed begins the execution of the object program that has just been produced. This type of compiler is called a "compile-and-execute" system. PL/C is unique in that it always produces a syntactically correct object program and executes it regardless of errors that may have been made in the source program.

Compilers can be used in many different environments, and for many very different purposes. A compiler that must serve many purposes is necessarily a compromise and is probably not optimal for any one particular use. The computer manufacturers, of necessity, must supply general-purpose compilers. Their bias is generally in the direction of producing efficient object programs. That is, the compiler is designed to produce an object program that will consume as little computer time as possible in its execution, even though the production of this relatively efficient object program will take more computer time during compilation. For production programs that will have a long life and be repeatedly executed, this makes admirable sense. However, for teaching purposes and program testing, where the execution is very brief and usually not ever repeated, the translation process dominates the situation. Moreover, in this environment a program may be expected to contain errors (sometimes more than a few), and the translation process must be repeated until all of those errors have been detected and eliminated in the source program. By concentrating on this particular environment, it is possible to produce a specialized compiler that will translate relatively rapidly, will not become unduly flustered when errors in the source program are encountered, and will provide the programmer with as much assistance as possible in the detection and correction of those errors. PL/C was designed to be just such a compiler.

The heart of the compiler is a program section named CONTROL (CONT). When MTS detects that it has a PL/C program to translate and execute, it loads a copy of the PL/C compiler and passes control to CONT. If given, CONT analyzes the /COMPILE card to confirm that it is in fact a PL/C program, and to determine what options the user requires. CONT then passes control to a program section called SYNTACTIC ANALYSIS (SYNA). SYNA'S task is to make a complete pass over the source program while doing the following:

(1) Printing a copy of the source program.

(2) Constructing an internal representation, called <u>beta-code</u>, of the source program so that the compiler never again has to deal with the card images of the actual source program. (Some compilers make more than a score of passes over these card images.)

(3) Constructing a region of storage called a symbol descriptor for each identifier in the source program. This space is used to record all of the various attributes of the identifier which the compiler must have to correctly manipulate the values represented by the identifiers.

(4) Detecting and repairing as many errors in the source program as possible, reporting each such action to the user by means of messages on the source program listing, and reflecting each such action in corrections to the beta-code.

SYNA scans the source program by calling upon a subroutine called LEXICAL ANALYSIS (LEXI) to deliver the next element (token) from the source program. LEXI is responsible for the tasks of reading cards, breaking the character strings that the cards contain down into distinct symbols, and encoding each symbol in a way that is convenient for SYNA to handle. In the process, LEXI takes care of comments, which are never seen by SYNA, and prints the source program listing. LEXI insulates SYNA from the actual form of the source program and permits SYNA to deal entirely with a continuous stream of highly encoded symbols.

Macros are processed by the MDEF program, which is called when LEXI encounters a /MACRO card. MDEF reads in the macro definition and enters the macro name in a dictionary of macros defined so far. When MDEF finds a /MEND card (or suspects that one is missing), it returns to LEXI, giving it the new list of macro names. LEXI uses this list to look for macros as it is processing the program source code. As soon as it finds a macro name, LEXI calls the macro expansion program MEXP which generates the text of the macro call and passes this back to LEXI. SYNA is thereby shielded from macros entirely.

Because the PL/I language permits identifiers to be used before they are declared, SYNA must be prepared to deal with identifiers before it has complete knowledge of what they represent. This means that on the first pass over the source program, SYNA can detect syntactical (grammatical) errors, but cannot ensure that a syntactically correct construction has any meaning. For example, SYNA can ensure that the keyword GOTO is followed by an identifier, but it may develop later in the program that that particular identifier refers to a floating-decimal variable and hence the GOTO statement does not make any sense. This "use before declare" possibility in PL/I makes it impossible for a single pass over the source program to detect all "static" errors of construction. To avoid this difficulty, and permit a single-pass scanner, many specialized compilers for PL/I subsets have ruled out "use before declare." PL/C does not place this restriction on the programmer, and as a consequence a second scanning pass is required.

