File Number S360-29
Order No. GC24-9005-6
DOS Release 25

DOS
TOS

Systems Reference Library

IBM System/360
Disk and Tape Operating Systems
PL/I Programmer's Guide

This publication complements the Systems Reference Library publication IBM System/360, PL/I Subset Reference Manual, Order No. GC28-8202. Its purpose is to aid the programmer and to familiarize him with the techniques of PL/I programming. This publication therefore provides all information that is not part of the PL/I Subset Reference Manual but required by the programmer to write programs in the PL/I Subset language and to have them compiled and executed in the DOS/TOS environment.

The main topics covered in this publication are:

- The DOS/TOS environment
- PL/I data file organization
- Storage requirements of PL/I programs and program elements
- The overlay facility
- Listings produced for PL/I programs
- Restrictions to the PL/I Subset language

In some instances, the programmer may desire detailed additional information on topics not directly connected with PL/I. A list of all pertinent Systems Reference Library publications is provided in the Introduction section of this publication.

SUMMARY OF CHANGES

The DOS system now provides for the support of private core-image libraries. In a DOS system supporting the batched-job foreground and private core-image libraries, the compiler and the linkage editor may execute in a foreground or in the background partition.

A description of errors that may arise due to multiple secondary entry points or due to multiple CSECT names has been added.

When processing INDEXED files, the KEY condition may arise in a number of cases. It is described how the programmer may identify specific situations at execution time.

A brief description of hardware stops caused by severe programming errors has been included.

Also added was an appendix containing programming examples together with explanations.

Seventh Edition (April, 1971)

This is a major revision of GC24-9005-5.

Changes to the text and to the illustrations as well as additions are indicated by a vertical line to the left of the change or addition.

This edition applies to change level 3-9 of the DOS PL/I compiler (DOS release 25) and change level 2-3 of the TOS PL/I compiler (TOS release 14) and to all subsequent levels until otherwise indicated in new editions or Technical Newsletters.

Changes are continually made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/360 SRL Newsletter, Order No. GN20-0360, for the editions that are applicable and current.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 printer using a special print chain.

Requests for copies of IBM publication should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Laboratories, Programming Publications, 7030 Boeblingen/Germany, P. O. Box 210.

Contents

Introduction

This publication complements the Systems Reference Library publication IBM System/360, PL/I Subset Reference Manual, Order No. GC28-8202 (hereafter referred to as the Subset Reference Manual). It provides all information that is not part of the language specifications but required by the programmer to write programs in the PL/I Subset language and to have them compiled and executed in the DOS/TOS environment.

This publication is divided into four logical parts:

Part I - provides all information regarding the DOS/TOS environment, PL/I data file organization including the ENVIRONMENT attribute, linkage between PL/I and Assembler modules, and PL/I programming in the DOS/TOS environment.

Part II - provides all information regarding storage requirements of programs written in the PL/I Subset language, and a description of the overlay facility.

Part III - describes all listings and diagnostic messages produced for PL/I programs running under DOS/TOS control.

Part IV - Appendix. Some of the individual appendixes provide information taken out of the corresponding sections to improve the readability, e.g., a list of all available I/O subroutines. The remaining appendixes furnish additional reference information the PL/I programmer might find useful.

The last section of the Appendix lists the implementation-dependent restrictions to the PL/I Subset language as it is described in the Subset Reference Manual. The individual restrictions are listed in alphabetical order.

To free the programmer of the necessity of referring to other publications for additional information, this publication is made as self-supporting as possible by duplicating some of the information given elsewhere. However, should this publication not give all the details the programmer needs for solving his problem, these details can be found in the pertinent SRL publication.

A list of all SRL publications the programmer may have to refer to is given below:

IBM System/360 Disk Operating System, System Programmer's Guide, Order No. GC24-5073

IBM System/360 Operating System, PL/I Library Computational Subroutines, Order No. GC28-6590

IBM System/360 Principles of Operation, Order No. GA22-6821

IBM System/360 Disk and Tape Operating Systems, Concepts and Facilities, Order No. GC24-5030

IBM System/360 Disk and Tape Operating Systems, Utility Program Specifications, Order No. GC24-3465

IBM System/360 Disk Operating System, System Control and System Service Programs, Order No. GC24-5036

IBM System/360 Tape Operating System, System Control and System Service Programs, Order No. GC24-3431

IBM System/360 Disk Operating System, Supervisor and Input/Output Macros, Order No. GC24-5037

IBM System/360 Tape Operating System, Supervisor and Input/Output Macros, Order No. GC24-3432

IBM System/360 Disk Operating System, Data Management Concepts, Order No. GC24-3427

IBM System/360 Tape Operating System, Data Management Concepts, Order No. GC24-3430

IBM System/360 Disk Operating System, PL/I DASD Macros, Order No. GC24-5059

Minimum Requirements for Compilation

1. 16,384 (16K) bytes of core storage on one of the compatible models of System/360 (not Model 20, 44). The compiler itself requires 10K. More than 10K are required if SYSIPT, SYSLST, and/or SYSPCH are DASD files. This is a system generation option.

2. a. Either one IBM 2311 Disk Storage Drive or one IBM 2314 or 2319 Direct Access Storage Facility or

b. four IBM Magnetic Tape Drives of the series 2400, 2420, or 3420. A 7-track tape may be used for SYSRES. The use of a 9-track tape for SYSRES will improve the performance. The data conversion feature is required for 7-track drives. One additional tape drive is required for compile-and-go operation.

3. One card read/punch or one card reader and one card punch.

4. One printer.

5. One IBM 1052 Printer-Keyboard (required for operator-to-system communication).

6. The optional supervisor feature Program Interrupt (PC).

Note: Either one or both of the units listed under items 3 and 4 may be replaced by one additional magnetic tape drive per replaced unit.

The speed of compilation is greatly reduced if (1) the source program contains more than 80 programmer-defined identifiers, and (2) a 16K system is used to compile a program greater than 16K.

For determination of the required workfile space refer to Workfile Requirements in Appendix G of IBM System/360 Disk Operating System, System Generation and Maintenance, Order No. GC24-5033.

Minimum Requirements for Execution

The execution-time requirements depend on the requirements of the system and the object program.

Additional machine features required for arithmetic, compare, and conversion are listed in Figure 1.

Note: At EXEC time all IJKSnn transients must be available in the core-image library.

Maximum Configuration Supported

The following units and features are supported:

1. All of the units and features specified for compilation. (Disk files are not supported for tape-resident systems.)

2. All of the following devices:
   IBM 2540*
   IBM 1403
   IBM 3211
   IBM 1404 (for continuous forms only)
   IBM 1442N1
   IBM 1442N2
   IBM 1443
   IBM 2501
   IBM 2520B1
   IBM 2520B2
   IBM 2520B3
   IBM 1445
   IBM 2321

   *The Punch/Read Feed (PRF) special feature is not implemented by PL/I.

3. Additional main storage up to 16 million bytes.

Comparison Of/With Arithmetic With/And Convert To	Coded Fixed Decimal	Fixed Binary	Coded Float	Numeric Fixed, Sterling	Numeric Float	Bit	Char.
From							
Coded fixed decimal	D	D,F[2]	D,F	D	D,F	D	NP
Fixed binary	D,F[2]	X	F	D,F[2]	D,F	X	NP
Coded float	D,F	F	F	D,F	D,F	F	NP
Numeric fixed and sterling	D	D,F[2]	D,F	D	D,F	D	X[1]
Numeric float	D,F	D,F	D,F	D,F	D,F	D,F	X[1]
Bit	D	X	F	D	D,F	X	X
Character	NP	NP	NP	NP[1]	NP[1]	X	X

D - Decimal feature required.
F - Floating-point feature required. Conversion only.
NP - Not permitted.
X - No special features required.

[1] - Conversion only.
[2] - Floating-point feature only if scale factor not equal to zero.

Figure 1. Additional Machine Feature for Arithmetic, Comparison, or Conversion

Running Programs Under DOS/TOS Control

This section describes the compilation and execution of PL/I programs under control of the Disk and Tape Operating Systems. The pertinent terminology, control statements, and their formats are discussed when required.

Basic Terminology

It is convenient to refer to each stage of program development by a particular name, because just the term program would be too general and, therefore, confusing.

In program development, the programmer writes sets of source statements that may form a complete program or part thereof. A card deck containing one external procedure written in the PL/I Subset language is referred to as a source module. A source module is the unit that is processed during a compilation. The compilation results in one or two object modules. The first object module is produced by the PL/I compiler for all of the file declarations, if any, contained in the source module. The second object module is produced for the source module. Object modules can be loaded by the DOS/TOS Linkage Editor program and then executed. An object module consists of standard ESD (External Symbol Dictionary), TXT (Text), RLD (Relocation Dictionary) cards, and one END card.

To start the execution of a PL/I program, control must be transferred from the Disk or Tape Operating System to the object program. The external procedure to which control is transferred from the Job Control program must have the option MAIN.

Some parts of the object program may not be required in storage throughout its execution. External procedures that are never active simultaneously may use the same storage area to save storage. Each part of the program that is in storage only for a fraction of the execution time is referred to as an overlay. Using the MAIN procedure as an overlay is not permitted. Each overlay as well as that part of the program that resides in storage throughout the execution of the object program is referred to as a phase. A phase consists of one or more external procedures. For detailed information refer to the sections Overlay and The Linkage Editor Program.

Some standard procedures such as PL/I built-in functions or conversion subroutines have been incorporated into the relocatable library as library subroutines. Only the code required for calling these subroutines is compiled into the object module. The library subroutines themselves are incorporated into the appropriate phases by the autolink feature of the DOS/TOS Linkage Editor program.

Extra code is required to allow some housekeeping during the execution of a PL/I program. This code, which is referred to as overhead, may either be generated in-line in an object module or incorporated due to an explicit library subroutine call.

The relationship between the user's PL/I mainline program, the PL/I control program, and the DOS/TOS system is shown in Figure 2.

Note: The PL/I control program is a set of library routines in the relocatable library which are included into object programs at linkage-edit time and perform certain control functions at execution time.

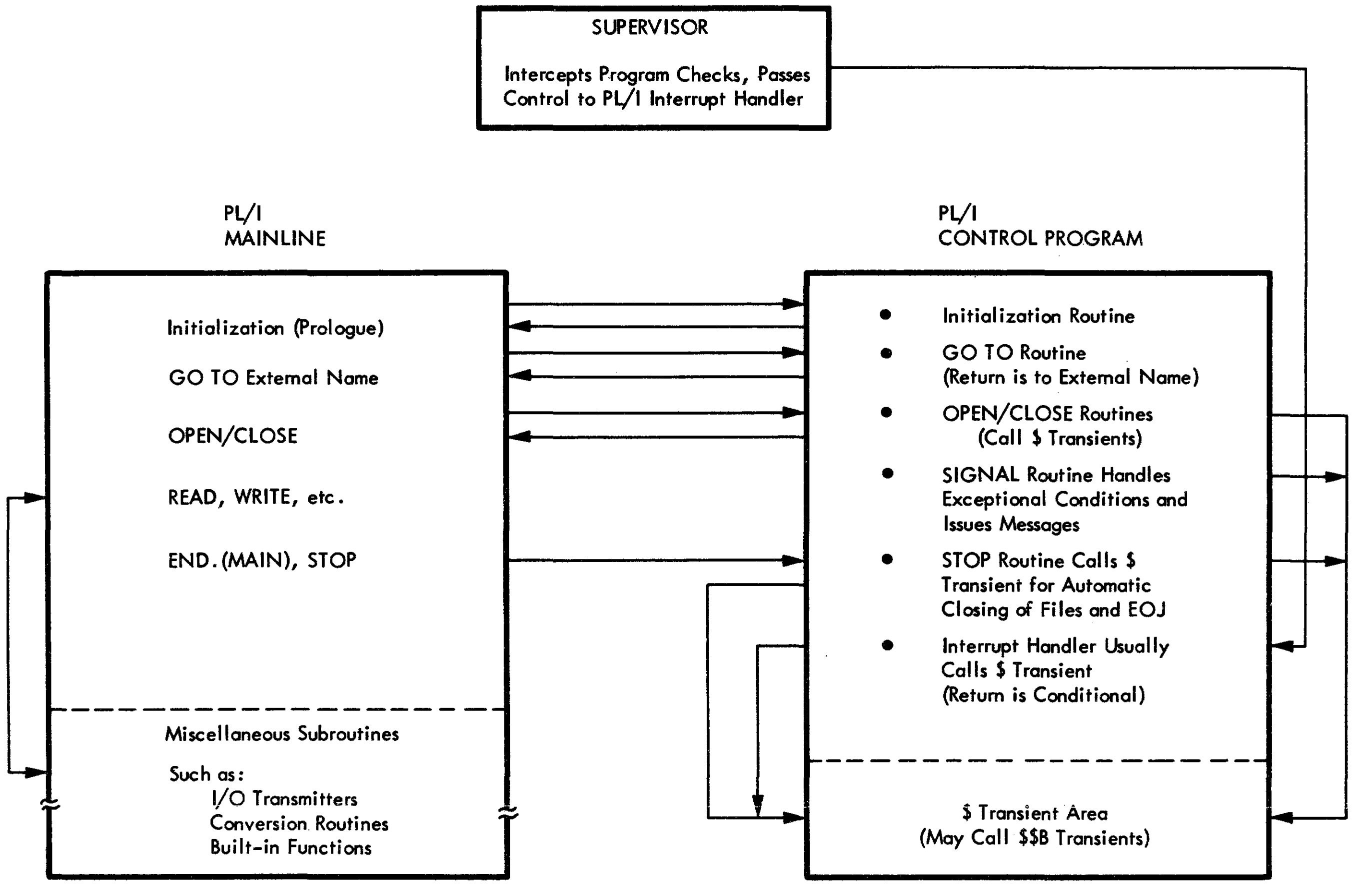


Figure 2. PL/I Program Structure

Object-Time Storage Layout

The layout of main storage during execution of a PL/I program is shown in Figure 3.

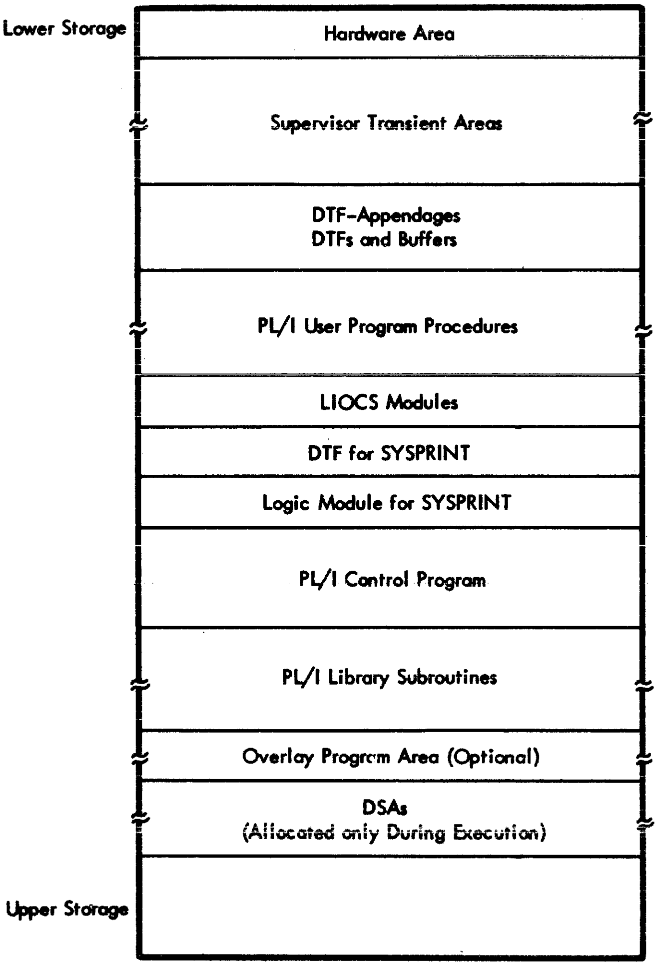


Figure 3. Object-Time Storage Layout

The Disk And Tape Operating Systems

The Disk and Tape Operating Systems (DOS/TOS) are a group of processing programs with the control and service programs required to maintain continuous operation. They are self-contained systems and require a minimum of operator intervention.

The processing programs consist of language translators and service programs. The group of processing programs can be expanded by adding user-written problem programs.

The system control program -- the frame work of DOS/TOS -- consists of three components:

- the Supervisor program,
- the Job Control program, and
- the Initial Program Loader (IPL).

These components are used to load the system and to prepare and control the execution of all processing and problem programs within the system.

The system service programs consist of the Linkage Editor and the Librarian. These programs are used to bring compiled source programs into an executable format and to maintain the libraries.

Figure 4 shows a schematic representation of the Disk and Tape Operating Systems.

To make full use of DOS/TOS, the user should be familiar with (1) the functions of the individual system components and (2) the interaction of these components. Users of the overlay feature should be thoroughly familiar with the DOS/TOS Linkage Editor program. Users of the label-processing facilities should be familiar with DOS/TOS data management concepts. This section briefly discusses those parts of the DOS/TOS that are of interest to users of the PL/I Subset language.

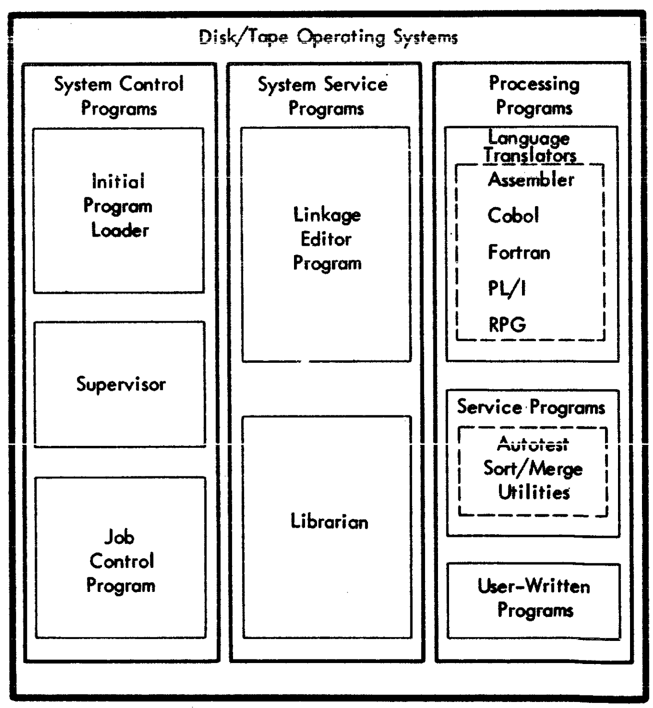


Figure 4. Schematic Representation of the Disk and Tape Operating Systems

System Control Programs

The Supervisor handles all hardware interrupts, causes I/O operations to be performed, and contains a fetch routine for fetching program phases from the core-image library. The Supervisor resides in storage throughout the execution of all IBM-supplied and user-written programs.

The Job Control program provides job-to-job transition within DOS/TOS. It performs its functions between job steps and does not reside in storage while a problem program is being executed.

The IPL is of no interest to the PL/I programmer.

System Service Programs

The Linkage Editor links all relocatable object modules that are produced by the language translators, i.e., it assigns absolute addresses and resolves cross-references between different object modules (external symbols). The output of the Linkage Editor can be either immediately executed or incorporated into the core-image library.

The Librarian is a group of programs used for maintaining the libraries and providing printed and/or punched output from these libraries.

The libraries are:

the system core-image library
the system relocatable library
the system source statement library
private core-image libraries
private relocatable libraries
private source-statement libraries

The core-image library contains object-program phases already processed by the Linkage Editor. These programs are ready for execution under control of the Supervisor. The core-image library contains, for instance, the system control and service programs themselves and the PL/I compiler.

A DOS system generation option provides for the support of private core-image libraries on the same type of direct access device as the system residence volume. A private core-image library has the same format and function as the system core-image library on the system residence volume. When searching for a program to be loaded, the system loader searches the private core-image library first, if assigned, and then the system core-image library.

The relocatable library contains object modules produced by the language translators. Object modules may be preceded by Linkage Editor control statements. The individual modules contained in the relocatable library are used as input to the Linkage Editor. Most of the built-in functions of PL/I as well as service routines required for the execution of PL/I object programs are contained in the relocatable library.

The source statement library is not used by the PL/I compiler or during object program execution.

Multiprogramming

DOS and TOS permit the switching of processing between one or two foreground programs and one background program, in which case all programs reside in storage simultaneously. This method increases the total throughput since some program may use the CPU while another program is waiting for input/output. If more than one program requires the CPU, the foreground-1 program has the highest and the background program the lowest priority. The program(s) of lower priority are dormant until the program(s) of higher priority start(s) waiting for a completion of input/output.

The storage areas - referred to as *partitions* - assigned to each of the three programs are defined at system generation time and may be changed by the operator between job steps.

In a DOS system which supports the batched-job foreground (MPS=BJF) and private core-image library options, the Linkage Editor can execute in either foreground partition (as well as the background partition) provided a minimum of 10K of storage is assigned to the partition. When executing in a foreground partition, a private core-image library must be assigned.

In a DOS multiprogramming environment described above, the DOS PL/I compiler can be executed in any partition in the following manner:

1. At system generation time, link edit the PL/I compiler in the background partition and place it in the system core-image library.

2. Link edit the PL/I compiler in any desired foreground partition and place it in a private core-image library assigned to that partition.

3. When executing the PL/I compiler in a foreground partition, assign the appropriate private core-image library.

Logical Device Address	Device Referred to
SYSRDR	Input device from which Job Control statements are read. Not used by PL/I compiler or object programs.
SYSIPT	Input device from which the input for the PL/I compiler is read. Can also be referred to by SYSIN.
SYSIN	Input device combining the functions of SYSRDR and SYSIPT.
SYSLST	Output device used by the PL/I compiler. The device used is the same as the PL/I standard output device for listing (SYSPRINT). (For PL/I object-time messages refer to PROCEDURE Statement in Appendix H.)
SYSPCH	Card punching device used by the PL/I compiler when a punched card object deck is specified.
SYSOUT	Output device combining the functions of SYSLST and SYSPCH. Cannot be assigned by an ASSGN statement.
SYSLNK	Input/output device used by the Linkage Editor and the PL/I compiler when compiling and subsequent link-editing is specified.
SYSCLB	A private core-image library. If such a library is assigned, the output from the Linkage Editor is placed here either permanently or temporarily. If such a library is not assigned, output from the Linkage Editor goes to the system core-image library.
SYSLOG	Console typewriter used for listing messages issued to the operator by the PL/I compiler and the object program. SYSLOG is also used when a DISPLAY statement appears in the PL/I program (For PL/I object-time messages refer to PROCEDURE Statement in Appendix J.)
SYS000 to SYS222	Logical device addresses available to the programmer (programmer logical units as opposed to the remaining units, which are also referred to as system logical units). SYS001, SYS002, and SYS003 are used as work file addresses by the language processors and the Linkage Editor. They may be used as work file or output file addresses, but the user must protect his input files from being destroyed by the compiler or Linkage Editor in the case of a compile-and-execute or link-and-execute job. For this purpose, he should use the DISPLAY statement with the REPLY option and instruct the operator to mount the input file immediately before opening the file at execution time if a sufficient number of I/O units is not available.

Figure 5. Logical Device Addresses Used by the PL/I Programmer

DOS programs compiled by the DOS PL/I compiler can be executed in a foreground partition, provided the supervisor was generated with the option MPS=BJF and a minimum of 10K of storage is assigned to the partition. PL/I object programs may only be executed in batched-job mode.

Under TOS, the PL/I compiler and the Linkage Editor exclusively work in the background partition. TOS programs compiled by the PL/I compiler cannot run as foreground programs.

I/O Device Assignment

The I/O devices used during compilation and execution are referred to by logical device addresses instead of by their physical device addresses. Thus, the user may disregard the physical device assignments of the system configuration he uses. Moreover, if a number of different system configurations is used, recompilation of a source program is required only if the device types (1442, 2540, etc.) change. The logical device addresses the PL/I programmer should know are listed in Figure 5.

Logical device addresses can be assigned to physical devices

1. when building the system,
2. by the operator, or
3. by means of the ASSGN statement (see the section The ASSGN Statement).

If multi-programming is included in the supervisor, independent sets of logical device addresses are provided for the background area and both foreground areas.

The Job Control Program

The Job Control program permits processing of batched jobs in background mode. A job is the execution of a problem and consists of one or more job steps. A job step is a single compilation of an external procedure, a Linkage Editor run, a Librarian run, or the execution of an object program.

JOB CONTROL STATEMENTS

The execution of the Job Control program is initiated by Job Control statements read from SYSRDR. The general format of Job Control statements is as follows:

1. Name
   Job Control statements are identified by two slashes (//) in columns 1 and 2. The second slash must be followed by one or more blanks. Exceptions are:

   a. The end-of-job statement contains /& in columns 1 and 2.

   b. The end-of-data-file statement contains /* in columns 1 and 2.

   c. The comments statement contains * in column 1 and a blank in column 2.

2. Operation
   The entry in the operation field of a Job Control statement describes the type of operation to be performed. It must be followed by one or more blanks.

3. Operand
   The operand may be blank or consist of one or more entries separated by commas. Interspersed blanks are not permitted. The last entry must be followed by one or more blanks unless its last character is in column 71.

4. Comments
   Comments are permitted anywhere after the trailing blank of the operand field.

The ASSGN Statement

The ASSGN statement is used to assign a logical device address to a physical device. The format of the ASSGN statement is as follows:

```
                                            ,X'ss'
// ASSGN SYSxxx,device-address
                                            ,ALT
```

SYSxxx is one of the logical devices listed in Figure 5 (with the exception of SYSOUT, which cannot be assigned by means of ASSGN statements). The system permits programmer logical units in the range from SYS000 to SYS222. The number of units actually permitted per partition in a specific installation is defined at system generation time and normally less than 223. SYS000 to SYS004 are the minimum provided by the system.

The following restrictions should be observed when re-assigning some of the logical units:

1. SYSRDR, SYSIPT, SYSIN, SYSLST, and SYSPCH cannot be assigned to 2311 or 2314 DASD extents by ASSGN statements. In case they are assigned to a 2311 or 2314 DASD extent either at system generation time or by the operator, a special version of the PL/I compiler that needs a minimum of 12K of storage for execution must have been cataloged at system generation time.

2. SYSLNK must be assigned to the same device type as SYSRES for DOS and to a magnetic tape drive for TOS. Any reassignments must be made before issuing an OPTION statement that contains the LINK or CATAL option.

3. SYSLOG should be assigned to a 1052 console typewriter. Assignment to a printer is possible but degrades the system functions and prevents the use of the DISPLAY statement with the REPLY option.

4. SYSCLB cannot be temporarily assigned (// in columns 1 and 2). The permanent assignment format must be used (ASSGN in columns 1 through 5).

5. SYS001 to SYS003 must be assigned to the same device type (either magnetic tape drives, 2319, 2311, or 2314 DASD extents) for the entire duration of a compilation.

Device-address permits three formats:

X'cuu' where c is the channel number and uu the unit number in hexadecimal notation.

UA Unassign. The job is canceled if a file attached to this logical unit is referred to by one of the I/O statements OPEN, CLOSE, GET, PUT, READ, WRITE, or REWRITE.

IGN Indicates the logical unit to be unassigned and that all program references to the logical device are to be ignored. All I/O commands issued to the file are ignored. The IGN option is not valid for SYSRDR, SYSIPT, and SYSIN. For PL/I files, the IGN-bit will be checked. And if the IGN-bit is on, no OPEN-bit will be set.

X'ss' is the device specification. It is used for specifying mode settings for 7-track and dual-density 9-track tapes. If X'ss' is not specified, the system assumes X'90' for 7-track tapes and X'C0' for 9-track tapes. The possible specifications for X'ss' are listed in Figure 6.

Note: When creating a 7-track tape file, the data-conversion feature must be off.

ss	Bytes per inch	Parity	Translate Feature	Convert Feature
10	200	odd	off	on
20	200	even	off	off
28	200	even	on	off
30	200	odd	off	off
38	200	odd	on	off
50	556	odd	off	on
60	556	even	off	off
68	556	even	on	off
70	556	odd	off	off
78	556	odd	on	off
90	800	odd	off	on
A0	800	even	off	off
A8	800	even	on	off
B0	800	odd	off	off
B8	800	odd	on	off
C0	800	single-density 9-track		
C0	1600	dual-density 9-track		
C8	800	dual-density 9-track		

Figure 6. Possible Specifications for X'ss' in the ASSGN Statement

ALT indicates an alternate magnetic tape unit that is used if the capacity of the original unit is reached. The characteristics of the original and the alternate unit must be the same. Multiple alternates may be assigned to one logical unit.

Note: All device assignments made with ASSGN statements are reset between jobs to the configuration specified at system generation time plus any modifications that may have been made by the operator. (See the section The JOB Statement.)

The EXEC Statement

The execution of a job step is initiated by the statement:

// EXEC name

Name is the name of the first phase of the program to be fetched from the core-image library and to be executed. Therefore, execution of a PL/I compilation would be initiated by the statement

// EXEC PL/I

The name must be omitted if a program linked in the previous job step of the same job is to be executed immediately after the link operation.

The JOB Statement

Each job begins with the statement:

// JOB job-name

Job-name is a user-defined name of 1 to 8 characters.

Note: The JOB statement cancels all previously issued OPTION and ASSGN statements.

The LISTIO Statement

The LISTIO statement is used to obtain a listing of the I/O assignments. The format of this statement is

// LISTIO

with one of the operands listed in Figure 7. The listing is produced on SYSLST. The listing varies according to the operand. For magnetic tape units, physical units are listed with current device specification.

Operand	Causes the Listing of
SYS	the physical units assigned to all system logical units.
PROG	the physical units assigned to all background programmer logical units.
ALL	the physical units assigned to all logical units.
SYSxxx	the physical units assigned to the specified logical unit.
UNITS	the logical units assigned to all physical units.
DOWN	all physical units specified as inoperative.
UA	all physical units not currently assigned to a logical unit.
X'cuu'	the logical units assigned to the specified physical unit.

Figure 7. Operands of LISTIO Statement and Corresponding Actions

The MTC Statement

The MTC statement is used to control operations on logical units assigned to magnetic tapes. The format of the MTC statement is

// MTC op-code,SYSxxx[,nn]

For further details refer to the section Multi-File Volumes and Backwards Files.

The OPTION Statement

The OPTION statement is used to specify options for the compilation of PL/I source programs. Its format is

```
// OPTION option1[,option2]....
```

If this statement is omitted, a set of standard options defined at system generation time will apply. If more than one OPTION statement is issued in one job, all further OPTION statements change only those options that are respecified. All other options will remain unchanged.

All options specified in the OPTION statement are canceled when a new JOB statement is read. (See the section The JOB Statement.)

The options LINK and CATAL are canceled

1. if severe or disastrous errors have been detected during a PL/I compilation.
2. after a new EXEC statement has been executed.

The options that may be used by the PL/I programmer are listed in Figure 8.

Option	Function
LOG	Causes all Job Control statements to be listed on SYSLST.
NOLOG	Suppresses the LOG option.
DUMP	Causes the contents of core storage and registers to be listed on SYSLST in case of an abnormal termination of the job.
NODUMP	Suppresses the DUMP option.
LINK	Causes the compiled PL/I program to be written on SYSLNK for later processing by the Linkage Editor. This option, if used, must precede all other Linkage Editor control statements, if any.
NOLINK	Suppresses the LINK option. The LINK option is also suppressed if a serious or disastrous error is detected during compilation of a PL/I source program or if an EXEC statement with a blank operand field is read.
CATAL	Causes the LINK option to be set. In addition, it causes the cataloging of a phase or program into the core-image library after either a /& or a // EXEC MAINT statement has been read.
DECK	Causes the PL/I compiler to punch an object deck if no disastrous compile-time error has been detected.
NODECK	Suppresses the DECK option.
LIST	Causes the PL/I compiler to list the source program on SYSLST.
NOLIST	Suppresses the LIST option.
LISTX	Causes the PL/I compiler to list the object program on SYSLST.
NOLISTX	Suppresses the LISTX option.
SYM	Causes the PL/I to list the symbol table, the block table, the offset table, and the external symbol table on SYSLST.
NOSYM	Suppresses the SYM option.
ERRS	Causes the PL/I compiler to list all detected errors on SYSLST.
NOERRS	Suppresses the ERRS option.
XREF	Causes the PL/I compiler to write a cross-reference listing on SYSLST.
NOXREF	Suppresses the XREF option.
48C	Informs the PL/I compiler that source programs are written in the 48-character set in EBCDIC notation. (No provision has been made for BCDIC and ASCII character sets.)
60C	Informs the PL/I compiler that source programs are written in 60-character set in EBCDIC notation.
MINSYS (TOS only)	Causes the Linkage Editor to produce minimum-size modules for later runs on systems with a background program area smaller than 24K, when link-editing on systems with a larger background program area.

Figure 8. Operands Used in the OPTION Statement

The PAUSE Statement

The PAUSE statement can be used to stop batched-mode processing in order to save output files produced by a previously executed program. Its format is

```
// PAUSE comments
```

The comments are printed on SYSLOG (provided SYSLOG has been assigned) to indicate the action to be taken by the operator.

The RESET Statement

The RESET statement resets I/O assignments to the standard assignments. The standard assignments are those specified at system generation time plus any modifications made by the operator by means of an ASSGN command (as opposed to using an ASSGN control statement) without the TEMP option. The format of the RESET statement is:

```
// RESET
```

with one of the operands SYS, PROG, ALL, SYSxxx. The meaning of the individual operands is described below.

SYS resets all system logical units to their standard assignments.

PROG resets all programmer logical units to their standard assignments.

ALL resets all programmer and system logical units to their standard assignments.

SYSxxx resets the specified logical unit to its standard assignment.

The UPSI Statement

This statement (User Program Switch Indicators) allows the user to set program switches that can be tested much the same as sense switches or lights used on other machines. The UPSI statement has the following format:

```
// UPSI nnnnnnnn
```

The operand consists of one to eight characters of 0, 1, or X. Positions containing 0 are set to 0. Positions containing 1 are set to 1. Positions containing X remain unchanged. Unspecified rightmost positions are assumed to be X.

Job Control clears the UPSI byte to zeros before reading control statements for each job. When Job Control reads the UPSI statement, it sets or ignores the bits of the UPSI byte in the communication region. Left to right in the UPSI statement, the digits correspond to bits 0 through 7 in the UPSI byte. Any combination of the eight bits may be tested by problem programs at execution time.

The DOS PL/I compiler checks bit 0 of the UPSI byte; the other bits are ingored.

If bit 0 is on (1) during compilation, Librarian and Linkage Editor statements are produced to permit to compile and catalog in one job step into the relocatable library. Bit 0 should be off (0) if cataloging into the relocatable library is not desired. For further details on cataloging refer to the section Cataloging into the Relocatable Library.

The End-of-Data-File Statement

The end-of-data-file statement (/* in columns 1 and 2) serves as a delimiter for the input read from SYSIPT. Therefore, PL/I programs must be terminated by an end-of-data-file statement. This statement is also recognized on the programmer logical units that are assigned to a card reader. This causes the ENDFILE condition to be raised for a PL/I input file.

The End-of-Job Statement

The end-of-job statement (/& in columns 1 and 2) indicates that a job has been completed. If this statement is omitted, the Job Control program may skip the next job stacked on SYSRDR and/or SYSIPT. If SYSRDR and SYSIPT are different units, the end-of-job statement must appear on both.

The Comments Statement

A special comments statement (* in column 1 and blank in column 2, followed by the desired comments) is available for longer messages. The comments are printed on SYSLOG, but no halt is caused by this statement.

File Label Job Control Statements

For all Job Control statements referring to disk and tape file labels see the section File Labels.

The PROCESS Statement

The PROCESS statement allows the programmer to specify compile-time options. More than one card may be used per external procedure.

General format:

```
* PROCESS  option [,option]...

  or

+ PROCESS option [,option]...
```

General rules:

1. The cards have to precede the PL/I source program. They must, however, follow the // EXEC PL/I statement.

2. The card has to start either with an asterisk or with a plus sign in column one, followed by one or more blanks. If the plus sign is used it is treated as an asterisk. The option list may not extend beyond column 71.

3. The options in the PROCESS statement override job-control options or any other options encountered in previous PROCESS statements.

The options that can appear in the operand field of a PROCESS card are:

1. Options supported by Job Control:

DECK	NOSYM
NODECK	ERRS
LIST	NOERRS
NOLIST	XREF
LISTX	NOXREF
NOLISTX	48C
SYM	60C

A description of the above options is given in Figure 8 in the section The Job Control Program.

2. Options not supported by Job Control:

a. OPT, NOOPT

OPT causes the optimization of compiled code.

NOOPT suppresses the OPT option.

The default is OPT.

Optimization implies the deletion of as much code as the compiler can diagnose as redundant.

If the option OPT is used, sequential assignment statements for the same variable (e.g., A=1; B=X; A=3; are optimized by deletion of the first assignment to A because there is no reference to A between the two assignments to A.

If the contents of 'A' were required between the two assignments (e.g., were used as control values in the event of an interrupt such as SIZE, CONVERSION, etc.) the assignment statement would have to be labeled because labeling a statement resets the internal optimization control.

b. STMT, NOSTMT

STMT causes statement numbers to be printed with object time diagnostics.

NOSTMT suppresses the STMT option.

The default is NOSTMT.

Note: In a program consisting of several external procedures, STMT must always have been specified for the first external procedure that is stored on SYSLNK if the object-time diagnostic messages for any of the external procedures of the program are to include the numbers of the source statements causing errors. If, for example, STMT is specified only for the second external procedure stored on SYSLNK, statement numbers are not printed for this procedure. In addition, STMT must also have been specified for the first external procedure.

c. LISTO, NOLISTO

LISTO causes the statement numbers to be listed and the offset of the first byte used after these statements to be printed.

NOLISTO suppresses the LISTO option.

The default is NOLISTO.

Note: LISTO overrides LISTX, i.e., if LISTO and LISTX are specified, the LISTX option is ignored.

COMPILATION UNDER DOS/TOS CONTROL

If a single PL/I source module is to be compiled under DOS/TOS control, the card sequence should be as follows:

```
// JOB     job-name
// OPTION  DECK,LIST,NOSYM,60C see note 1
// EXEC    PL/I
   .....
   .....   PL/I source module
   .....
/*
/&                              see note 2
```

Note 1: This statement causes the PL/I compiler to punch an object module on SYSPCH and to list the source program on SYSLST. The listing of source module symbols is suppressed. The source program is written in the 60-character set. LOG, DUMP, LISTX, and ERRS are assumed to have been established as standard options at system generation time.

Note 2: Another /& card must be read from SYSIPT if SYSRDR and SYSIPT do not refer to the same input device.

```
Deck on SYSRDR
// JOB     MYJOB
// OPTION  DECK,48C
// ASSGN   SYSIPT,X'271',X'50'
*  PLEASE  MOUNT REEL 4711 ON UNIT 271
// PAUSE   PROCEED
// EXEC    PL/I
// EXEC    PL/I
// EXEC    PL/I
/&

Records on SYSIPT
   First PL/I source module
/*
   Second PL/I source module
/*
   Third PL/I source module
/*
/&
```

Figure 9. Coding for a Job Consisting of three PL/I Compilations

ASSGN statements to change the assignment of logical device addresses for this job may be placed anywhere between the JOB and the EXEC statement. Assignments for SYSLNK must not be changed after OPTION LINK has been specified.

Figure 9 shows the coding for a job consisting of three PL/I compilations. SYSRDR and SYSIPT are assumed to refer to different input devices. SYSIPT is assumed to be a 7-track tape drive.

Since a job step comprises only one single compilation, an EXEC statement as well as a /* statement is required for the compilation of each source module (external procedure).

Main storage requirements for compilation: In the background partition, without system files on disk:

10K plus supervisor area required for the compiler version used

In the background partition, with system files on disk:

12K plus supervisor area required for the compiler version used

In the foreground partition, without system files on disk:

10K plus foreground save area required for the compiler version used

In the foreground partition, with system files on disk:

12K plus foreground save area required for the compiler version used

The Linkage Editor Program

The Linkage Editor program relocates the object modules produced by the PL/I compiler into an absolute object program. Modules retrieved from the relocatable library may be incorporated into the object program during the Linkage Editor run. Programs written in Assembler language and assembled by means of the DOS/TOS Assembler may also be incorporated. For details on the communication with programs written in Assembler language refer to the section Linkage Conventions. The object program produced by the Linkage Editor may either be executed by using the EXEC statement with a blank operand or be incorporated into the core-image library.

If a Linkage Editor run is desired, the first Linkage Editor control statement and the first EXEC statement must be preceded by an OPTION statement with either the LINK or the CATAL option.

In a TOS or DOS non-multiprogramming system, the Linkage Editor can run in the background partition only. In a DOS multiprogramming system which also supports the private core-image library option, the Linkage Editor can run in either foreground partition (as well as the background) provided a minimum of 10K of main storage and a private core-image library is assigned.

LINKAGE EDITOR CONTROL STATEMENTS

The execution of the Linkage Editor program is initiated by Linkage Editor control statements read from SYSRDR. The general format of Linkage Editor control statements is similar to that of the Job Control statements, except that Linkage Editor control statements have a blank in column 1 instead of // in columns 1 and 2.