When the first pass over the source program is completed, SYNA returns control to CONT, which in turn passes control to a section called SEMANTIC ANALYSIS (SEMA). SEMA is responsible for making a second pass over the source program, but does so entirely from the internal beta-code representation rather than from the original card images. Since all of the declarations have been processed by the time that the SEMA pass begins, SEMA can check to see that the attributes of each identifier are correct for the context in which that identifier appears. Even SEMA cannot do a complete job of this since some identifiers (for example, parameters) assume attributes "dynamically" in program execution, and checking must be performed during the execution of the object program. Since the same identifier can be used repeatedly in PL/I and have different meanings in different scopes, it is also the task of SEMA to "resolve" each identifier reference to determine exactly which "object" it refers to. Also during this pass, SEMA "parses" expressions into a "postfix operator notation" that facilitates the generation of machine instructions in the next pass. See R. Conway and D. Gries, An Introduction to Programming, A Structured Approach Using PL/I and PL/C-7 (Winthrop, 1975.) SEMA is concerned only with the identifiers and expressions in beta-code and modifies these, producing an internal representation of the source program called gamma-code.

When the second pass is completed, SEMA returns control to CONT, which in turn passes control to a program section called CODE GENERATION (CGEN). CGEN translates gamma-code into actual machine language instructions for the Amdahl 470 computer. This is a rather complicated process in a "block structured" language with a rich variety of data-types. The technique employed by PL/C is unusual and interesting, but it is beyond the scope of this brief section. It is described in the Cornell University Research Report by T. R. Wilcox entitled, "Code Generation in PLC."

Code generation in some compilers includes a task called "optimization." This involves such exercises as the search for common expressions, loop analysis to move operations to the outermost level possible, and very careful allocation of machine registers. Careful optimization can make very significant improvements in the efficiency of the object program, but it also significantly increases the time required to compile the program. PL/C CGEN does not include optimization. Considering the environment in which PL/C is primarily used, a process that increases compilation time to achieve a reduction in execution time did not seem appropriate. PL/I (F) is not an optimizing compiler either. Some PL/I programs will actually execute faster under PL/C than under PL/I (F), but typically the opposite is true. This is primarily due to various diagnostic features built into the PL/C object program, rather than an inherent inefficiency of the object program itself. The most important, and expensive, of these features is the monitoring of subscript values (which the user can suppress with the NOSUBSCRIPTRANGE prefix to improve execution efficiency). The essential compatibility between PL/C and PL/I makes optimization in PL/C even less attractive, since when a program reaches the stage where execution time is becoming important, one can switch to an optimizing compiler. PL/C is a true compiler, even though the degree of diagnostic assistance provided by PL/C is usually associated only with a type of translator called an "interpreter." Interpretive execution is at least ten (and sometimes as much as thirty) times as time-consuming as the execution of a compiled-object program.

When the generation of the object program is complete, CGEN returns control to CONT which, in turn, passes control to the EXECUTION SUPERVISOR (EXEC). EXEC begins the execution of the object program. The object program intermittently calls upon EXEC for diagnostic services, for built-in functions, and for I/O services. These latter tasks require the presence of other program sections which are in fact major pieces of the compiler. The I/O service module, in turn, calls upon LEXICAL ANALYSIS to scan data cards. When the execution of the object program terminates (or is terminated), EXEC returns control to CONT. If CONT is sitting with another /COMPILE card in hand, it reinitializes the compiler and begins the process all over again. If not, CONT returns to MTS.

PL/C can also be characterized as a "memory-resident compiler." This refers to the fact that no use is made of auxiliary storage (tape or disk) during the compilation process, and that major sections of the compiler itself remain in memory along with the object program. The entire compiler remains resident throughout execution.

The PL/C compiler was written in IBM 360/370 assembly language. Extensive use was made of the macro capability in the assembler but this could not be considered a "translator-writing system" in the sense described by Gries. This is an old-fashioned way to produce a compiler, but at the current state-of-the-art, it may be the only way to obtain a high-performance compiler.

PL/I BIBLIOGRAPHY

The following bibliography lists material that is relevant to programming in PL/I and PL/C. Most of these books are available in the University of Michigan library system.