The Linkage Editor program uses the following four control statements:

- the PHASE statement,
- the INCLUDE statement,
- the ENTRY statement, and
- the ACTION statement.

The exact format of these statements is given in those parts of this section where their application is described.

The ACTION Statement

This is an optional statement for directing the Linkage Editor. If ACTION statements are issued to the Linkage Editor, they must precede all other input to the Linkage Editor on SYSLNK. This can be ensured by placing the ACTION statement(s) immediately after the OPTION statement with the operand LINK or CATAL. The format of the ACTION statement is:

```
ACTION   operand
```

The following operands are of interest to the PL/I user:

BG
F1
F2 — The program is link-edited to execute in the specified partition. The start address of the appropriate partition is assumed to be the end of the Supervisor (for background) or the address of the specified foreground partition allocated at link edit time. Only one of these operands may be specified for one link-edit step. Use of these operands allows the program to be link-edited to execute in a partition other than the one in which the link-edit function is taking place.

In the absence of these operands the program is link-edited to execute in the partition in which link-editing is taking place. (These operands are not available in TOS).

NOMAP — Suppresses listing of the Linkage Editor storage map on SYSLST. Diagnostics are written on SYSLOG.

CANCEL — The job is canceled if any error is detected during link-editing.

More than one ACTION statement may be issued for one link-editing step.

The PHASE Statement

If the program consists of more than one phase or if the program is to be cataloged, each phase to be link-edited must be preceded by a PHASE statement of the following format:

```
PHASE phase-name,origin
```

Phase-name is a symbol consisting of 1 to 8 characters, the first of which must be alphabetic but should not be a $ sign. In case of multi-phase programs, the phase-name must be longer than four characters and the first four characters must be identical for all phase names of that program. Different programs must differ in the first four characters of their phase name(s) in order to avoid incorrect storage allocation. (See the section Processing of Overlays by the Linkage Editor.)

Origin indicates to the Linkage Editor the begin address of this specific phase. An asterisk may be used as an origin specification to indicate that this phase is to follow either

> the previous phase or the Supervisor at the next double-word boundary (for background programs) or the start of the partition (for foreground programs).

This simple format of the PHASE statement covers all normal applications in the background partition. For the format of the phase origin in overlay structures refer to the section Overlay.

Three methods are available for link-editing foreground programs:

1. Using the statement ACTION Fn. In this case, the same set of PHASE statements may be used as for background programs.

2. Using the operand format F+address of the PHASE statement for the origin of the first (or only) phase.
   address is the absolute address of the foreground area in which the link-edited program is to be executed. It may be specified by a hexadecimal number of four to six digits (X'hhhhhh') or by a decimal number of five to eight digits (dddddddd) or in the form nnnnK, where nnnn is two to four digits and K equals 1024. For example, an origin may be specified as F+X'8000' or F+32768 or F+32K.

3. Executing the link-edit function in the desired foreground partition. In this case, the same set of PHASE statements may be used as for background programs.

For each method, a foreground save area is created at the specified address. The (first) phase starts at the first double-word boundary following this save area. The space allocated to a foreground program by the Linkage Editor plus sufficient space following the end of the program for dynamic allocation of PL/I automatic storage must be allocated at execution time to the appropriate foreground partition.

Programs compiled by the PL/I compiler and PL/I library routines are not self-relocating.

Note: The autolink feature of the Linkage Editor is required to include routines from the relocatable library that are to be linked with the object modules compiled by the PL/I compiler. Therefore, the option NOAUTO of the PHASE or ACTION statement must never be used.

INCLUDING OBJECT MODULES INTO THE OBJECT PROGRAM

The appropriate object modules can be incorporated into the object program by:

- compilation,
- including object card decks,
- including object modules from the relocatable library, or
- using the autolink feature.

Compilation

To have the source module compiled and the output written on SYSLNK, the card sequence must be as follows:

```
// EXEC PL/I
   ....
   ....  PL/I source module
   ....
/*
```

If SYSRDR and SYSIPT refer to different input devices, the PL/I source module and the /* card must be read from SYSIPT.

Processing by the Linkage Editor and execution is suppressed in case severe or disastrous programming errors are detected during compilation.

Source modules written in Assembler language may be added in the same manner by using the statement // EXEC ASSEMBLY for calling the Assembler. For details on the communication with programs written in Assembler language refer to the section Linkage Conventions.

Including Object Card Decks

To include one or more object card decks into the object program, the required control cards as well as the sequence in which they must be read from SYSIPT or SYSRDR, respectively, are shown in Figure 10.

Note: The INCLUDE card, when used for this application, must have the following format:

INCLUDE preceded and followed by blanks only

Cards	Read from
INCLUDE	SYSRDR
...	
... one or more object modules	SYSIPT
...	
...	
/*	SYSIPT

Figure 10. Including Object Card Decks

Including Object Modules from the Relocatable Library

An INCLUDE statement must be read from SYSRDR for each module to be incorporated into the object program from the relocatable library. When used for this application, the INCLUDE statement must have the format:

INCLUDE module-name

Using the Autolink Feature

If some references to external names remain unresolved after all modules have been read in from SYSLNK, SYSIPT, and/or from the relocatable library, the autolink feature of the Linkage Editor searches the relocatable library for module names identical to the unresolved names and includes the corresponding modules into the object program.

Private Relocatable Library under DOS

Cataloging and including of relocatable modules may be performed by means of a private relocatable library. For DOS, the private relocatable library resides on an extra 1316 disk pack. The 2311 disk drive on which this pack is mounted has the logical device address SYSRLB.

For including modules, the DOS Linkage Editor first searches the pack assigned to SYSRLB and, if the requested module is not found there or if SYSRLB is not assigned, it searches the relocatable library on the system residence pack.

If SYSRLB is assigned, relocatable modules are cataloged into the private relocatable library. Otherwise, they are cataloged into the system residence pack.

For creating private relocatable libraries refer to the SRL publication IBM System/360, Disk Operating System, System Control and System Service Programs, Order No. GC24-5036.

For private relocatable libraries under TOS see Special Considerations on TOS.

The ENTRY Statement

The card input to the Linkage Editor may be delimited by an ENTRY statement of the following format:

```
ENTRY [name]
```

Name is the external name of the entry point used. The entry point must be a primary or secondary entry of the external procedure that has the option MAIN. If the primary entry point of the MAIN procedure is used, the name may be omitted.

If no ENTRY statement is issued, ENTRY with a blank operand is assumed.

Note: If modules written in Assembler language are to be incorporated into the object program, the Assembler END statement should have a blank operand field in order to avoid confusion of entry points.

Errors Due to Multiple Secondary Entry Points

For each file specified in the source program, the compiler generates a separate DTF table which includes the names of the I/O modules to be called.

Sometimes different I/O modules have identical secondary entry names. For example, if a program uses ISAM files and ADDBUFF is specified for one of these files and INDEXAREA for another, then the secondary entry point IJHAARZZ occurs in modules IJHAARCZ and IJHAARZP, which are provided for these two files. In a case like this, the linkage editor error message 2143I (Content of statement in error) is generated during link-editing. Inspite of this error message, the program may execute correctly.

There is no way of determining beforehand whether or not a program with linkage error message 2143I will execute correctly. To make sure that the correct module is linked to the program, the following should be done:

Message 2143I gives the multiple entry name in print positions 64 through 71 and the name of the module that was linked to the program in print positions 39 through 46. The linkage editor output listing repeats, in the LABEL column, the multiple secondary entry name and also lists the name of the other module in which this entry name occurs.

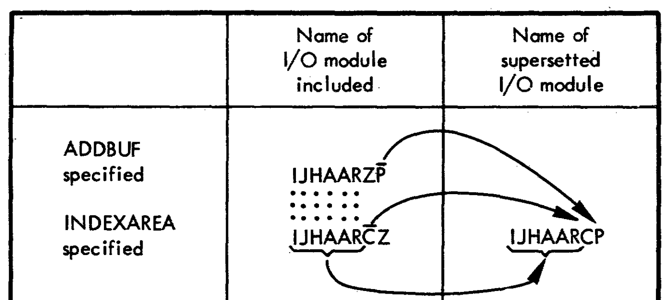
In a new linkage-editor run, now, a supersetted I/O module must be specified for inclusion in the program. This supersetted module will contain the individual modules whose inclusion caused the error message 2143I to be generated. The name of the supersetted module is found as follows:

Assume that two ISAM files have been specified, one with the ENVIRONMENT attribute ADDBUFF and another with the ENVIRONMENT attribute INDEXAREA. In this example, the linkage editor generates message 2143I with the multiple secondary name IJHAARZZ printed in positions 64 through 71 and the name of the I/O module in which this secondary entry name occurs (IJHAARZP, which is provided for an ISAM file with the ADDBUFF option) printed in positions 39 through 46.

In the linkage editor output listing, the secondary entry name IJHAARZZ appears in the column LABEL under the associated CSECT name (IJHAARCZ, the name of the module which is provided for an ISAM file with the INDEXAREA option).

Figure 11 below shows how to build the name of a supersetted I/O module; this name should then be specified in an INCLUDE card and inserted before // EXEC LNKEDT statement:

```
   INCLUDE IJHAARCP
// EXEC LNKEDT
```



Where the characters of the module names are identical, these characters are used in the same positions of the name for the supersetted module.

Where the characters of the module names differ, the lowest character is used in the corresponding position of the name for the supersetted module.

Figure 11. Building the Name of a Supersetted I/O Module

Errors Due to Multiple CSECT Names

Different user-written or PL/I modules to be linked to a program by the linkage editor may have identical CSECT names, as shown in Figure 12.

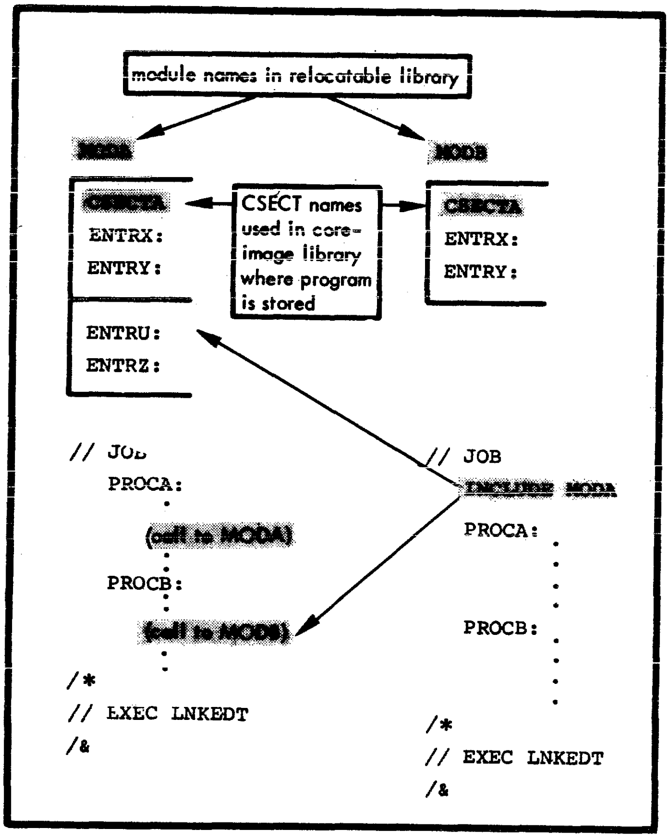


Figure 12. Multiple CSECT Names

The modules shown in Figure 12 may have completely different code, or the code of the two modules may be identical to a certain point, with one of the modules having one or more additional entry points (as in Figure 12).

A linkage-edit time, now, the linkage editor fetches the required module from the relocatable area, where the modules are stored under different names, and checks whether the corresponding CSECT name is already on SYSLNK. If the CSECT name is already contained in SYSLNK the second module with the identical CSECT name (but the different module name) is not linked. If a request to any additional entry (ENTRV or ENTRZ) exists, an error message is issued, but link-editing is continued. At execution time, an error will result.

To avoid errors due to multiple CSECT names, the modules required must be included in the appropriate external procedure by means of an INCLUDE statement as shown in Figure 12.

SAMPLE COMPILATION

The example shown in Figure 13 illustrates a combination of all three possibilities to build an object program. Four modules plus the appropriate library subroutines are to be combined into an object program, which is to be executed upon completion of the compilation. The example is based on the following assumptions:

1. One module (A) is a PL/I source module.

2. Two modules (P1, P2) have been previously compiled and punched.

3. One module (R) is contained in the relocatable library.

4. A listing of the source program and the symbol table is required for module A.

5. A is the entry point to be used.

6. The job is being executed in the background partition.

Note: The numbers at the left in Figure 13 are for reference purposes only; they are not part of the coding.

```
  1  |// JOB NO1234
  2  |// OPTION LINK,SYM,LIST
  3  |   PHASE EXAMPLE,*
  4  |// EXEC PL/I
     |   A: PROCEDURE OPTIONS (MAIN);
  5  |      .
     |      .
     |      END /*A*/;
     |/*
  6  |   INCLUDE
     |   .
     |   .  deck P1
     |   .
     |   .
     |   .  deck P2
     |   .
     |/*
  7  |   INCLUDE R
  8  |   ENTRY
  9  |// EXEC LNKEDT
 10  |// EXEC
     |   .
     |   .  data
     |   .
 11  |/*
 12  |/&
```

Figure 13. Sample Compilation

Explanation

1 Furnishes the Communication Region of the Supervisor with the name of the job.

2 Specifies the compiler options SYM and LIST and enables the PL/I compiler and Job Control to write or copy the output on SYSLNK for later processing by the Linkage Editor.

3 The PHASE statement precedes all modules to be processed by the Linkage Editor. The asterisk indicates that the program is to be loaded immediately following the Supervisor.

4 Calls the PL/I compiler.

5 PL/I source program. A (the name of the MAIN procedure) is the primary entry point.

6 Causes the subsequent modules P1 and P2 to be copied onto SYSLNK.

7 This statement is copied onto SYSLNK. When encountered by the Linkage Editor, the module R is fetched from the relocatable library and incorporated.

8 Delimits the input to the Linkage Editor. The blank operand causes the primary entry point A to be entered by Job Control at execution time.

9 Calls the Linkage Editor to produce the object program. The names of all modules called by A, P1, P2, and R must be names of modules contained in the relocatable library. These modules are automatically incorporated by the autolink feature of the Linkage Editor.

10 Causes Job Control to fetch the executable object program and transfers control to A for execution.

11 The end-of-data-file statement delimits the input data. If the file name is explicitly declared, this statement may be tested by means of an ON ENDFILE statement.

12 End-of-job statement. In case of an abnormal termination of the job, Job Control skips all input up to this statement.

Assumed that all input to be read from SYSIPT has been loaded onto a 7-track tape reel and that SYSIPT is assigned to the tape drive whose physical address is 281, the input from SYSRDR and SYSIPT for the above example is as shown in Figure 14.

	Cards read from SYSRDR
	`// JOB NO1234`
13	`// ASSGN SYSIPT,X'281',X'90'`
	`// OPTION LINK,SYM,LIST`
	`PHASE EXAMPLE,*`
	`// EXEC PL/I`
	`INCLUDE`
	`INCLUDE R`
	`ENTRY`
	`// EXEC LNKEDT`
	`// EXEC`
14	`/&`

Figure 14. Control Cards and Input Units for Deck Shown in Figure 13 (Part 1 of 2)

Explanation

13 SYSIPT is assigned to a 7-track tape drive. (The assignment differs from the installation standard.)

14 /& must appear on both SYSRDR and SYSIPT.

	Cards read from SYSIPT
	`A: PROCEDURE OPTIONS (MAIN);`
	`...`
	`...`
	`END /*A*/;`
	`/*`
	`... deck P1`
	`... deck P2`
	`/*`
	`... data`
	`/*`
14	`/&`

Figure 14. Control Cards and Input Units for Deck Shown in Figure 13 (Part 2 of 2)

To execute the same job in a foreground partition with a private core-image library on a disk unit whose physical address is 191, the statement

ASSGN SYSCLB,X'191'

must follow the JOB card. In this case, the program is link-edited to be loaded at the start of the foreground partition.

Cataloging

Cataloging of frequently used program phases or object modules into one of the DOS/TOS libraries greatly reduces the time required for card reading and/or Linkage Editor processing. Object modules may be cataloged into the relocatable library. Executable programs already processed by the Linkage Editor may be cataloged into the system or, if assigned, a private core-image library.

The name of a phase or module must be unique for each library. If phases or modules are cataloged, any module or phase already contained in the respective library and having the same name is automatically deleted. This necessitates some naming conventions for each installation in order to prevent a user from deleting programs that are either part of the system or cataloged into the library by other programmers using the same installation. Core-image library phase names starting with $ as well as relocatable library module names starting with IJ are names of system programs. For this reason, the user should be very careful when cataloging phases or modules the names of which start with the above characters.

The Library routine that handles cataloging and deleting is called by the Job Control statement // EXEC MAINT.

Cataloging Into The Core-Image Library

If a program is to be cataloged into the core-image library, the statement // OPTION with the CATAL option must be given prior to Linkage Editor processing, i.e., this statement must precede the first PHASE card of the program to be cataloged in case of compile-and-link runs. Upon successful completion of Linkage Editor processing the program is then automatically cataloged when an // EXEC LNKEDT and /& card is read. (Note that no // EXEC statement without name must precede the // EXEC LNKEDT or /& statement in this job.) No further catalog control statements are required.

If a private core-image library is assigned, the program is cataloged into that library rather than into the system core-image library.

Note: An error may occur if a phase exists in the core-image library whose name starts with the same four characters as the program to be cataloged (see the publication IBM System/360 Disk Operating System, System Control and System Service Programs, Order No. GC24-5036).

Programs or phases that are no longer required in the core-image library may be deleted by using the DELETC statement, the two possible formats of which are as follows:

```
DELETC phase1[,phase2]...
DELETC prg1.ALL[,prg2.ALL]...
```

The first format is used to delete single phases. The operands phase1, phase2, etc., each specify the name of one phase to be deleted. The second format is used to delete entire programs. Since the first four characters of all phase names of any program are identical, the entire program is deleted if these four characters are specified. prg1, prg2, etc., must therefore be exactly four characters long.

Cataloging Into The Relocatable Library

Each card deck to be cataloged into the relocatable library must be preceded by the control statement

```
CATALR module-name[,v.m]
```

The module specified by the operand module-name is then incorporated into the relocatable library. Cataloging stops when the END card of the module has been cataloged. The module may be preceded but not followed by Linkage Editor control statements.

v.m specifies the change level at which the module is to be cataloged. v may be any decimal number from 0 through 127. m may be any decimal number from 0 through 255. A change level of 0.0 is assumed if this operand is omitted.

Compilation of a PL/I source module may result in two object modules. (The first one will be referred to as file module and the second one as procedure module in this section.) The file module is produced for all of the file declarations (except file name parameters) contained in the source module. The procedure module is produced for the source module itself. Note that each individual object module requires a separate CATALR statement for cataloging. The file module may be cataloged under any of the file names.

The DOS PL/I compiler facilitates cataloging into the relocatable library by optionally producing control statements on SYSPCH. If bit 0 of the UPSI byte (see the section The UPSI Statement) is on during compilation, the following output is generated on SYSPCH depending on whether or not a file module is generated with the external procedure:

```
with file module          without file module

 CATALR  Fname              CATALR  name
   file module                procedure module
 CATALR  name
 INCLUDE Fname
   procedure module
```

name is the primary entry point of the external procedure. Fname means that the name of the external procedure, immediately preceded by the character F, is used as the name of the file module. The INCLUDE statement is generated to have the file module automatically included with the procedure module.

There is no automatic catalog feature for compile-and-catalog into the relocatable library. However, if a sufficient number of tape drives is available, it is recommended to assign SYSPCH to a magnetic tape drive and to reassign the same drive to SYSIPT for the catalog step, thus eliminating unnecessary card handling.

The following example shows what control statements are required for compile-and-catalog into the relocatable library:

```
  // JOB      COMPILE AND CATALOG
  *           INTO THE RELOCATABLE LIBRARY
  // OPTION   SYM,LISTX,DECK
1 // UPSI     1
2 // ASSGN    SYSPCH,X'182'
2 // MTC      REW,SYSPCH
  // EXEC     PL/I
     ...
     ...      PL/I source program
     ...
  /*
3 // MTC      WTM,SYSPCH
3 // MTC      REW,SYSPCH
3 // RESET    SYSPCH
4 // ASSGN    SYSIPT,X'182'
5 // EXEC     MAINT
  /&
```

Explanation

1. This statement causes the DOS PL/I compiler to generate control statements that precede the object module(s).

2. Assigns magnetic tape unit 182 to SYSPCH and positions the tape at the load point.

3. Closes and repositions SYSPCH. (Do not use the // CLOSE statement since this statement unloads the tape, thus causing unnecessary operator action).