Textbooks Related to PL/I and PL/C

Abel, P. Structured Programming in PL/I and PL/C, Reston, 1981.

Anger, A. L. Computer Science: The PL/I Language, Wiley, 1972.

Bates, F., and Douglas, M. Programming Language/One, 3d ed., Prentice-Hall, 1975.

Bauer, C., et al. Basic PL/One Programming, Addison-Wesley, 1968.

Bohl, M., and Walter, A. Introduction to PL/I Programming and PL/C, Science Research Associates, 1973.

Brown, G. D. FORTRAN to PL/I Dictionary, PL/I to FORTRAN Dictionary, Wiley, 1975.

Cassel, D. Programming Language One, Reston, 1972.

Clark, F. J. Introduction to PL/I Programming, Allyn and Bacon, 1971.

Cole, R. W. Introduction to Computing, McGraw-Hill, 1969.

Conway, R. Primer on Disciplined Programming, Winthrop, 1977.

------. Programming for Poets, Winthrop, 1977.

Conway, R., and Gries, D. An Introduction to Programming: A Structured Approach Using PL/I and PL/C, 2d ed., Winthrop, 1975.

------. A Primer on Structured Programming, Winthrop, 1976.

Davidson, M. PL/I Programming With PL/C, Houghton Mifflin, 1973.

Edwards, L. E. PL/I for Business Applications, Reston, 1973.

Fike, C. T. PL/I for Scientific Programmers, Prentice-Hall, 1970.

Germain, C. B. PL/I for the IBM 360, Prentice-Hall, 1972.

Groner, G. F. PL/I Programming in Technological Applications, Wiley, 1971.

Hughes, J. K. PL/I Programming, Wiley, 1973.

Hume J., and Holt, R. C. Structured Programming Using PL/I and SP/k, Reston, 1975.

Katzen, H. A PL/I Approach to Programming, Auerback Publishers, 1972.

Kennedy, M., and Solomon, M. B. Eight Statement PL/C (PL/ZERO) Plus PL/ONE, Prentice-Hall, 1972.

------. Structured PL/ZERO Plus PL/ONE, Prentice-Hall, 1977.

Kieburtz, R. B. Structured Programming and Problem-Solving with PL/1, Prentice-Hall, 1977.

Lecht, C. P. The Programmer's PL/I: A Complete Reference, McGraw-Hill, 1968.

Kochenburger, R. J., and Turcio, C. J. Introduction to PL/1 and PL/C Programming, Hamilton, 1974.

Maisel, H. Computers: Programming and Applications, Science Research Associates, 1976.

Marcotty, M. Structured Programming With PL/I, Prentice-Hall, 1977.
Mazlack, L. J. PL/C Essentials, McGraw-Hill, 1977.
Meek, B. Fortran, PL/I and the Algols, North-Holland, 1978.
Mott, T. H., et al. Introduction to PL/I Programming for Library and Information Science, Academy Press, 1972.
Neuhold, E., and Lawson, H. The PL/I Machine: An Introduction to Programming, Addison-Wesley, 1972.
Pollack, S. V., and Sterling, T. D., A Guide to PL/I, 3rd ed., Holt, Rinehart and Winston, 1980.
------. Essentials of PL/I, Holt, Rinehart and Winston, 1974.
Rich, R. P. Internal Sorting Methods Illustrated with PL/I Programming, Prentice-Hall, 1972.
Richardson, G., and Birkin, S. Problem Solving Using PL/C, Wiley, 1975.
Roper. PL/I in Easy Stages, Dickenson, 1976.
Scott, G. L., and Scott, J. PL/I, A Self-Instructional Manual, Dickenson, 1969.
Scott, R. C., and Sondak, N. E. PL/I for Programmers, Addison-Wesley, 1970.
Shortt, and Wilson. Problem Solving and the Computer: A Structured Concept with PL/I (PL/C), Addison-Wesley, 1976.
Smith, C. L., and Murrill, P. W. PL/I Programming with PL/C, Intext Educational Publishers, 1973.
Sprowls, R. C. PL/C: A Processor for PL/I, Canfield, 1972.
Sprowls, R. C. Introduction to PL/I Programming, Harper & Row, 1969.
Walker. Fundamentals of PL/I, Programming with PL/C, Allyn and Bacon, 1975.
Weinberg, G. M. PL/I Programming Primer, McGraw-Hill, 1966.
------. PL/I Programming: A Manual of Style, McGraw-Hill, 1970.
Weinberg, G. M., et al. Structured Programming in PL/C, Wiley, 1973.

PL/C References

PL/C Release 7.6 Installation Instructions, Department of Computer Science, Cornell University, Ithaca, New York.
Conway, R. W., and Wilcox, T. R. "The Design and Implementation of a Diagnostic Compiler for PL/I," Communications of the ACM, 16:3, March 1973.
Dunigan, T. H. PLCD, PL/I for the DEC PDP-11/45, University of North Carolina, 1973.
Dunigan, T. H. and Kehs, D. R., User's Guide to PLCD, University of North Carolina, 1975.
Kehs, D. R. Extensions to the PLCD Compiler, University of North Carolina, 1974.
Moore, C. G., and Conway, R. W. "PL/CT - An Interactive Terminal Version of PL/C," Department of Computer Science and Office of Computer Services, Cornell University, Ithaca, New York.
Morgan, H. L. "Spelling Correction in Systems Programs," Communications of the ACM, 16:2, February 1973.
Uzgalis, R., et al. "Compiler Measures in the Perspective of Program Development," Sixth Hawaii International Conference on System Science, 1973.
Wagner, R. A. "Common Phrases and Minimum Space Text Storage," Com-

munications of the ACM, 16:3, March 1973.

Wilcox, T. R. Code Generation in PL/C, Research Report No. 70-89, Department of Computer Science, Cornell University, Ithaca, New York.

Wolfe, J. M. User's Guide to PL/C, Department of Computer and Information Science, Brooklyn College, City University of New York, New York, New York.

IBM PL/I (F) Publications

IBM System/360 Operating System PL/I (F) Language Reference Manual, form GC28-8201.

IBM System/360 Operating System PL/I (F) Programmer's Guide, form GC28-6594.

IBM System/360 PL/I (F) Subroutine Library, Computational Subroutines, form GC28-6590.

IBM PL/I Optimizing Compiler Publications

OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, form GC33-0009.

OS PL/I Optimizing Compiler: General Information, form GC33-0001.

OS PL/I Optimizing Compiler: Programmer's Guide, form SC33-0007.

OS PL/I Optimizing Compiler: Execution Logic, form SC33-0025.

OS PL/I Optimizing Compiler: Messages, form SC33-0027.

INDEX

Reader’s Comment Form

PL/I in MTS
Volume 7
September 1982

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or BSAD.

Date ____________________

Name ____

Address ____

____

____

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109

Update Request Form

PL/I in MTS
Volume 7
September 1982

Updates to this manual will be issued periodically as errors are noted or as changes are made to MTS. If you desire to have these updates mailed to you, please submit this form.

Updates are also available in the memo files at both the Computing Center and NUBS; there you may obtain any updates to this volume that may have been issued before the Computing Center receives your form. Please indicate below if you desire to have the Computing Center mail to you any previously issued updates.

Name ____

Address ____

____

____

Previous updates needed (if applicable): __________

The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or BSAD. Campus Mail addresses should be given for local users.

Publications
Computing Center
The University of Michigan
Ann Arbor, Michigan 48109

Users associated with other MTS installations (except the University of British Columbia) should return this form to their respective installations. Addresses are given on the reverse side.

Addresses of other MTS installations:

Publications Clerk
352 General Services Bldg.
Computing Services
The University of Alberta
Edmonton, Alberta
Canada T6G 2H1

Information Officer, NUMAC
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne
England NE1 7RU

Rensselaer Polytechnic Institute
Documentation Librarian
310 Voorhees Computing Center
Troy, New York 12181

Simon Fraser University
Computing Centre
User Services Information Group
Burnaby, British Columbia
Canada V5A 1S6

Wayne State University
Computing Services Center
Academic Services Documentation Librarian
5950 Cass Ave.
Detroit, Michigan 48202