4. The compiler output is now assigned to SYSIPT.

5. The Librarian is called. The CATALR statements cause cataloging into the relocatable library.

Note: The control statements are generated only on SYSPCH, not on SYSLNK. Thus, compile-and-catalog into the relocatable library does not preclude the LINK and CATAL options in the same job.

The DELETR statement may be used to delete either single modules or entire programs contained in the relocatable library. All modules whose names start with the same 3-character combination are considered to be part of the same program. The two possible formats of the control statement are

```
DELETR module-name1[,module-name2]...

DELETR prg1.ALL[,prg2.ALL]...
```

The operands prg1, prg2, etc., must consist of exactly 3 characters.

Library Maintenance Runs

Cataloging and deleting for all libraries can be done in one single job step. In the following example, the program LNCT is deleted from the core-image library and the modules BCDFIR and BCDSEC are cataloged in the same job step. BCDSEC is preceded by a PHASE statement that is to be cataloged with the module.

```
// JOB     CATALOG TWO DECKS,
*          SECOND WITH PHASE CARD
// EXEC    MAINT
   DELETC  LNCT.ALL
   CATALR  BCDFIR
   ...
   ...  deck BCDFIR
   ...
   CATALR  BCDSEC
   PHASE   BCDPR2,*
* THIS STATEMENT IS ALSO CATALOGED
   ...
   ...  deck BCDSEC
   ...
/*         END OF MAINT. DECK
/&
```

The input deck must be followed by an end-of-data-file statement if another job step within the same job follows the maintenance run. The Librarian control statements and input decks to be cataloged are read from SYSIPT. (In TOS, Librarian control statements are read from SYSRDR.)

Example for Cataloging a Foreground Program

Two methods are available for cataloging a foreground program:

1. If the program is compiled and link-edited in the background, the following job stream can be used:

```
   // JOB     CATALFG
   // OPTION  CATAL
1     ACTION  F2
2     PHASE   FGPXYZ,*
   // EXEC    PL/I
      •
      PL/I    source deck
      •
   /*
3  // ASSGN   SYSRLB,X'192'
   // EXEC    LNKEDT
   /&
```

The ACTION statement (1) causes the Linkage Editor to allocate storage for the program in the storage presently allocated to the foreground-two partition. The PHASE statement (2) gives the program the name FGPXYZ. The second operand (*) specifies that the program is to start n bytes behind the location assigned at link-edit time as the start address of the foreground-two partition (n is the length of a foreground save area required by the system). The program to be cataloged is compiled in the same job. The ASSGN statement (3) assigns SYSRLB so that the Linkage Editor can obtain modules to be included by the AUTOLINK feature from a private relocatable library.

2. If the program is compiled and link-edited in the foreground, two changes are made to the above job stream:

   a. The ACTION statement is removed because the program will be link-edited to execute in the foreground partition in which the link-edit function is taking place.

   b. The statement

      ASSGN SYSCLB,X'191'

      is added to assign a private core-image library. The program will be cataloged in the private core-image library.

Special Considerations on TOS

If TOS is used, phases in the core-image and modules in the relocatable library are not stored at random locations but in alphameric order. Therefore, all phases and/or modules to be cataloged must also be in alphameric order. Maintenance requests for the core-image and the relocatable library may be given in the same job step but must not be intermixed. Note that a maintenance run under TOS control causes copying of the full system onto a new volume that will be located on SYS002. SYS001 must be assigned to a tape drive for intermediate use in this case.

The TOS compiler does not generate CATALR statements. However, the user may prepare his own CATALR statements and put them into the job stream on SYSRDR following // EXEC MAINT. (In TOS, Librarian control statements are read from SYSRDR instead of from SYSIPT.) The file module should be given a name equal to one of the file names to avoid the use of an INCLUDE statement for including the file module.

Users needing a large number of relocatable modules should use a private relocatable library. Using a private relocatable library yields the following advantages:

1. Only the relocatable library is copied during updating.

2. The performance of INCLUDE and AUTOLINK is considerably faster during processing by the Linkage Editor.

During Linkage Editor processing and library maintenance, the private relocatable library resides on an additional magnetic tape unit assigned to SYSRLB. A private relocatable library is produced by preceding the first CATALR or DELETR statement by the special Librarian statement NEWVOL. (The tape reel on SYS002 to accommodate the newly created relocatable library must be initialized with a standard volume label.)

If a private relocatable library is to be used on TOS, it must contain all modules to be included from the relocatable library because SYSRLB and the relocatable library on the system's resident library are never searched both.

Data Files

Terminology

A file is a set of data stored on an external storage medium. Its purpose is either one or a combination of the following:

- To provide the program with the required input.
- To store intermediate results obtained during the execution of the program. This may be required because the storage capacity does not suffice to accommodate both the program and the data.
- To store the results obtained by the execution of the program (maybe for use as input either to the same program at a later execution or to another program).

A block is the physical unit of information transferred between internal storage and the external storage medium of the file.

A record is the unit of information which is logically transferred between the program and the file by a single PL/I READ, WRITE, or REWRITE statement. A block may contain more than one record (blocked records). In blocked record files, the records are buffered until a full block has been gathered and then physically transmitted to the file. In the case of input files, one block is read into a buffer, and each READ statement transfers (locates) one single record to the program.

A label is a special set of records that identifies a magnetic tape file or a direct access storage device (DASD) file. Labels are processed by the PL/I statements OPEN and CLOSE.

A key is the information required to locate a record within a DASD file declared with the attribute DIRECT.

File Organization Schemes

The organization of a file may be consecutive, regional, or indexed.

The term file organization is synonymous with an algorithm for identifying and locating blocks and records on the storage medium holding the file.

CONSECUTIVE FILES

The blocks contained in CONSECUTIVE files are identified by the sequence in which they are stored. This renders it impossible to access (or store) the blocks in any manner other than sequential. This, in turn, implies that the DIRECT attribute is not permitted for CONSECUTIVE files.

A PL/I file declared to be CONSECUTIVE may consist of a deck of punched cards, a listing on a printer, one or more reels of magnetic tape, or some space on one or more 1316 disk packs used with the 2311 disk drive. Other storage media for CONSECUTIVE files like the paper tape reader, the optical character reader, or teleprocessing lines (DOS only) may be addressed by using subroutines written in Assembler language that will process these files.

For an example showing the creation and updating of a sequential disk file, refer to Appendix L. Programming Examples, "Creating And Updating a Sequential Disk File".

A magnetic tape file may be contained on a single tape reel or on more than one reel (multi-reel file). The logical unit where the file is located must be declared in the MEDIUM option of the ENVIRONMENT attribute. When using a multi-reel file, more than one tape drive may be assigned to this logical unit by specifying the ALT option in the ASSGN statement to overlap processing and mounting of tape reels. Only labeled files should be used for multi-reel files.

A magnetic tape may also contain more than one file. To position the file correctly an MTC statement may be used to space the tape forward over as many tape marks as precede the file to be opened. (Refer to Multi-File Volumes and Backwards Files in the section File Labels.)

REGIONAL FILES

The regional file organization is possible only for DIRECT DASD files. REGIONAL files are processed using the DOS Direct Access method. Two different methods are used:

- REGIONAL(1) where records are addressed by their relative position in the file
- REGIONAL(3) where records are addressed (1) by the number of the track on which they reside, the track number being

relative to the first track of the file and (2) by means of a key associated with the record.

For further details refer to the section Disk Organization.

Restrictions. REGIONAL files must be declared with the attributes DIRECT and KEYED, which exclude the use of the STREAM, PRINT, SEQUENTIAL, and the buffering attributes. The KEYLENGTH option of the ENVIRONMENT attribute is not permitted for REGIONAL(1) files but must be specified for REGIONAL (3) files. REGIONAL files permit only fixed unblocked records. The V, U, BUFFERS, LEAVE, and NOLABEL options of the ENVIRONMENT attribute are not permitted for REGIONAL files. The maximum relative record or track number is $2^{24}-1$. The EXTENT statements for REGIONAL files must be supplied in ascending symbolic-unit order. If there are multi-volume files, the symbolic units must be assigned in consecutive order.

Note on Compatibility. In OS PL/I, certain information contained in the key field or data field of REGIONAL files is used to flag a record of that file as deleted. Therefore, if the user plans to create files with DOS PL/I and read and/or update them with OS PL/I, he should avoid keys or data that would cause OS PL/I to consider the record as deleted. For detailed information refer to the pertinent section of the OS PL/I Programmer's Guide, Order No. GC28-6594.

REGIONAL(1) Files

The individual records in a REGIONAL(1) file are identified by their position relative to the position of the first record in the file, which has a relative record number of 0. A track is assumed to contain as many records as may fit, i.e., if some parts of the track are still empty, these "holes" are nevertheless counted as real records. The number of records per track is shown on the Programmer's Reference Chart, Form X20-1705. The key used to identify individual records and issued with the KEY or KEYFROM option of a READ, WRITE, or REWRITE statement is not written onto the DASD file but specified as a numeric field declared as PICTURE '(8)9'. Therefore, records to be read from a REGIONAL(1) file must not contain keys on the DASD. The value contained in the numeric field (key) is the relative number of the record in the file.

Creating a REGIONAL(1) File. The extents to be used by a PL/I REGIONAL(1) file must be preformatted by the DOS Clear Disk utility program. (For details refer to the SRL publication IBM System/360, Disk and Tape Operating Systems, Utility Program Specifications, Order No. GC24-3465.) This utility program creates dummy records that contain a string filled with user-defined characters. The file can then be actually created by specifying the OUTPUT attribute. Figure 15 shows a sample card deck used for preformatting a REGIONAL(1) file.

The DLBL and EXTENT statements are described in the section File Labels. Note

```
// JOB       MYNAME
// ASSGN     SYSØØ6,X'192'
// DLBL      UOUT,'REGIONAL 1 FILE',66/ØØ1
// EXTENT    SYSØØ6,666666,1,,1ØØ,93Ø
// EXEC      CLRDSK
// UCL B=(K=Ø,D=1ØØ),C'$',OY
// END
/&


// JOB       PREFRMT
// ASSGN     SYSØØ1,X'192'
// DLBL      UOUT,'REGIONAL 3 FILE',66/ØØ1
// EXTENT    SYSØØ1,333333,1,,116Ø,4Ø
// EXEC      CLRDSK
// UCL B=(K=16,D=8Ø),C'L',OY
// END
/&
```

Figure 15. Sample Card Decks for Preformatting REGIONAL Files

that the dummy file should have an expiration date that has already been passed because, otherwise, the unexpired-file condition would prevent the PL/I output file from being opened. Note further that the dummy file is sequential and that its name is UOUT, independent of the actual name of the file to be used in the PL/I program. The UCL statement and the END statement are utility control statements and have a fixed format, i.e., no additional blanks must be inserted. K=0 means that no key is associated with the records. D=100 means that the block length is 100. This value may be modified to the user's requirements and must be identical with the actual block length of the PL/I file. The dollar sign is the character to which the file is cleared. It may be replaced by any other character.

The KEY and KEYFROM Options for REGIONAL(1) Files. The expression in the KEY or KEYFROM option in READ, WRITE, or REWRITE statements must result in a character string of the form PICTURE '(8)9'. The value n represented by this expression is used to access the n-th record of the file relative to the beginning of the file. n must be less than 2^{24}.

For a programming example refer to Appendix L. Programming Examples, "Creating And Retrieving a REGIONAL(1) File".

REGIONAL(3) Files

Contrary to REGIONAL(1) files, records in REGIONAL(3) files are addressed by the number of the track on which they are located, the track being relative to the first track occupied by the file. The first track of a REGIONAL(3) file is counted as track 0. Each individual record contained in one track is associated with a key on the DASD in order to distinguish it from other records in that track. The length of this key is declared in the KEYLENGTH option of the ENVIRONMENT attribute. The key is a concatenation of two strings. The first (left) key string is a character string of a maxi-mum length of 247 characters and contains the information required to distinguish the records from the remaining records on the same track. The second (right) key string is a numeric field declared as PICTURE '(8)9' which contains the relative track number. The full key is written onto, or read from, the DASD file.

Like REGIONAL(1) files, REGIONAL(3) files require preformatting by the DOS Clear Disk Utility program. In addition to its clearing function, the utility program resets the record R0 (capacity record) to reflect that all tracks are empty. The file can then be actually created by specifying the OUTPUT attribute.

If an attempt is made to write more records onto a track than its capacity permits, the ON KEY condition is raised.

The KEY and KEYFROM Options for REGIONAL(3) Files. The expression in the KEY or KEYFROM option in READ, WRITE, or REWRITE statements must result in a character string whose length is the same as the length specified in the KEYLENGTH option of the ENVIRONMENT attribute. The last 8 characters must be in the form PICTURE '(8)9'. The numeric value n represented by the last 8 characters is used to access the n-th track of the file with a key identical to the character-string expression. n must be less than 2^{24}.

INDEXED FILES

This file organization is supported by the DOS PL/I compiler and by the PL/I DASD macro instructions. Both methods may be used to create, access, and update files with the indexed-sequential file organization. For details on the PL/I DASD macro instructions refer to the publication IBM System/360 Disk Operating System, PL/I DASD Macros, Order No. GC24-5059.

Indexed-Sequential Organization

An indexed-sequential file is one whose records are organized on the basis of a collating sequence determined by control fields (referred to as keys) that precede each block of data. The key for each block of data is from 1 to 255 bytes in length and contains the identifier of the last logical record in that block. Indexed-sequential files are contained in some space allocated on direct access volumes as prime areas and index areas.

The indexed-sequential file organization gives the programmer great flexibility in the operations he can perform on a file. Using this scheme of file organization, he has the ability to

- read or write (in a manner similar to that for sequential files) logical records whose keys are in ascending collating sequence.
- read or write random logical records. If a large portion of the file is being processed, reading records in this manner is somewhat slower than reading according to a collating sequence since a search for pointers in indexes is required for the retrieval of each record.

- add logical records with new keys. The system locates the proper position in the file for the new record and modifies the indexes accordingly.

Indexes. The ability to read and write records from anywhere in an indexed-sequential file is provided by indexes that are part of the file. There are always two types of indexes: a cylinder index for the entire file, and a track index for each cylinder. An entry in a cylinder or track index contains the identification of a specific cylinder or track and the highest key associated with that cylinder or track. The system locates a given record by its key after a search of a cylinder index and a track index within that cylinder.

A third type of index, the master index, is optionally available for very large files. A master index is generated only if the INDEXMULTIPLE option is specified in the declaration of the respective output file. The master index contains an entry for each track of the cylinder index. If a master index is present, the search in the cylinder index is limited to a search on one track. For usual applications, a master index is not recommended if the cylinder index consists of less than four tracks.

The track index always resides on the same extent as the prime data area. The cylinder and master index may reside on the same volume as the prime data area; however, they may also reside on a different volume of a different DASD type. The cylinder index must be immediately adjacent to the master index, if any, on the same volume. Master and cylinder index must be completely contained in one volume.

Insertion of Records. A new record added to an indexed-sequential file is placed into a location on a track which is determined by the value of its key field. If records were inserted in precise physical sequence, insertion would necessitate shifting all records of the file that have keys higher than that of the one inserted. However, an overflow area is available for each cylinder. Thus, a record can be inserted into its proper position with only those records on the track being shifted in which the insertion is made.

Overflow Area. In addition to the prime area, whose tracks initially receive the records of an indexed-sequential file, there is an overflow area for records forced off their original tracks by insertion of new records. When a record is to be inserted, the records already on the track that are to follow the new record are written back onto the track after the new record. The last record on the track is written onto an overflow track. Figure 16 illustrates this adjustment for addition of records to an indexed-sequential file whose keys are in a numerical ascending sequence.

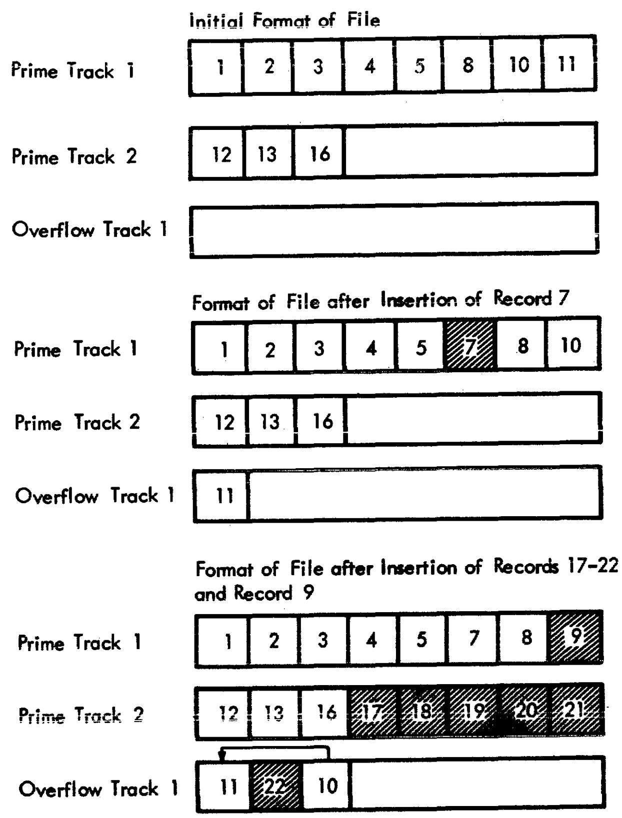


Figure 16. Addition of Records to a 1-Cylinder, 3-Track Indexed-Sequential File

When this file is created, its records are placed on the prime tracks in the storage area allocated to the file. If a record, e.g., record 7, is to be inserted into the file, the indexes indicate that record 7 belongs on primary track 1. Record 7 is then written immediately following record 5, and records 8 and 10 are retained on this track. Since record 11 no longer fits, it is written onto an overflow track and the proper track index is adjusted to show that the highest key on prime track 1 is 10 and that an overflow record exists. When records 17 to 22 are added to the end of the file, prime track 2 receives records 17 to 21 but record 22 does not fit and is written following record 11 on the overflow track. When record 9 is inserted, record 10 is shifted to the overflow track after record 22. Note that records 10 and 11 on the overflow track are chained together to show their logical sequence and to indicate that they belong to the same prime track.

Independent Overflow Area. An independent overflow area can be specified by an EXTENT statement (before the program is executed) to specify the area extent. If one or more of the (cylinder) overflow area(s) become full, additional overflow records are written on the independent overflow area. This area may be on the same volume as the data records or on another volume, but must be contained on one single volume. The number of overflow tracks reserved on each cylinder of the prime data area is determined by the OFLTRACKS option of the ENVIRONMENT attribute.

When using the PL/I DASD macro instructions, two tracks per cylinder are reserved as overflow area. The number of extents per file with PL/I DASD macro instructions is restricted to ten. Note that the cylinder index constitutes a separate extent.

The location of index areas, overflow areas, and the prime data areas on DASD devices are specified by means of DLBL and EXTENT statements. (Refer to the section File Labels.)

Record Format and Keys. With indexed files, all records must be of fixed length (blocked or unblocked). Since only one key is permitted per block on DASD devices, the access method for blocked records requires that the key be embedded in the data field of the record. The location of the key within the record is specified by the KEYLOC option of the ENVIRONMENT attribute. The key must be embedded in the data field if records are blocked; it may be embedded if the records are unblocked. If KEYLOC is specified to indicate embedding, the key is inserted automatically into the field during creation of the file or during addition of records to the file.

When the PL/I DASD macros are used, a record key is located within each record, and one extra key is associated with each block. This key is identical with the the highest (or only) record key in the block.

When processing INDEXED files, the KEY condition is raised in a number of cases. If the programmer wants to identify a specific situation at execution time, take action, and continue processing, it is suggested to include the following coding in the program:

```
DCL ERRBYTE BIT(8) EXTERNAL;
   CALL ERROUT  (ERROUT is an Assembler
                routine which returns in
                ERRBYTE the contents of the
                error byte)

   IF ERRBYTE = '00000100'B THEN ...
                (duplicate record, for
                example)
```

The contents of the error byte for indexed-sequential output files and the corresponding ON-conditions raised are as shown in Figure 17.

The contents of the error byte for all other indexed files and the corresponding ON-conditions raised are as shown in Figure 18.

The Assembler routine ERROUT is shown in Figure 19. (The address of the error byte is in the 10th word of the DTF-appendage.)

Bits	Cause	Explanation	ON condition raised	Followed by PL/I Message
0	DASD error	Any uncorrectable DASD error has occurred (except wrong length record).	0C	-
1	Wrong length record	A wrong length record has been detected during an I/O operation.	0E	-
2	Prime data area full	The next to the last track of the prime data area has been filled during the load or extension of the data file. The problem programmer should issue the ENDFL macro, then do a load extend on the file with new extents given.	0D	82
3	Cylinder Index area full	The Cylinder Index area is not large enough to contain all the entries needed to index each cylinder specified for the prime data area. This condition can occur during the execution of the SETFL. The user must extend the upper limit of the cylinder index by using a new extent card.	-	71
4	Master Index full	The Master Index area is not large enough to contain all entries needed to index each track of the Cylinder Index. This condition can occur during SETFL. The user must extend the upper limit, if he is creating the file, by using an extent card. Or, he must reorganize the data file and assign a larger area.	-	72
5	Duplicate record	The record being loaded is a duplicate of the previous record.	0D	83
6	Sequence check	The record being loaded is not in the sequential order required for loading.	0D	84
7	Prime data area overflow	There is not enough space in the prime data area to write an EOF record. This condition can occur during the execution of the ENDFL macro.	-	-

Figure 17. Contents of Error Byte For Indexed-Sequential Output Files

Bit	Cause	Explanation	ON-conditon raised	Followed by PL/I Message
0	DASD error	Any uncorrectable DASD error has occurred (except wrong length record).	0C -	- 70*
1	Wrong length record	A wrong length record has been detected during an I/O operation.	0E	-
2	End of file	The EOF condition has been encountered during execution of the sequential retrieval function.	0A	-
3	No record found	The record to be retrieved has not been found in the data file. This applies to Random (RANSEQ) and to SETL in SEQNTL (RANSEQ) when KEY is specified, or after GKEY.	0A**or 0D	80
4	Illegal ID specified	The ID specified to the SETL in SEQNTL (RANSEQ) is outside the prime data file limits.	-	-
5	Duplicate record	The record to be added to the file has a duplicate record key of another record in the file.	0D	83
6	Overflow area full	An overflow area in a cylinder if full, and no independent overflow area has been specified, or an independent overflow area is full, and the addition cannot be made. The user should assign an independent overflow area or extend the limit.	0D	81
7	Overflow	The record being processed in one of the retrieval functions (RANDOM/SEQNTL) is an overflow record.	-	-

* For indexed-sequential with READ KEY

** 0A is raised if the key is too high, otherwise 0D is raised.

Figure 18. Contents of Error Byte For All Other Indexed Files

```
ERROR     START  0
          EXTRN  ERRBYTE
          USING  *,9
          STM    14,12,12(13)
          LR     9,15
          LA     10,SAVEA
          ST     13,4(10)
          LR     13,10
          MVC    INDIC+2(1),INDIC+3
          L      1,FILE
          L      1,36(1)
          L      4,AERRBYTE
          USING  ERRBYTE,4
          MVC    ERRBYTE(1),0(1)
          L      13,4(13)
          LM     14,12,12(13)
          BR     14
          DS     0F
SAVEA     DC     X'03'
          DC     AL3(INDIC)
          DC     F'0'
INDIC     DC     3X'0'
          DC     X'F8'
FILE      DC     V(filename)
AERRBYTE  DC     A(ERRBYTE)
          END
```

Figure 19. Assembler Routine ERROUT

No RECORD condition will be raised for retrieving or updating files. The IOCS module gets the record length during OPEN time from the format-2 file label as it was written at creation time. No checking is made between this entry and the entry in the DTF table.

For blocked records, the RECORD condition will be raised when the first record of a block with one or more wrong-length records is read. With each execution of a READ statement the RECORD condition is raised until the last record of the block has been read.

The IOCS modules used by the PL/I (D) compiler are not reentrant. The PL/I library routine IJKTXCF deblocks such a wrong-length record. No other file using the same IOCS module may raise the RECORD condition before the wrong-length record has been deblocked to the end.

The KEY, KEYFROM, and KEYTO Options for INDEXED Files. The expression or variable in the KEY, KEYFROM, or KEYTO option of READ, WRITE, or REWRITE statements must result in or be a character string of the same length as the length specified in the KEYLENGTH option of the ENVIRONMENT attribute.

Note: In indexed-sequential files, retrieval, updating, and adding of records can be performed either sequentially or at random. However, indexed-sequential files can be created only sequentially.

Note on Compatibility. In OS PL/I, certain information contained in the key field or data field of INDEXED files is used to flag a record of that file as deleted. Therefore, if the user plans to create files with DOS PL/I and read and/or update them with OS PL/I, he should avoid keys or data that would cause OS PL/I to consider the record as deleted. For detailed information refer to the pertinent section of the OS PL/I Programmer's Guide, Form C28-6594.

Disk Organization

As an example of a DASD organization, this section describes the 1316 disk pack used with the 2311 Disk Storage Drive. The 2316 disk pack used with the 2314 or 2319 Direct Access Storage Facility is organized very similarly. However, the 2316 disk pack consists of 11 disks with 20 surfaces on which data is recorded with double density. For further details (also on the 2321 Data Cell Drive) refer to the publications IBM System/360 Component Descriptions, Order No. GA26-3599 (for the 2314) and IBM System/360 Component Descriptions, Order No. GA26-5988 (for the 2311 and 2321).

The 2311 DASD uses 1316 disk packs as recording medium. One disk pack consists of 6 disks. The top surface of the upper disk and the bottom surface of the lowest disk are not used, which leaves 10 surfaces for recording. Each disk surface contains 203 concentric tracks. Track 1, 2, 3, etc., on each surface is physically located below or above track 1, 2, 3, etc., of the other surfaces. Therefore, the corresponding tracks are referred to as 203 concentric cylinders. 200 cylinders are used for actual recording; the remaining 3 are reserved.

The 2311 is provided with one access arm equipped with 10 read/write heads. The heads are mounted vertically so that data contained in one cylinder can be accessed without any mechanical movement. This, however, renders it necessary to internally switch from surface to surface within a cylinder in case one track (of a consecutive file) is completely filled. When a cylinder is filled, reading or writing is resumed on the first track of the next cylinder. This technique minimizes the access-arm movement time.

Thus, a disk pack is thought of as consisting of 200 cylinders, each cylinder consisting of 10 tracks. A consecutive part of cylinders (or tracks) set aside for usage by a specific file is referred to as an extent. An extent is defined by an EXTENT statement (refer to the section File Labels). In case two or more files are to be accessed alternatingly, each individual

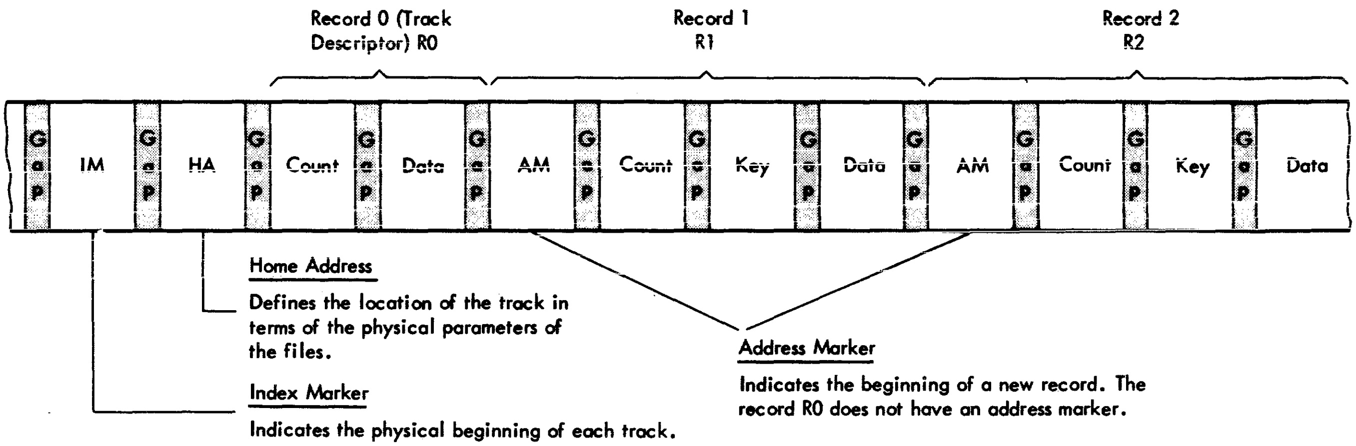


Figure 20. Contents of a Track

file may be assigned a part of consecutive tracks per cylinder instead of full cylinders. For instance, tracks 0 to 4 of cylinders 10 to 99 may be assigned to FILEA, while tracks 5 to 9 of the same set of cylinders may be assigned to FILEB. The latter technique is referred to as _split-cylinder_ technique.

The information contained on a track is recorded in physical records (see Figure 20). Each physical record consists of 2 or 3 fields.

The first field is a count field (C) identifying the record. The programmer is not concerned with this field. The second field is the key field (K). It has the length given in the KEYLENGTH option of the ENVIRONMENT attribute or in the KEYLEN operand of a PL/I DASD macro instruction and contains the key given in the KEY or KEYFROM option. CONSECUTIVE and REGIONAL(1) files have no key field. The last field is the data field (D) and contains the block to be read or written. The first record (Track Descriptor) of each track (R0) is not part of the information transferred by a PL/I program but contains some statistical information. The home address (HA) is of no interest to the PL/I programmer.

Record Types

These are five record types that can be handled by PL/I programs:

fixed unblocked
fixed blocked
variable unblocked
variable blocked
undefined

Fixed Unblocked Records

All records are of the same length. Each block contains exactly one record. The ENVIRONMENT option used is F(m).

Fixed Blocked Records

All records are of the same length. Each block contains a fixed number of records. (Only the last block of a file may contain less records.) The ENVIRONMENT option used is F(m,n).

Variable Blocked Records

The records are of variable length, each block containing a variable number of records. However, a maximum block length is specified. To enable the input/output control routines to determine the lengths of blocks and records, the blocks contain extra fields that are not part of the actual record. The first 4 bytes of each block contain a block control field. Each record in the block is also preceded by a 4-byte record control field. The ENVIRONMENT option used is V(m), where m is the maximum block size. m must include the number of bytes required by both the records and the control fields.

The D Compiler automatically supports variable-length blocked records if V(m) is specified, i.e., it always accommodates as many records in a block as will fit.

If at the end of a track there is not enough space for the whole block, the I/O routines write part of the block (but complete user-defined records) at the end of the track and shifts the remaining records onto the next track. Boundary problems may occur, however, if the rules for using the LOCATE statement with the SET option are not followed.

Variable Unblocked Records

This is a real subset of variable blocked. With variable unblocked records, the value of m in V(m) is 8 higher than the largest possible record in the file. Variable blocked and variable unblocked records may be intermixed.

Undefined Records

The records are of variable length. Each block contains one record. No control fields are used. The ENVIRONMENT option used is U(m).

Restrictions

For the restrictions regarding the block length refer to Appendix J under Blocksize Options.

A block has the meaning that the physical storage medium is advanced one block after the corresponding operation has been performed. In the case of punched cards, for instance, this implies that one card is read or punched. This, in turn, implies that the remainder of the card is ignored, and the next block starts with transmission of column 1 of the next card in case a block length of less than 80 bytes is specified for a card file.

Input/Output Processing

Access Methods

Since records in files declared with the CONSECUTIVE option are identified merely by the sequence in which they are created, the only possibility to read, write, or update records in such files is to sequentially process the file from its starting point. This procedure is referred to as the sequential access method, and files so accessed have the attribute SEQUENTIAL.

In other files, the records are identified by keys. In this case, each individual record can be accessed by use of the key regardless of the physical location of the record. This procedure is referred to as the direct access method, and a file so accessed has the attribute DIRECT.

Note: Indexed-sequential files may be read or updated either sequentially or direct.

Note: If two or more files are simultaneously open on the same physical non-DASD device or DASD extent, the order of access to the files is unpredictable. Read and punch feed of a 2540 Card Read-Punch count as two different devices. For example, a read and a punch file cannot be open at the same time using the same 1442 or 2520 Card Read-Punch. As another example, if there is a record file assigned to a printer and the standard system - STREAM - file uses the same printer, both files have their own buffers and print independently of each other, i.e., the printed lines will not necessarily appear in the same sequence as the WRITE and PUT statements are executed.

Buffering

A buffer is a part of storage used to accommodate data to be read or written. Buffers are used to allow transmission of data asynchronously to the program flow.

Files with the UNBUFFERED attribute allow no overlapping of input/output operations. In files declared with the BUFFERED attribute, execution of I/O operations is overlapped if the option BUFFERS(2) is specified in the ENVIRONMENT attribute. For files declared with the BUFFERED attribute, the buffers can be made available for use as work areas by using the READ statement with the SET option or the LOCATE statement, i.e., the based record variables are located directly in the buffers.

Tape files with the UNBUFFERED attribute must also have the NOLABEL attribute. Therefore, no multi-volume files or alternate-tape specifications are permitted.

If OUTPUT is specified in addition to UNBUFFERED and NOLABEL, tape labels are not checked and not overwritten.

Disk input and update files with the UNBUFFERED attribute are opened with the output OPEN routine. Therefore, the expiration date for such files must be lower than the current date.

Although buffering attributes are not permitted for DIRECT files, one buffer is assigned to REGIONAL and INDEXED DIRECT files. The minimum length of the buffer is the record length. The maximum length of the buffer is the record length + keylength + 8 for REGIONAL files and INDEXED DIRECT INPUT files. For INDEXED DIRECT UPDATE files, the maximum length of the buffer is the block length + keylength + 8 + 10 (for the sequence link field).

File Labels

A tape reel or disk pack may contain information that is required for a certain period of time. Therefore, each file (tape reel or disk extent) must be checked for its expiration date. In addition, a check must be performed to determine if the proper volume has been mounted for processing. These checks are performed by reading and comparing special records that are contained in the respective volume. These special records, which are referred to as labels, are processed whenever an OPEN or CLOSE statement is executed for a particular file.

The label information is furnished by means of special Job Control statements, which are described later in this section. There are two types of labels: volume labels and file labels.

Volume labels are used to identify the volume (tape reel or disk pack). During execution of the OPEN routine, the volume serial number is compared against the information supplied to the Supervisor. Volume labels can be created by means of IBM-supplied utility programs.

File labels describe the file to be processed by the program and indicate whether or not the file must be retained for a certain period of time. When an OPEN statement is encountered, the information contained in the file labels of input and update files is compared against the information supplied to the Supervisor. If a mismatch is found, a message to the operator is printed. When an OPEN statement is encountered for an output file, the expiration date in the file label is checked against the date stored in the communication region of the Supervisor. If the expiration date has been neither reached nor passed, a message to the operator is printed and the execution of the program is interrupted. In case the expiration date has been reached or passed, a new file label is created from the information supplied through the control cards. The old file label is overwritten by the new file label.

Labeled tape files have two types of labels: header labels and trailer labels. The header label precedes each file and defines it. The trailer label is written at the end of the file. It furnishes the information required to determine whether the end of the file has been reached or whether the file is continued on another volume. Tape files may also be unlabeled. This condition is specified by the option NOLABEL in the ENVIRONMENT attribute.

Disk files must be labeled. Disk file labels do not precede or follow the individual file. They are contained in a special region referred to as the VTOC (Volume Table Of Contents). Disk labels are updated either during execution of the CLOSE routine or when an end-of-extent is reached. Switching from volume to volume for multi-volume files is effected automatically without any programming effort.

Note: Punched-card and print files must not be labeled.

For detailed information and restrictions on label processing see the SRL publications describing the DOS/TOS data management concepts, the DOS/TOS Supervisor and I/O macro instructions, and the DOS system control and service programs.

Processing of File Labels by PL/I

PL/I does not provide for label processing of UNBUFFERED files. However, file labels are checked for expiration (also if INPUT is specified) and cleared. The volume label is maintained.

No provision has been made for label processing of the standard PL/I files SYSIN and SYSPRINT.

As far as label processing is concerned, UPDATE and INPUT files are handled in the same manner.

Job-Control Statements

A set of Job Control statements is required for each labeled file. This set of statements must be in a specific sequence and immediately precede the // EXEC statement for the job step in which the file is processed.

Each set of label information submitted within a job or job step is written in the appropriate temporary label information area. This information is not carried from job to job. Unless overwritten by a succeeding job step, any label information submitted at the beginning of a job can be used by a subsequent job step. For example, if a job consists of three job steps, label information submitted at the beginning of the first job step can be used by the second and third job steps of the job. However, label information submitted at the beginning of the second job step would destroy the label information written at the beginning of the first job step.

Note: DLBL and EXTENT Job Control statements for SYSIPT, SYSLST, or SYSPCH must precede the corresponding permanent ASSGN commands.

The sequence of Job Control statements for disk labels is as follows:

```
// DLBL
// EXTENT (one or more)
```

The Job Control statement for tape labels is as follows:

```
// TLBL
```

The syntax rules are the same as those for the other Job Control statements. Trailing commas not followed by an operand may be suppressed.

Note: The former disk and tape label Job Control statements DLAB, VOL, XTENT, and TPLAB may still be used. However, the old and new disk label statements must not be intermixed, i.e., XTENT is associated with DLAB and VOL, and EXTENT is associated with DLBL.

The DLBL Statement

The DLBL statement furnishes the disk file label information. The format of this statement is as follows:

```
// DLBL filename,['file-ID'],[date],[codes]
```

The meaning and format of the operands is described below:

filename is identical to the name of the PL/I file.

'file-ID' is the name of the file that is recorded on the disk drive as an identification of the file. It may comprise from 1 to 44 bytes of alphameric data. If less than 44 characters are used, the field is left-justified and padded on the right with blanks. If this field is omitted, the file name is used as file-ID.

date is a field of one to six numeric characters. Two formats are possible. The first format is yy/ddd, which indicates the expiration date of the file for output or the creation date for input. (The day of the year may have from one to three characters.) Optionally, a 1- to 4-digit retention period may be specified for output files. If this operand is omitted, a 7-day retention period is assumed for output files. For input files, no checking is performed if this operand is omitted or if a retention period is specified.

codes is a 2- or 3-character field indicating the type of file label as follows:

SD for Sequential Disk,

DA for REGIONAL files,

ISC for Indexed Sequential using Load Create, or

ISE for Indexed Sequential using Load Extension, Add, or Retrieve.

SD is assumed if this parameter is omitted.

For output files, the current date is used as the creation date.

The EXTENT Statement

The EXTENT statement defines an extent of a DASD file. One or more EXTENT statements must follow each DLBL statement. The EXTENT statement has the format

```
// EXTENT  [SYSxxx],[ssssss],[t],[nnn],
   [rrrrr],[mmmmm],[dd]
```

The meaning and format of the operands is described below.

SYSxxx (symbolic unit) is a 6-character field that indicates the symbolic unit of the volume to which this extent applies. If this operand is omitted, the symbolic unit of the preceding EXTENT statement is used.

For multi-volume REGIONAL files the symbolic unit numbers in the corresponding EXTENT statements must be in direct ascending sequence (e.g., SYS006, SYS007, SYS008).

ssssss (serial number) is a field of one to six characters that indicates the volume serial number of the volume to which this extent applies. If less than six characters are used, the field is right-justified and padded to the left with zeros. If this operand is omitted, the volume serial number of the preceding EXTENT statement is used. If no volume serial number was provided in that statement, the serial number will not be checked. (Files may be destroyed in this case due to mounting of the wrong volume.)

t (type) is a 1-digit field indicating the type of extent as follows:

1 - data area (no split cylinder)
2 - independent overflow area (for indexed sequential file)
4 - index area (for indexed sequential file)
8 - data area (split cylinder)

Type 1 is assumed if this operand is omitted.

nnn (sequence number) is a field of one to three characters that contains a decimal number from 0 to 255. The decimal number indicates the sequence number of the extent within a multi-extent file. For indexed files, the sequence number 0 is always associated with the master index. Thus, if a master index is specified, the sequence number for indexed files starts with 0; otherwise, i.e., if no master index is used, the first extent of an indexed file has the sequence number 1. The extent sequence number for all other types of files begins with 0. If this operand is omitted for the first extent of ISFMS files, the extent is not accepted. This operand is not required for SD and DA files.

rrrrr (relative track number) is a field of one to five characters that indicates the sequential number of the track (relative to zero) where the data extent is to begin. For instance, track 0 of cylinder 150 on a 2311 has the relative track number 1500. If this operand is omitted on an ISFMS file, the extent is not accepted. The operand is not required for SD or DA input files (the extents from the file labels are used in this case).

mmmmm (number of tracks) is a field of one to five characters that indicates the number of tracks to be alotted to the file. The operand may be omitted for SD or DA input files. For split cylinders, the number of tracks must be an even multiple of the number of tracks per cylinder specified for the file.

dd (split cylinder track) is a field of one or two digits that indicates the upper track number for the split cylinder in SD files.

Note: For INDEXED and REGIONAL files the LBLTYP card must also be present.

The TLBL Statement

The TLBL statement contains file label information for tape label checking and writing. Its format is as follows:

```
// TLBL filename,['file-ID'],[date],
        [file-serial-number],
        [volume-sequence-number],
        [file-sequence-number],
        [generation-number],
        [version-number]
```

The meaning and format of the operands is described below.

filename is a field of one to six characters identical to the name of the PL/I file.

'file-ID' is a field of one to 17 characters, contained within apostrophes, that indicates the name associated with the file on the volume. This operand may contain embedded blanks. If this operand is omitted for output files, filename is used instead. If this operand is omitted for input files, no labels are checked.

date is a field of one to six numeric characters. Two formats are possible. The first format is yy/ddd, which indicates the expiration date of the file or output or the creation date for input. (The day of the year may have from one to three characters.) Optionally, a 1- to 4-digit retention period may be specified for output files. If this operand is omitted, a 0-day retention period is assumed for output files. For input files, no checking is performed if this operand is omitted or if a retention period is specified.

file-serial-number is a field of one to six characters that indicates the volume serial number of the first (or only) reel of the file. If less than six characters are specified, the field is right-justified and padded with zeros. If this operand is omitted for output files, the volume serial number of the first (or only) reel of the file is used. If this operand is omitted on input, no checking is performed.

volume-sequence-number is a field of one to four digits. The sequence numbers of the volumes of a multi-volume file must be in ascending order. If this operand is omitted for output files, BCD 0001 is assumed. No checking is performed if this operand is omitted for input files.

file-sequence-number is a field of one to four digits. The sequence numbers of the files of a multi-file volume must be in ascending order. If this operand is omitted for output files, BCD 0001 is assumed. No checking is performed if this operand is omitted for input files.

generation-number is f field of one to four characters that modifies the file-ID. If this operand is omitted for output files, BCD 0001 is assumed. No checking is performed if this operand is omitted for input files.

version-number is a field of one or two characters that modifies the generation number. If this operand is omitted for output files, BCD 01 is assumed. No checking is performed if this operand is omitted for input files.

Notes:

1. For output files, the current date is used as the creation date.
2. As far as label processing is concerned, UPDATE files are handled the same as INPUT files.

Examples for Label Statements

The first two statements in Figure 21 show an example of DLBL and EXTENT statements used for a sequential 2311 disk input file. The statements identify the file declared as MASTIN in a PL/I program. Its external identification (stored in the VTOC) is MASTER-INVENTORY- FILE. No further entries in the DLBL statement are required for an input file.

The logical unit used for the file is SYS005 and the volume identification of the 1316 disk pack to be mounted on SYS005 is VOL172. No further EXTENT statement operands are required.

The following three statements in Figure 21 show an example the creation of an indexed sequential file by usage of the PL/I DASD macro LODIS. In addition to the previous example, the expiration date of the file (March 1, 1969) and the code ISC is specified in the DLBL statement. There are two EXTENT statements. The first one specifies the extent of the cylinder index, which is extent 1. It starts at track number 1000 (i.e., cylinder 100, track 0) on logical unit SYS004 and consists of 10 tracks. The data area, which is the second extent, resides on a different logical unit: SYS005. The extent number need not be specified in this case, but the delimiting comma must be written.

The last statement in Figure 21 gives an example of a TLBL statement. The file is assumed to be an input file. It is no multi-file volume and a version number is not used. Since it is an input file, no date entry has been specified.

Multi-File Volumes And Backwards Files

The handling of multi-file volumes on the 2311 requires no special preparation since all file labels are available when the file is opened. When using tape files, the tape must be positioned so that the label can be checked during execution of the OPEN routine. Positioning is not required for the first file on the tape because it is automatically positioned unless the LEAVE option has been specified. For correctly positioning the tape for the second, third, ..., nth file, the LEAVE option must be specified in the ENVIRONMENT attribute. This prevents the OPEN routine from rewinding the tape reel. A labeled tape file has the format shown in Figure 22.

```
// DLBL     MASTIN,'MASTER-INVENTORY-FILE'
// EXTENT   SYSØØ5,VOL172


// DLBL     CUSTIN,'CUSTOMER-FILE 37',69/Ø6Ø,ISC
// EXTENT   SYSØØ4,PNOØ18,4,1,1ØØØ,1Ø
// EXTENT   SYSØØ5,PNOØ27,1,2,1Ø,1ØØØ


// TLBL     PAYROL,'PAYROLL-FILE',,397,1,,19
```

Figure 21. Examples for Disk and Tape Label Statements

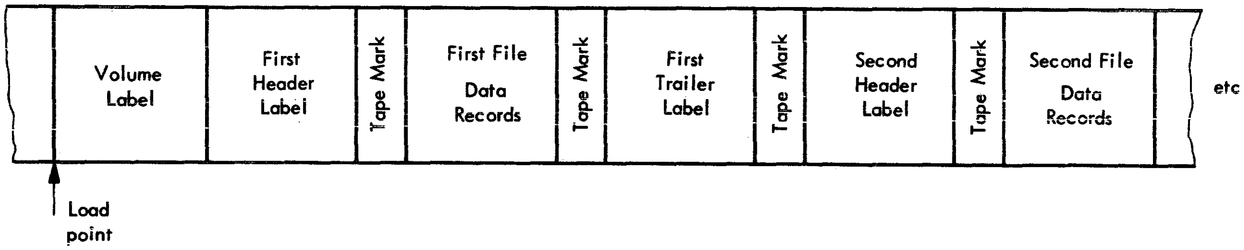


Figure 22. Format of Labeled Tape File

To position a tape reel that contains labeled files at the nth file, the tape reel must first be rewound and then spaced forward in such a manner that the first information read is the header label of the nth file. The Job Control statement used to control tape drive operations has the following form:

// MTC code,SYSxxx[,nn]

The operand code is one of the following function codes:

BSF backspace file
BSR backspace record
ERG erase gap
FSF forward space file
FSR forward space record
REW rewind
RUN rewind and unload
WTM write tape mark

Forward-space-file and backspace-file cause the read head to be positioned at the record following the next tape mark that is encountered.

The operand SYSxxx is the logical device address of the tape drive on which the pertinent tape reel is mounted.

The operand nn is a decimal number from 01 through 99 that specifies the number of times the specified function is to be performed. If this field and the comma preceding it are omitted, nn is assumed to be 01.

The following example shows the MTC statements required to position the tape reel on SYS006 at the header label of the third data file.

// MTC REW,SYS006
// MTC FSF,SYS006,03

In unlabeled tape volumes, the end of each file is indicated by a tape mark. A tape mark may or may not precede the first file. Unlabeled tape files written by PL/I programs have a tape mark preceding the first file unless NOTAPEMK is specified in the ENVIRONMENT attribute.

If a magnetic tape file has the BACKWARDS attribute, the read head must be positioned behind the trailer label of this file before the file is opened. In case a file has been written and closed just before it is re-opened to be read backwards, it is positioned correctly if the LEAVE option was specified for the written file. Unlabeled BACKWARDS files must start with a tape mark.

If an input file of a multi-file volume declared with the LEAVE option has been closed and the next file of this volume is to be opened (or the same file is to be opened in the reserve direction), the magnetic tape is positioned correctly only if the ENDFILE condition was raised prior to the closing of the file. In the case of STREAM input, additional (dummy) GET statements must be issued to synchronize the input stream with the ENDFILE condition. To prevent raising of the CONVERSION condition, the variables read by these dummy GET statements should be of the character type.

Link-Editing And Labeled Files

Before a program that uses and/or processes labeled files can be processed by the Linkage Editor, the Linkage Editor must be instructed to reserve a label area. This area must precede the area occupied by the program, except in the case of CONSECUTIVE disk files where no such area is required. To reserve the label area, a special Job Control statement must precede the statement // EXEC LNKEDT. The type of statement used depends on whether the program runs under control of the Disk Operating System or of the Tape Operating System.

Job Control Statements for DOS

The format of the Job Control statement for processing disk files with the REGIONAL or INDEXED option is as follows:

// LBLTYP NSD(nn)

The operand nn is the largest number of extents to be used by any single file.

Note that this number must be enclosed in parentheses.

Note that nn must specify the number of EXTENT cards and not the EXTENTNUMBER in the ENVIRONMENT attribute.

The format of this statement for the processing of labeled tape files is as follows:

// LBLTYP TAPE

Note: This statement is not required for processing of labeled tape files if REGIONAL files are used at the same time.

Job Control Statements for TOS

The format of the Job Control statement for the processing of labeled tape files is as follows:

// LBLTYP TAPE(nn)

The operand nn is the number of labeled tape files to be processed.

Figure 23 shows a source deck including Job Control statements for processing one REGIONAL file with two extents, and two tape files.

```
// JOB INVENTRY
// OPTION LINK,LIST,ERRS,60C
   PHASE UPDATE,*
// EXEC PL/I
   INVENTRY: PROCEDURE OPTIONS (MAIN);
             DECLARE MASTER FILE UPDATE RECORD ENVIRONMENT
                (REGIONAL(3)....)....,
                BACKUP FILE OUTPUT ENVIRONMENT (MEDIUM
                (SYS007,2400)....)....,
                EXEPT FILE OUTPUT ENVIRONMENT (MEDIUM
                (SYS008,2400)....)....,
             :
             END;
/*
   ENTRY
// LBLTYP NSD(02)
// EXEC   LNKEDT
// DLBL   MASTER,'MASTER INVENTORY FILE',,DA
// EXTENT SYS005,1427
// EXTENT SYS006,1431
// TLBL   BACKUP,'BACKUP INVENTORY',100,2711,,,10,8
// TLBL   EXEPT,'EXCEPTION INVENTRY',30,2614,,,10,0
// EXEC
   data
/*
/&
```

Figure 23. Sample Source Deck with Control Statements

CATALOGING OF LABEL INFORMATION

For DOS, the DLBL, EXTENT, and TLBL statements for sequential files may be cataloged as standard files so that the programmer is relieved from issuing the control cards with each execution of the program. For details refer to the SRL publication describing the DOS system control and system service programs.

Program-Label Communication

Figure 24 shows the communication between a PL/I source program, the object program, Job Control statements, and a 2311 disk unit with a 1316 disk pack.

The LIOCS (Logical Input/Output Control System) table produced by the PL/I compiler somewhere contains the file name as a character string. The communication between this table and the actual file extent(s) is established by storing the extent information in the table during execution of the OPEN statement.

The set of label statements (DLBL, EXTENT) to be used for opening the file is the one whose DLBL statement contains the same file name as stored in the character string of the LIOCS table. The logical device address is taken from the EXTENT card. The physical unit -- in this case a 2311 disk drive -- is then determined from the standard assignment or from the temporary assignments, respectively. The serial number field of the EXTENT statement is compared against the volume label of the 1316 disk pack to determine whether the right pack has been mounted.

The remaining action depends on the file type. For INPUT or UPDATE files, the VTOC on the disk pack is searched for a label matching the file-ID issued in the DLBL statement (MY DEAR FILE in Figure 24.) When a matching label is found, the remaining file information is checked against the label information in the VTOC, and the extent information is passed to the LIOCS table to allow proper addressing of the blocks to be transferred.

In case of OUTPUT files, all existing labels in the VTOC are checked against overlap with the file to be created. The file is opened only if there is no overlap with any unexpired file. The new label is then written into the VTOC.

In case of CONSECUTIVE multi-volume files, one volume will be opened at a time, i.e., the second volume is opened when the last extent of the first volume has been processed, etc. Opening of the second and following volumes is automatic. Thus, no explicit OPEN statement need be given. For all other files, all volumes will be opened at once. Therefore, all volumes to be processed must be mounted at the same time in this case.

The handling of tape label information is similar.

Assignment of System Files to Disk

In systems with at least 24K positions of main storage, the system logical units SYSIPT, SYSLST and/or SYSPCH may be assigned to an extent of 2311, 2314, or 2319 disk storage.

It should be noted that the assignment of system files to disk requires operator intervention. For a complete description (also of ASSGN and CLOSE commands) refer to the SRL publication System/360 Disk Operating, System Control and System Service Programs, Order No. GC24-5036.

The PL/I programmer should be aware of the fact that the PL/I standard files SYSIN and SYSPRINT are assigned to SYSIPT and SYSLST respectively. Since these files cannot be closed by the programmer and only one PL/I file can be opened for one System logical unit on Disk at any one time, the use of GET or PUT statements without the FILE option should be avoided if there are user-declared files for SYSIPT and SYSLST. In order to avoid implied usage of SYSLST for comments as a result of error conditions, it is recommended to use the ONSYSLOG option in the OPTIONS attribute of the MAIN procedure.

The assignment of system logical units to disk storage drives must be permanent. The operator ASSGN command must be used instead of the programmer statement (// ASSGN). Temporary assignments (via the // ASSGN statement) to other device types are permitted.

Note: The system generation parameter SYSFIL is required to allow assignment of system logical units to a disk drive.

System input and output files are assigned to disk by providing a set of DLBL and EXTENT statements and then submitting a permanent ASSGN Command. The set of DLBL and EXTENT statements preceding the ASSGN command may contain only one EXTENT statement.

The filename in the DLBL statement (which will be associated with the SYSxxx entry from the accompanying EXTENT statement) must be one of the following:

IJSYSIN for SYSRDR, SYSIPT, or the combined SYSRDR/SYSIPT file SYSIN

IJSYSPH for SYSPCH

IJSYSLS for SYSLST

In the DLBL statement, the codes operand must specify SD (or blank, which means SD) to indicate sequential DASD file type.

In the EXTENT statement, type may be 1 (data area, no split cylinder) or 8 (data area, split cylinder). There is no unique requirement for the remaining operands of the EXTENT statement.

The ASSGN command must be one of the following:

1. ASSGN SYSIN,X'cuu' (for a combined SYSRDR/SYSIPT file).
2. ASSGN SYSRDR,X'cuu' (for SYSRDR only).
3. ASSGN SYSIPT,X'cuu' (for SYSIPT only).
4. ASSGN SYSPCH,X'cuu' (for SYSPCH).
5. ASSGN SYSLST,X'cuu' (for SYSLST).

Note that all must be permanent assignments.

System logical units assigned to disk must be closed by the operator. The operator CLOSE command must be used to specify a system input or output file which has been previously assigned to a 2311, 2314, or 2319. The optional second parameter (X'cuu') of the CLOSE command may be used (instead of an ASSGN command) to assign the system logical unit to a physical device. The system will notify the operator that a CLOSE is required when the limit of the file has been exhausted. If a program attempts to read or write beyond the limits of the file, the program will be terminated and the file must be closed.

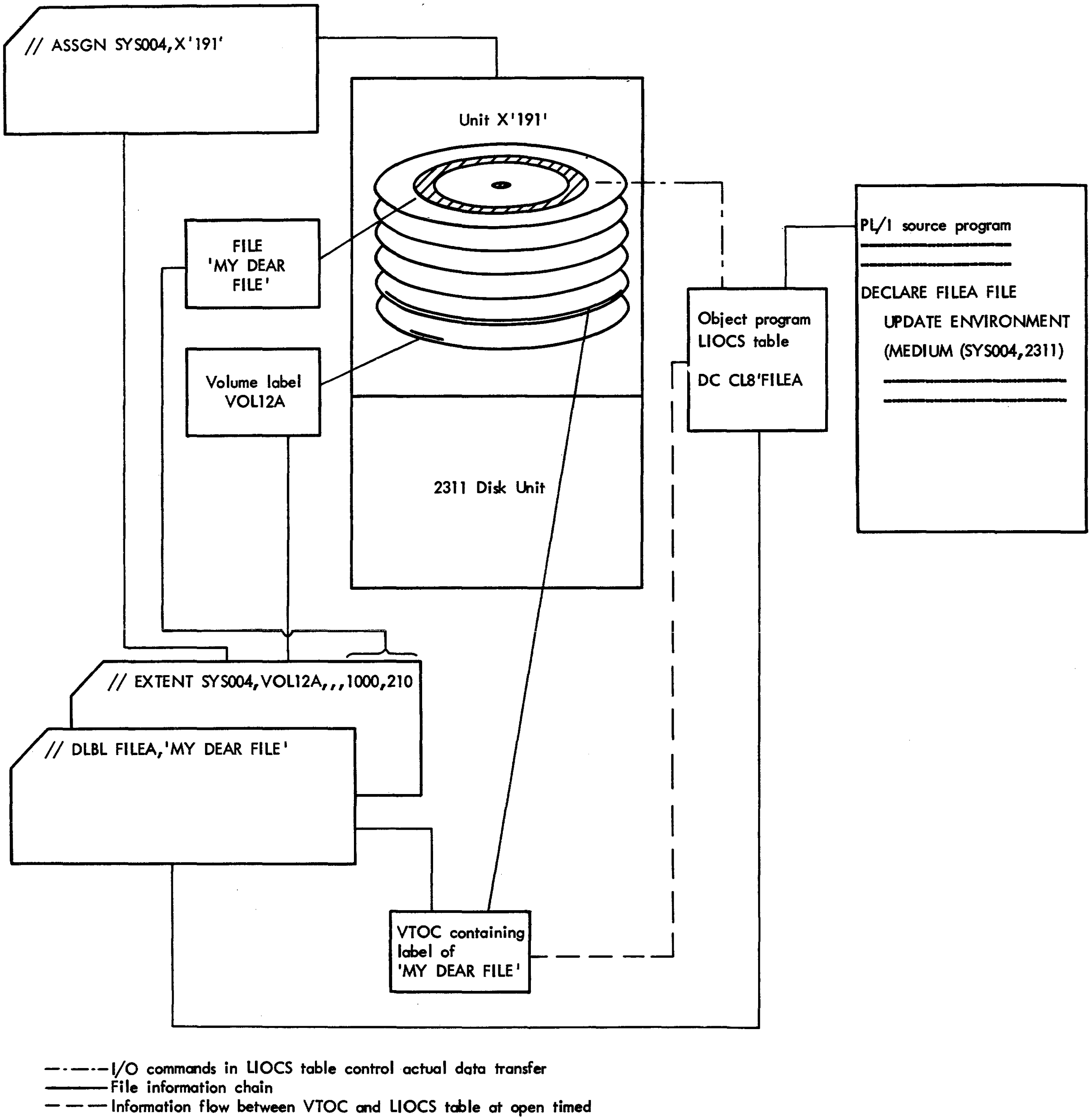


Figure 24. Program - Label Communication

Linkage Conventions

The user of PL/I programs is not concerned with internal linkage during activation and de-activation of blocks. To increase the capability and/or efficieny of his program he may, however, wish to combine modules written in the PL/I Subset language with modules written in Assembler language. For example, the programmer may wish to make use of the checkpoint facility. Since there is no checkpoint facility in PL/I, the user may call a subroutine written in Assembler language. Calling of subroutines written in FORTRAN or COBOL is not permitted.

Register Conventions

Some registers may have to be used during the execution of the called program. The user must save the contents of these registers by providing a save area. The address of the save area is contained in register 13. The general registers involved in linking a called procedure to the main program are listed in Figure 25. Note that floating-point registers are not saved by the called subroutine.

REGISTER	CONTENTS
1	Address of an argument list. This list contains the addresses of the arguments in the sequence stated in the argument (or parameter) list in the CALL, PROCEDURE, or ENTRY statement. Each argument requires one full-word on full-word boundary. In function references, the argument list is immediately followed by the address of the field where the information computed by the subroutine is stored.
13	Address of the save area.
14	Address to which the called subroutine returns when execution has been completed.
15	Branch address, i.e., the address in the called subroutine to which control is transferred for execution.

Figure 25. General Registers Used for Linking to a Subroutine Written in Assembler Language

Note: If control is transferred from an Assembler routine to another PL/I subroutine, registers 7 and 8 must contain the same values as when control was transferred to the Assembler routine.

Calling

Assume that register 13 has been set earlier in the program. To accomplish correct linkage, three additional registers (1, 14, and 15) must be set. Register 1 need not be set if no arguments are passed on and the call is not a function reference. The three different sequences that may be used to establish the required linkage between the main program and the called subroutine are shown in Figure 26.

Note: The DOS/TOS macro instruction CALL may be used to facilitate programming in cases 2 and 3 shown in Figure 26.

```
         L     15,=V(subroutine)
         BALR  14,15
------------------------------------------
         CNOP  2,4
         L     15,=V(subroutine)
         LA    14,*+6+4*n
         BALR  1,15
         DC    A(address1)
         DC    A(address2)
         ...
         ...
         DC    A(addressn)
------------------------------------------
         L     15,=V(subroutine)
         L     1,=A(listaddr)
         BALR  14,15
         ...
         ...
listaddr DC    A(address1)
         ...
         ...
```

Figure 26. Three Different Codings for Linking the Main Program and the Called Subroutine

Saving

Each calling program must provide a save area to store the contents of the general registers used by the called subroutine. When communicating with PL/I, the minimum length of this area is 20 full-words (80 bytes). The area may be expanded for storing intermediate results or data of

WORD	DISPLACEMENT	CONTENTS	STORED BY
1	0	DC X'03' DC AL3(INDIC)[1]	Calling module
2	4	Save area address of program that called the calling program	Calling module
3	8	Save area address of called program	Calling module if initialized by IJKSZCN[2]
4	12	Register 14	Called module
5	16	Register 15	Called module
6	20	Register 0	Called module
7	24	Register 1	Called module
. .	. .	....	....
18	68	Register 12	Called module
19	72	Invocation count	PL/I library
20	76	DSA pointer to embracing Static block	PL/I internal procedures

[1]INDIC is a full-word containing the information on the status of statement prefixes.
[2]Modules written in PL/I are initialized by IJKSZCN.

Figure 27. Layout of the First 20 Words of the DSA of a Calling Program

the storage class AUTOMATIC. This storage is called the DSA (Dynamic Storage Area).

Figure 27 shows the layout of the first 20 full-words of the DSA of a calling program. Assume that register 13 contains the address of the first word of the DSA.

The first instruction of a subroutine written in Assembler language must save the general registers 14, 15, 0, ..., 12. The DOS/TOS macro instruction SAVE can be used for this purpose. These registers must be saved even if their contents are not destroyed during execution of the subroutine. Otherwise, ON-conditions that may occur might not be handled correctly. The next steps to be taken are:

1. Store the contents of register 13 in word 2 of the subroutine save area.
2. Ensure that word 3 of the save area of the calling PL/I program is not destroyed by the Assembler subroutine.
3. Set register 13 to the address of the subroutine save area.
4. Ensure the addressability in case register 15 is destroyed during execution of this module.

Returning

Before returning control from the subroutine to the calling program, the contents of all registers must be restored. This is done as follows:

```
L    13,4(13)       RESTORES REG13
LM   14,12,12(13)   RESTORES REG14-12
BR   14
```

The last two instructions may be replaced by DOS/TOS macro RETURN (14,12)

The usage of LABEL parameters for returning from subroutines written in Assembler language necessitates a library call instead of a RETURN macro instruction. Therefore, the address of the LABEL parameter must be loaded into register 1. The routine IJKSZCP must be called next. The contents of register 13 are automatically saved by this routine. Therefore, they must not have been changed previously.

The following example shows how a library call can be used to return from a subroutine written in Assembler language by means of LABEL parameters.

```
        L       1,8(3)
* LOADS ADDRESS OF TABLE VARIABLE
        CALL    IJKSZCP
```

Note: The library subroutine IJKSZCN must be used to initialize the DSA if LABEL parameters are used.

Correlation Between PL/I And Assembler Modules

Modules written in the PL/I Subset language may call modules written in Assembler language and vice versa. However, if the program is combined of both PL/I and Assembler modules, one PL/I module with the attribute MAIN is required for correct initialization of the PL/I modules. Note that this MAIN procedure must be the first module to be executed.

Calling an Assembler Module

A module written in Assembler language is called according to the rules for calling external procedures either by means of a CALL statement or by means of a function reference. The Assembler module must satisfy all linkage rules given in this section. If the Assembler module does not call any other module, it must provide a minimum save area of two full-words. The 4-byte field INDIC pointed to by bytes 1 to 3 of the first word must contain the following information:

Byte 3 contains the standard prefix option switches, whereas byte 2 contains the actual prefix option switches. If INDIC is not initialized by the library subroutine IJKSZCN, the contents of byte 3 must be moved into byte 2 by the prologue of the module. The contents of byte 2 may be changed during execution of the module.

Bits 0 to 5 are used as switches with the following functions:

0 ZERODIVIDE
1 UNDERFLOW
2 OVERFLOW
3 FIXEDOVERFLOW
4 CONVERSION
5 SIZE

If the respective bits are on (1), the corresponding ON-condition is enabled. If they are off (0), the ON-condition is disabled.

If bit 7 is on, the PL/I interrupt-handling routine interprets a hardware fixed-point or decimal overflow condition as a SIZE error. If bit 7 is off, the condition is interpreted as FIXEDOVERFLOW.

Note: Word 2 of the save area and register 13 must be correctly initialized prior to the occurrence of any interrupt.

Assembler Module Calling PL/I Modules

Assembler modules that directly or indirectly call PL/I modules must provide a full DSA with a minimum of 20 full-words. This can be done by using the PL/I library subroutine IJKSZCN, which creates the DSA and provides correct handling of register 13. The subroutine sets the words 1, 2, 3, 19, and 20 of the DSA. Word 20 accommodates the contents of register 0 at the time when IJKSZCN was called. In internal PL/I procedures, this will be the address of the DSA of the statically embracing block. Word 3 contains the address of the storage location where IJKSZCN will construct the next DSA in case the present module calls another module.

Calling IJKSZCN destroys register 5. Therefore, register 5 should not be initialized by an Assembler module before IJKSZCN is called. IJKSZCN is called as shown below:

```
    LA    1,PBL
    L     15,=V(IJKSZCN)
    BALR  14,15
```

PBL is an 8-byte area containing the following information:

```
      DS   0F
PBL   DC   X'03'
      DC   AL3(INDIC)
      DC   A(length)
```

Note: Length is the length of the DSA in bytes.

The calling sequence for IJKSZCN should be preceded only by the SAVE macro instruction and two LR instructions providing for the addressability of the module itself and the argument list.

Passing Arguments

The argument addresses in the argument list point to the first byte of the data, array, or structure to be passed on. The address of a V-type constant is passed for an ENTRY argument. The word following the V-type constant contains a pointer to the DSA of the block statically embracing the passed procedure if the passed procedure is internal.

```
         L     R,address-in-parameter-list  LOADS ADDRESS OF PASSED ARGUMENT
         L     1,=A(argument-list)          ONLY IF ARGUMENTS ARE PASSED TO
*                                           CALLED ROUTINE. IT IS ASSUMED THAT
*                                           ARGUMENT LIST IS LOCATED REMOTELY
         LM    15,Ø,Ø(R)                    LOADS REG 15 AND Ø
         BALR  14,15
```

Figure 28. Format of Call to Entry Parameter

To allow for addressing of AUTOMATIC variables contained within the embracing block of an entry parameter, a call to the entry parameter should have the format shown in Figure 28.

If FILE arguments are used, the address in the argument list points to the file appendage. In addition to the information in byte 0, the first word of the file appendage contains the address of the DTF table for this file.

File arguments should be used very carefully in Assembler subroutines. Issuing an IOCS macro to a CCB which is part of a DTF table used by a PL/I program may destroy the synchronization between the PL/I program and logical IOCS. (Note that the CCB address must be inserted at object time when IOCS macros are used for filename parameters.) However, a programmer experienced in DOS/TOS IOCS may use filename parameters in Assembler subroutines to improve the capability of his program. For example, he may:

1. change DTF tables to allow handling of additional user labels or non-standard tape labels before opening a file.

2. change DTF tables to accomplish special stacker selection. PL/I programs use normal stackers for card input files and stacker 2 for punched-card output files.

3. issue a CNTRL macro instruction for seeking on a REGIONAL file to allow overlapping of seek time.

Figure 29 shows a PL/I procedure that calls a module written in Assembler language, which itself contains a function reference to another external PL/I procedure.

Data of the scope EXTERNAL may be shared between PL/I and Assembler modules.

Case 1. Data items within PL/I modules which are referred to by Assembler language modules:

Use EXTRN statement in the Assembler modules.

Case 2. Data items within Assembler modules which are referred to by PL/I modules:

Each data item must be a separate CSECT; otherwise incorrect addresses will be assigned when the programs are link-edited.

If a data item is a structure, all its individual elements can be coded together as a series of DCs or as a DS in the Assembler-language module under a single CSECT. In preparing such a structure, the PL/I structure mapping rules as described under Data Storage Mapping must, however, be observed.

The CSECT statement must be used if the respective name is not declared to be EXTERNAL in any PL/I program within the same phase. In all other cases, the programmer may use the instruction he considers convenient.

Note: Values returned by routines written in Assembler language must have the format specified for PL/I. Floating-point data must be normalized.

Checkpoint And Restart

A typical example for a procedure written in Assembler language is checkpointing and restarting. For convenience, both the checkpoint part and the restart part should be contained in the same routine.

If checkpointing is desired, the restart address, the end address, and the tape file positioning information must be provided. (Some additional information is required if the checkpoint is to be written on disk.) The restart address is known if it is in the same module as the checkpoint routine. The end address can be taken from word 3 of the save area, since this is the address of the next (not yet allocated) save area. Names of tape files can either be passed as parameters or addressed directly by using a V-type constant. (See the discussion on file parameters in the section Passing Arguments. The same applies to the usage of V-type constants.)

After the job has been restarted with the RSTRT statement, the restart routine must issue an STXIT macro for Program Check Interruption. The two address operands to be issued with STXIT are the external names IJKSZCI and IJKZWSI for the routine address and the save area, respectively. Moreover, the program mask must be reset.

Note: PL/I input files must not contain interspersed checkpoint records.

Figure 30 shows a coding example of a routine combining the checkpoint and the restart part. For detailed information refer to the following SRL publications:

```
// EXEC      PL/I
   CALLER:   PROCEDURE  OPTIONS (MAIN);
             DECLARE C CHARACTER (25) STATIC;
             .
             CALL SUBASM (A,B,C)   /* CALLS SUBROUTINE WRITTEN IN ASSEMBLER LANGUAGE */;
             .
             END;
/* - - - - - - - - - - - - - - - - - - - - - - - -
// EXEC      ASSEMBLY

             TITLE     'SUBROUTINE CALLED BY PL/I AND CALLING PL/I'
SUBASM       START     0                      PARAMETERS ARE A, B, C
             USING     *,9
             SAVE      (14,12)                SAVE REGISTERS
             LR        9,15                   ASSURE PROGRAM ADDRESSABILITY
             LR        3,1                    ASSURE ADDRESSABILITY OF PARAMETERS
             LA        1,PBL                  CREATE OWN DSA
             CALL      IJKSZCN
             .
             L         1,0(3)                 MAKE A ADDRESSABLE
             LE        0,0(1)                 LOAD A
             L         1,4(3)                 MAKE B ADDRESSABLE
             AE        0,0(1)                 ADD B
             .
             CALL      LEVEL3,(X,Y,RETURN)    CALL PL/I FUNCTION PROCEDURE
             .
             L         1,8(3)                 MAKE C ADDRESSABLE
             MVC       0(24,1),RETURN         C = RETURN || '.';
             MVI       24(1),X'4B'
             .
             L         13,4(13)
             RETURN    (14,12)                RETURN TO CALLING PL/I PROCEDURE
X            DS        F                      ARGUMENT  X
Y            DS        CL3                    ARGUMENT  Y
             DS        0F
PBL          DC        X'03'                  DATA TO CREATE DSA
             DC        AL3(ONINDICT)          POINTER TO ON-INDICATOR WORD
             DC        20A                    20-WORD DSA
ONINDICT     DC        3X'0'
             DC        B'11110000'            SIZE AND CONVERSION DISABLED
RETURN       DS        CL24                   SPACE FOR RECEIVING STRING  FROM
*                                             PL/I FUNCTION LEVEL3
             .
             .
             END
/*
- - - - - - - - - - - - - - - - - - - - - - - -
// EXEC      PL/I
   LEVEL3:   PROCEDURE (U,V) CHARACTER (24);
             DECLARE STR CHARACTER (21), V FIXED DECIMAL (5,2);
             .
             RETURN ('$$' || STR)             /* ONE BLANK AUTOMATICALLY
                                              ADDED AT THE END TO OBTAIN
                                              CORRECT LENGTH */;
             .
             END;
/*
```

Figure 29. Example of Linkages between PL/I Procedure and Assembler Module

For DOS

IBM System/360 Disk Operating System, System Control and System Service Programs, Order No. GC24-5036

IBM System/360 Disk Operating System, Supervisor and Input/Output Macros, Order No. GC24-5037

For TOS

IBM System/360 Tape Operating System, System Control and System Service Programs, Order No. GC24-5034

IBM System/360 Tape Operating System, Supervisor and Input/Output Macros, Order No. GC24-5035

```
CPRS     TITLE   'CHECKPOINT-RESTART ROUTINE'
*                CALLED BY A PL/I PROCEDURE. THE INFORMATION ON THE
*                POSITIONING OF THE TWO FILES TAPEIN AND TAPEOUT IS
*                TO BE CHECKPOINTED.
CHPRES   START
         USING   *,12
         SAVE    (14,12)
         LR      12,15                   SET BASE REGISTER
         LA      1,PBL                   CALL PL/I PROLOGUE ROUTINE
         L       15,=V(IJKSZCN)
         BALR    14,15
         L       2,=V(TAPEIN)            PREPARE FILE TABLE
         L       2,0(2)
         ST      2,FILETAB+2
         L       2,=V(TAPEOUT)
         L       2,0(2)
         ST      2,FILETAB+6
         L       2,8(13)                 LOAD END ADDRESS
         BALR    3,0                     SAVE PROGRAM MASK IN AUTOMATIC
         ST      3,80(13)                          STORAGE
         CHKPT   SYS007,RESTART,(2),TPOINT         CHECKPOINT ON SYS007
         B       RETURN
*                                        RESTART PART. NOTICE THAT ALL GENERAL
*                                        REGISTERS ARE AUTOMATICALLY RESTORED.
RESTART  L       0,=V(IJKSZCI)           SET PROGR. CHECK INTERRUPTION EXIT.
         L       1,=V(IJKZWSI)
         STXIT   PC,(0),(1)
         L       2,80(13)                SET PROGRAM MASK.
         SPM     2
RETURN   L       13,4(13)                RETURN TO PL/I CALLER
         RETURN  (14,12)
         DS      0F
PBL      DC      X'03'                   ARGUMENT FOR IJKSZCN
         DC      AL3(INDIC)
         DC      22A                     PL/I SAVE AREA DEFINITION + 1 WORD FOR
*                                        SAVING PROGRAM MASK (MUST BE MULTIPLE
*                                        OF EIGHT).
INDIC    DC      A(0)                    ON INDICATORS
TPOINT   DC      A(FILETAB)              POINTER TO FILETABLE
         DC      A(0)                    PIOCS FILES NOT USED
         CNOP    2,4
FILETAB  DC      H'2'           *        FILE TABLE
         DS      2F             *
         END
```

Figure 30. Coding Example of Combined Checkpoint and Restart Routine

General Programming Information

This section describes some programming techniques to save storage, produce a faster object program, perform functions not easily achieved with more conventional PL/I language facilities, make a program fit into the available storage, etc.

STATEMENT FORMAT

The first column of every source text card must be blank. Columns 73-80 are ignored; they may contain any information.

PROGRAM SEGMENTATION

Every program should be written so that it can be segmented if necessary. The case of storage overflow should be provided for so that, if it does occur, it can be handled easily. Breakpoints in the logic of a program, i.e., points where a program phase can be terminated and a subsequent phase entered, should be numerous.

Data common to successive programs can be kept through the proper use of the EXTERNAL attribute. However, not all data need be external.

Programs that read data, compute, and write results lend themselves to segmentation most readily. Wherever practical, entire programs should be written as sequences of calls for subroutine procedures because each call is a logical breakpoint. Thus, the entire storage can be loaded with as many subroutines as can be accommodated. The next phase then repeats the process of loading the storage with the next group of subroutines, etc.

PROGRAM EXPANSION

In general, no more than 90 % of the storage available for any program phase should be used during the first six months of its life because, at one time or another, every program tends to expand due to

1. programming errors,
2. the need to expand the original function,
3. errors in the system program or in the associated subroutines, and/or
4. an increase of the data storage requirements.

If a program uses the entire storage and no space is left for eventualities, reasonable solutions become difficult. If, however, normal expansion was provided for, the overall job is much easier.

CONVERSIONS

If a numeric variable is to be used frequently in expressions, it is much more economical to convert the variable to coded form once and use the coded form in all expressions. This is easily done by means of an assignment statement.

Conversions implicit in IF statements follow the rules for arithmetic conversions, and the intermediate precisions should be considered when using such expressions.

For example, in case 3 (IF X=U THEN...) of the following sample program the conversion rules are applied to X, giving a short-precision floating-point number which is then expanded (padded) with trailing zeros to long precision before the actual comparison operation. Thus expression 2 will be executed, not expression 1. However, if X and U are assigned with a value which will be the same in both short and long precision (e.g. 0.5), then expression 1 will be executed.

In evaluating the following program, refer to Section F: Data Conversion in IBM System/360, Disk and Tape Operating Systems, PL/I Subset Reference Manual, Order No. GC28-8202.

```
Z:  PROCEDURE OPTIONS(MAIN);
    DECLARE X DECIMAL FIXED(5,2);
    DECLARE T DECIMAL FIXED(15,2);
    DECLARE Y FLOAT(6);
    DECLARE U FLOAT(16);
    X=123.45;
    Y=123.45;
    T=123.45;
    U=123.45;
    IF X=Y THEN expression 1; /* Yes */
           ELSE expression 2; /* No */
    IF X=T THEN expression 1; /* Yes */
           ELSE expression 2; /* No */
    IF X=U THEN expression 1; /* No */
           ELSE expression 2; /* Yes */
    IF Y=T THEN expression 1; /* No */
           ELSE expression 2; /* Yes */
    IF Y=U THEN expression 1; /* No */
           ELSE expression 2; /* Yes */
    IF T=U THEN expression 1; /* Yes */
           ELSE expression 2; /* No */
END;
```

For an example showing the conversion of characters into binary numbers, refer to Appendix L. Programming Examples, "Conversion of Numbers in Character Form Into Binary Numbers".

USE OF UNSPEC

The UNSPEC pseudo variable and the UNSPEC built-in function handle the internal representation of data. The internal representation of data is summarized in Figure 50 and described in detail in the section Data Storage Requirements.

The programmer must make sure that values assigned by the UNSPEC pseudo variable have the correct format. Otherwise, the results are unpredictable. Note that the internal representation of floating-point data is normalized. Consider the following example:

```
DECLARE A FLOAT, B CHARACTER(1), C FIXED
   DECIMAL(5,3);
B= '8';
X:  PUT EDIT (UNSPEC(B)) (SKIP,B);
Y:  UNSPEC(A)=(31)'0'B || '1'B;
Z:  UNSPEC(C)=(16)'0'B || '01100000'B;
```

The result of statement X is 11111000. Statement Y yields unpredictable results since the value to be assigned is not normalized. Statement Z also yields unpredictable results since the last half-byte does not contain a valid sign for packed decimal data representation.

For an example of the UNSPEC built-in function, see Appendix L. Programming Examples, "Conversion of Numbers in Character Form to Binary Numbers".

COMPUTATIONS WITH OVERLAY

Whenever possible, input/output phases should be performed separately from computational phases. Thus, the I/O subroutines including the E and/or F conversion subroutines are never in storage simultaneously with the other subroutines (arithmetic, base, and scale conversion, etc.). This can result in considerable storage savings (see Figure 31).

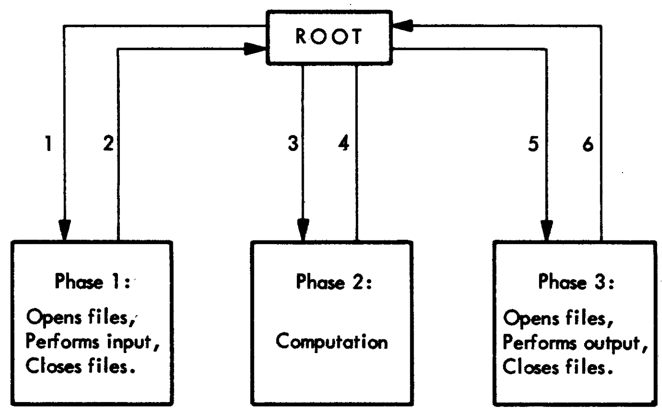


Figure 31. Example of Using Overlays to Perform Computations and I/O Operations Separately

BLOCKING

It may happen that one large set of data is used in a program only at one specific point, that another large set of data is used at another point, etc. In this case, each set of data used at one point should appear in a separate block so that the data is AUTOMATIC by default (unless declared to be STATIC) and allocated only when the respective block is active. Thus, the same storage area can be used for all data sets to be used.

SIMULATION OF P-FORMAT ITEMS

The PICTURE-format items of OS PL/I are a more powerful tool for editing than the format items available in DOS/TOS PL/I. However, numeric fields in edit-directed I/O operations can easily be simulated by overlaying numeric fields with character strings using the DEFINED attribute. An example is shown below:

```
DECLARE U PICTURE '$$,$$9.V99BCR',
B CHARACTER (12) DEFINED U;
...
...
U= ...
PUT SKIP EDIT ('U = ', B) (2 A);
```

SIMULATION OF ARRAYS OF STRUCTURES

Since arrays of structures are not permitted in the PL/I Subset language, it is recommended to simulate arrays of structures by using arrays in structures, i.e., by arrays that are not themselves structures. Should this not be feasible, arrays of structures may be simulated by using based structures. This can be accomplished by assigning to the pointer the value of an element of a character-string array. The programmer is responsible for satisfying all boundary requirements.

The following example shows the handling of structures in OS PL/I versus DOS/TOS PL/I:

OS PL/I

```
 DECLARE 1 A, 2 B FLOAT, 2 C(10), 3 D
         PICTURE '9999',
         3 E PICTURE 'XX',
         3 F PICTURE '99V99';
         .
         .
   DO I=1 TO 10;
      .
      .
   A.D(I)=....
      .
      .
   END;
```

This could be written in DOS/TOS as follows:

```
 DECLARE PTR POINTER, 1 A, 2 B FLOAT, 2 C
         (10) CHARACTER(10), 1 X BASED
         (PTR), 2 D PICTURE '9999', 2 E
         PICTURE 'XX', 2 F PICTURE '99V99';
         .
         .
   DO I=1 to 10;
   PTR=ADDR(A.C(I));
      .
      .
   X.D=....
      .
      .
   END;
```

USE OF THE DEFINED ATTRIBUTE

For scalar variables or arrays, the DEFINED attribute is used when

1. a variable is to have more than one name (correspondence defining), or

2. two separate variables are to occupy the same storage area provided they are never required simultaneously (overlay defining).

In either case, the actual storage requirement is that of the base identifier and not the sum of the storage requirements of all variables. For restrictions on the use of the DEFINED attribute for scalar variables and arrays see the Subset language publication.

The use of the DEFINED attribute can result in considerable savings of storage. This is obvious for arrays, e.g., the statement

```
DECLARE A (5,9,7), B (5,9,7) DEFINED A;
```

merely requires the storage area for array A (315 data items). Without the DEFINED attribute, the storage requirements would be twice as much. But in spite of the more severe restictions on the use of the DEFINED attribute for structures, it can also be of considerable use in this case.

USE OF BASED VARIABLES WITH STRUCTURES

The restrictions on the use of the DEFINED attribute for structures can be circumvented by using based variables instead of the DEFINED attribute. For example, in the statement shown below structures U and I are based variables. They are never allocated any storage. Instead, the pointer variable P can be used to utilize the storage occupied by structure A whenever structures U and I are referred to (provided that structure A is not required at the same time).

```
 DECLARE P POINTER,
     1 A ALIGNED,
       2 B BIT(7),
       2 C FIXED DECIMAL(13,2),
       2 D CHARACTER (21),
     1 U ALIGNED BASED (P),
       2 V BINARY,
       2 W,
       2 X BIT(19),
     1 I BASED (P),
       2 J,
       2 K,
       2 L;
```

The statement

```
        P = ADDR (A);
```

would cause any subsequent reference to either U or I or any component of U or I to point to the storage area occupied by A. This simulates the use of the DEFINED attribute with all of its restrictions removed except that the based structures must be mapped in the same or less storage than the map of the overlaid structure. This process may be extended even further so that a based variable structure occupies the storage area of any one of many structures. This is demonstrated below:

```
DECLARE (V1,V2) POINTER,
   1 A, 2 B, 2 C,.......,
   1 U ALIGNED, 2 F, 3 Q BIT (9),...,
   1 R, 2 Z, 2 M, 3 S CHARACTER(2),...,
   1 P1 BASED (V1), 2 L, 2 X,....,
   1 P2 ALIGNED BASED (V2),
      2 D BIT(9),.;

V1=ADDR(A);
.
.  using P1 here points to A
.
V1=ADDR(U);
.
.  using P1 here points to U
.
V2=ADDR(R);
.
.  using P2 here points to R
.
V1=ADDR(R);
.
.  using P1 here points to R
etc.
```

Of course, the storage requirement of structure P1 must not exceed that of the smallest of either A, U, or R. Since the structure P2 does not point to A or U in this procedure, the only prerequisite is that its storage requirement must not exceed that of R.

Note on Compatibility: The structure-mapping technique for OS PL/I is identical to that for DOS/TOS PL/I in every respect but one. The exception is that DOS/TOS PL/I causes all structures to begin at double-word boundaries. This is accomplished by padding to the left of the first addressable element until byte zero is reached. (See the section Structure Mapping Rules, rule 11.)

OS PL/I begins structures at the first addressable element. This difference is of no significance in PL/I programming unless the above-described technique is employed. When this technique is used, compatibility is guaranteed if at least one element of the non-based structure has a stringency level that is as high as that of the element (or elements) of the highest stringency level of the based structure.

For the D Compiler the pointer associated with a based structure must be assigned an address value which insures that the first element of the structure has the same distance to a double-word boundary as it would have if the structure was not based.

Note: The use of based structures to avoid the use of the DEFINED attribute is dependent on structure mapping which, in turn, is implementation-defined.

REDEFINITION OF ATTRIBUTES

The two preceding sections showed that a number of structures can be made to occupy the same storage area. Similarly, a single character-class variable may be conceived of in many different ways. Consider the declaration shown below.

```
DECLARE A CHARACTER (80),
      1 B DEFINED A,
        2 C CHARACTER (40),
        2 D CHARACTER (30),
        2 E CHARACTER (10),
      1 F DEFINED A,
        2 G PICTURE '(8)9',
        2 H PICTURE '9',
        2 I CHARACTER (61),
        2 J PICTURE '(5)9V(5)9',
      1 K DEFINED A,
        2 L (10) PICTURE '$$(4)9V(2)9';
```

A represents a string of 80 characters whereas B, F, and K represent three distinct structures. However, these three distinct structures refer to the same storage area as A. This technique is especially useful in programs with many different structures to be read. For instance, the program may read a character string and, depending on its first character, treat it in any one of many different ways without requiring space for each possible structure.

USE OF THE 48-CHARACTER SET

If the 48-character set is used, the word PT, in addition to those listed in the Subset language publication, is a reserved keyword. Programs written in the 60-character set can be read if 48C is specified in the OPTION statement (but not vice versa).

SIZE OVERFLOW

If a size overflow occurs during F-format output, the output field will contain asterisks, even if SIZE is disabled.

USE OF THE DISPLAY STATEMENT WITH THE REPLY OPTION

Using the DISPLAY statement with the REPLY option is possible only if a 1052 Printer-Keyboard is available.

PRECISION OF DECIMAL DATA

The use of an odd precision for decimal data will keep the generated code at a minimum and thus improve the program performance.

CHANGING THE TAB CONTROL TABLE

List-directed output to PRINT files automatically aligns data on preset tab positions. For the D-level compiler, these tab positions are 1, 25, 49, 73, 97, and 121.

The tab positions are determined from the control table IJKTLTB which is catalogued under this name in the relocatable library. To obtain different tab positions, the programmer only has to change this table by specifying the following macro instruction:

```
IJKZL  (tab,[tab,...,]FF)
```

In this macro instruction, 'tab' is a decimal constant indicating the desired tab postion, and 'FF' indicates the end of the table. Tabs must be specified in ascending sequence, and their values must range between 1 and 144. The length of the tab list specified in the IJKZL macro instruction must not exceed 127 characters, including opening and closing parentheses and commas.

Following is an example of the IJKZL macro instruction and the control statements required to change the tab settings.

```
// JOB IJKTLTB
// OPTION DECK
// EXEC ASSEMBLY
   IJKZL (1,25,50,75,100,FF)
   END
/*
* THE RESULTING OBJECT DECK IS INPUT
* FOR THE FOLLOWING EXEC MAINT PROGRAM
// EXEC MAINT

   (Object deck)

/*
/&
```

If the specified tab positions do not fall between the values 1 and 144, or if they are not in ascending sequence, one of the following messages is issued:

```
PARAMETER GT 144

PARAMETER NOT IN ASCENDING ORDER
```

IMPROVEMENT OF DO-LOOPS

The execution time of a DO-loop can be reduced if a fixed binary variable is used as control variable in the DO statement.

For example, if in the statement

```
DO var = exp1 TO exp2 [BY exp3]
         [WHILE (exp4)];
```

'var' is a fixed binary value, all constants used as exp1, exp2, and exp3 will be converted to fixed binary during compilation, in order to avoid conversions during execution.

ROUNDING CN OUTPUT WITH E AND F FORMAT ITEMS

On output, data edited by the E- or F-format are rounded at the last numeric position, and not truncated.

HANDLING BLANK NUMERIC FIELDS

When using a PICTURE specification with '9's for numeric fields and the field is blank, a program check (data exception) occurs.

This is a particular problem for card input where fields are often left blank rather than filled with zeroes.

The problem can be avoided by declaring the field with PICTURE using 'Z' rather than '9' or with PICTURE using '9' plus one of the overpunch picture characters T, I, or R.

Assume card columns 1-10 are numerical and may or may not be punched.

```
DECLARE COL_1 PICTURE '(10)9';
DECLARE COL_1 PICTURE '(10)Z';
DECLARE COL_1 PICTURE '(9)9(1)I';
```

The first DECLARE statement causes a data exception if the field is blank. No data exception occurs for blanks in columns 1 through 10 if the field is declared as shown by the second and third DECLARE statements.

The programmer should, however, be aware that the exclusive use of '9's in a PICTURE specification results in more efficient code.

USE OF LIST-DIRECTED AND EDIT-DIRECTED DATA TRANSMISSION

When the list-directed and edit-directed transmission modes are used for the same file, the user is responsible for the correct positioning of the file.

USE OF PICTURES WITH STREAM-ORIENTED DATA TRANSMISSION

1. Character-string pictures:

   The D Compiler handles them in the same way as normal character-string variables.

2. Arithmetic pictures:

   All kinds of arithmetic pictures are possible in the data lists of GET and PUT statements.

   a. Edit-directed transmission: Only such items in the data stream which can be described by the E or F format can be transferred from (PUT) or into (GET) arithmetic pictures. If, on output, the programmer wants the character representation of the picture, he should use the CHAR built-in function as pseudo-variable with the picture as argument in the data list.

   b. List-directed transmission: On input, only [+|-] arithmetic constants can be transferred into arithmetic pictures. On output, the character representation will be transferred into the data stream.

PICTURE SPECIFICATIONS

Storage can be saved by proper declaration of fixed numeric PICTURE fields.

1. PICTURE specifications without drifting characters: make the first digit position 'Z' or '*' and avoid writing the first '9' in the field immediately following an insertion character.

   ```
   'Z9,99.V99' is better than '99,99.V99'
   'SZZ9999'   is better than 'S999999'
   '+ZZ,Z999'  is better than '+ZZ,9999'
   ```

2. Specifying "V." rather than ".V" results in better code in the following cases:

   (a) If the first fractional digit position is the first '9' in the field, then 'ZZ,ZZZV.99' is better than 'ZZ,ZZZ.V99'.

   (b) If a drifting character or zero-suppression is specified past the decimal point, then '$$$$$V.$$' is better than '$$$$$.V$$' '*****V.**' is better than '*****.V**'

3. Give the variable in the right-hand side of an assignment statement the attribute DECIMAL FIXED with the same scale and precision as the PICTURE. If there is an expression on the right-hand side try to produce the desired scale and precision.

4. Zero-suppression with "*" costs more storage (code) than zero-suppression with "Z" if

   "+" or "-" is used (static or drifting) or

   "B" is used <u>after</u> the last digit position.

5. If the PICTURE does not contain at least one "9", "T", "I" or "R", but does contain a "V", additional code is required for clearing the field in case of a zero value.

ENDPAGE WITH MULTIPLE-LINE PUT

When using a PUT statement producing multiple lines, the ENDPAGE condition should not be enabled because of possible loss of data:

```
ON ENDPAGE(F) GOTO X;
PUT FILE(F) EDIT(data-list)(format-list);
X: new header;
```

In this example the ENDPAGE condition may be raised during execution of the data list (assuming multiple-line output); but no return from X is possible, so that the rest of the data list will be ignored.

Program-Checkout Facilities

Certain language features are provided in PL/I to assist the programmer in debugging his program. These facilities are described below.

For a detailed discussion of how to debug a PL/I program, refer to the section Debugging PL/I Programs in the SRL publication IBM System/360 Disk Operating System, System Programmer's Guide, GC24-5073.

Exhibit Changed

The EXHIBIT CHANGED feature uses the library routine IJKEXHC which requires approximately 1200 bytes of main storage.

In addition, each variable appearing in a CALL IJKEXHC statement requires about 12 bytes of storage plus a field containing the variable name plus a field containing the value of the variable in static storage.

Function:

The first execution of the CALL IJKEXHC statement causes the printing of the names listed in the statement, and their values in hexadecimal notation.

General Format:

CALL IJKEXHC (name , name ....);

The argument 'name' can be an unsubscripted, unqualified name representing an element, an array, or a structure which are not contained in an array or structure, or it can be a string or arithmetic constant. However, it cannot be a label constant, an entry name, or a file name.

General Rules:

1. Names with the attribute AUTOMATIC are printed each time the CALL IJKEXHC statement is first executed after a new block activation. Names with the attribute STATIC are printed only the first time the CALL IJKEXHC is executed if the activated block is internal. They are printed each time the CALL IJKEXHC statement is executed if the activated block is external.

2. On subsequent passes of the CALL IJKEXHC statement, the names and values are printed only if the value has changed since the time the statement was last executed.

3. If there are several CALL IJKEXHC statements in one program, they are independent from each other.

4. The maximum number of arguments for one CALL IJKEXHC statement is 12. If an argument has the BASED or DEFINED attribute, the related pointer or base variable is counted as an argument, regardless of whether it has been specified in the argument list or not.

5. Up to 30 names can be checked by CALL IJKEXHC statements within one block, if 10K bytes are available to the compiler. For each additional 4K, up to 46K, 30 additional names can be checked.

6. The values of element variables having the attributes BINARY FIXED, BINARY FLOAT, DECIMAL FIXED, DECIMAL FLOAT, CHARACTER, BIT, or PICTURE are also printed in their external form.

Tracing

The TRACING feature uses the library routine IJKTRON which requires 1258 bytes of main storage.

In addition, about 34 bytes of storage are required for each CALL IJKTRON statement and about 12 bytes for each CALL IJKTROF statement.

Function:

The two statements, CALL IJKTRON and CALL IJKTROF, function like a switch. IJKTRON switches tracing on, while IJKTROF turns it off.

If tracing is enabled for a block, the following information is printed on SYSLST:

1. On entry, the external name of the block, or, if the block has no label, the internal name of the block.

2. On leaving a block via an END or RETURN statement, a message is given to indicate the exit. If the STMT option is active, the statement number of the END or RETURN statement is printed as well as the number of the statement to which the program returns.
Note: If for 'CALL entry name' information should be printed, tracing must

be enabled for the block which contains the entry name.

3. For each executed GOTO statement
   a. the external name (up to eight characters) and value of the label variable or constant if the GOTO statement is not in an on-unit, or
   b. the ON-condition and the value of the label variable or constant if the GOTO statement is in an on-unit.

   If the STMT option is active, the statement number of the GOTO statement and the statement number of the target statement are also displayed.

General Format:

```
CALL IJKTRON;
CALL IJKTROF;
```

General Rules;

1. Tracing can be explicitly enabled in a block by a CALL IJKTRON statement.

2. A CALL IJKTROF statement explicitly disables tracing in a block.

3. If tracing is neither explicitly enabled nor disabled in a block, the tracing status of the dynamically containing block is applied.

4. The dynamically containing block of the main procedure has tracing disabled.

5. At least one of the two statements has to be specified if tracing is to appear in an external procedure.

6. When calling an external procedure (provided tracing is enabled at the time of the call), the called phase must have a call for either IJKTRON or IJKTROF. If this condition is not satisfied, the results are unpredictable in the event of an interrupt.

Example:

```
 1) A1:  PROCEDURE OPTIONS (MAIN);
                 .
                 .
                 .
 2)      CALL IJKTRON;
                 .
 3)      GOTO A11;
                 .
                 .
                 .
 4)      A11: CALL B1;
 5)           C=3;
                .
                .
 6)           GOTO A2;
                .
                .
 7)           A2: BEGIN;
                    .
                    .
 8)                 CALL IJKTROF;
                    .
                    .
 9)                 GOTO A21;
                    .
                    .
10)                 A21: CALL IJKTRON;
                    .
                    .
                    .
11)                 END A2;
               .
               .
12)      END A1;
13) B1:  PROCEDURE;
               .
               .
14)      CALL IJKTROF;
               .
               .
15)      RETURN;
               .
               .
16)      END B1;
```

This example causes the following (the statement numbers in the above example are referenced in the left-hand margin below):

1) When the main procedure is invoked, no tracing status is specified and, therefore, tracing for this block and, per definition, for the dynamically containing block is disabled.

2) Tracing is explicitly enabled in block A1.

3) The external name and value of label A11 are printed.

4,13) No tracing status is specified for this block; therefore, the (enabled) status of the containing block A1 is adopted and the name of the procedure B1 is printed.

14,15) Tracing is explicitly disabled for this block, and no message is printed when control returns to statement 5.

6) The external name and value of the label A2 are printed since tracing is still enabled in A1.

7) With the activation of block A2 tracing is neither enabled nor disabled, therefore the (enabled) status of block A1 is adopted and the external name of block A2 is printed.

8,9) Tracing is disabled for block A2 and no message is printed.

10,11) Tracing is again enabled and the pass of the END statement is indicated on SYSLST.

12) Since tracing in the main routine is still enabled, the pass of this END statement is also indicated on SYSLST.

The DYNDUMP Routine

The statement

CALL DYNDUMP (argument-list);

may be used to have the internal representation of the items in the argument list displayed in hexadecimal notation. The argument list may contain up to 12 items. Each argument must be either a scalar expression or a variable name.

The DYNDUMP routine (56 bytes in length) uses the PL/I Control routine and the SYSPRINT file with the associated module. No additional I/O subroutines are required. Thus, the DYNDUMP routine provides an economical way of displaying intermediate results during checkout of PL/I programs with a minimum of library and I/O module overhead.

The following example shows the use of the DYNDUMP routine.

```
DECLARE A FIXED(5,2), B(10), C BIT(1);
....
.....
CALL DYNDUMP (A,B,C);
```

Three items are displayed: A as 3 bytes (6 hexadecimal digits), B as 40 bytes (80 hexadecimal digits), and C as one byte (2 hexadecimal digits).

Note: The current value of C is indicated by the first bit. If the variable length is an exact multiple of 48 bytes, the end address+1 will be printed on the next line in order to delimit the variables for ease of reading.

Locating Execution-Time Errors

If a PL/I object program is terminated by the PL/I Control routine and the DUMP option is active, the problem program area is printed (dumped) on the device assigned to SYSLST. The following information is intended to assist the programmer in analyzing a program dump and to locate the error that caused the termination of this program.

Note: There is no guarantee that main storage organization will always be as described below. Severe programming errors, e.g., illegal use of based variables, the UNSPEC pseudo variable, or use of user-written Assembler subroutines may yield unpredictable results.

If the error was caused by an I/O operation, look up the Linkage Editor storage map to find the address of the DTF table for the respective file. The first word of the DTF table contains the address of the corresponding CCB. For details on the CCB refer to the SRL publications describing the DOS/TOS Supervisor and I/O macro instructions.

Data declared with the attribute EXTERNAL can be found using the addresses given in the Linkage Editor storage map.

To determine the absolute address of static internal data refer to the offset table listing (see the section Offset Table Listing).

To locate the storage allocated to an automatic variable, the offset of the variable within the DSA (Dynamic Storage Area) is determined from the offset table, and this offset is added to the DSA address of the block to which the variable is internal. The address of the DSA is automatically loaded into register 13 at prologue time. Word 20 of the DSA contains the DSA address of the statically embracing block.

The load point of the main DSA is the next double-word boundary after the highest high-core address of all external blocks linked in the program.

More than one DSA may be allocated, i.e., if more than one block is active. To find the DSA of the block where the error is detected, check the byte pointed to by register 13. If this byte contains either X'h1' or X'h3' (h may be any hexadecimal digit), register 13 points to the relevant DSA. In this case, the error message was most probably caused by a Program Check interrupt.

The instruction that caused the interrupt can be found by means of the diagnostic message. The old PSW and the registers can be found at the location with the external label IJKZWSI.

If the byte contains X'05', register 13 points to a LSSA (Library Standard Save Area), the second word of which contains the chain-back word. If this again points to a LSSA, repeat the chain-back process until the chain-back word points to a DSA. This DSA then belongs to the block where the error was detected.

To identify the block, go to the chain-back address of the relevant DSA. If this points to another DSA, word 5 of the DSA contains the absolute address of the block. The block can then be identified using the object code listing and the Linkage Editor storage map. If the chain-back word does not point to a DSA, the relevant DSA is the DSA of the MAIN procedure (see Figure 32).

The chain of DSAs resembles the current environment at the point of execution where the error was detected. Each DSA in the chain has its corresponding currently active block. From where and at which location a specific block is activated can be determined by means of the DSA of the calling block. For detailed information on the first 20 words of the DSA refer to the section Linkage Conventions.

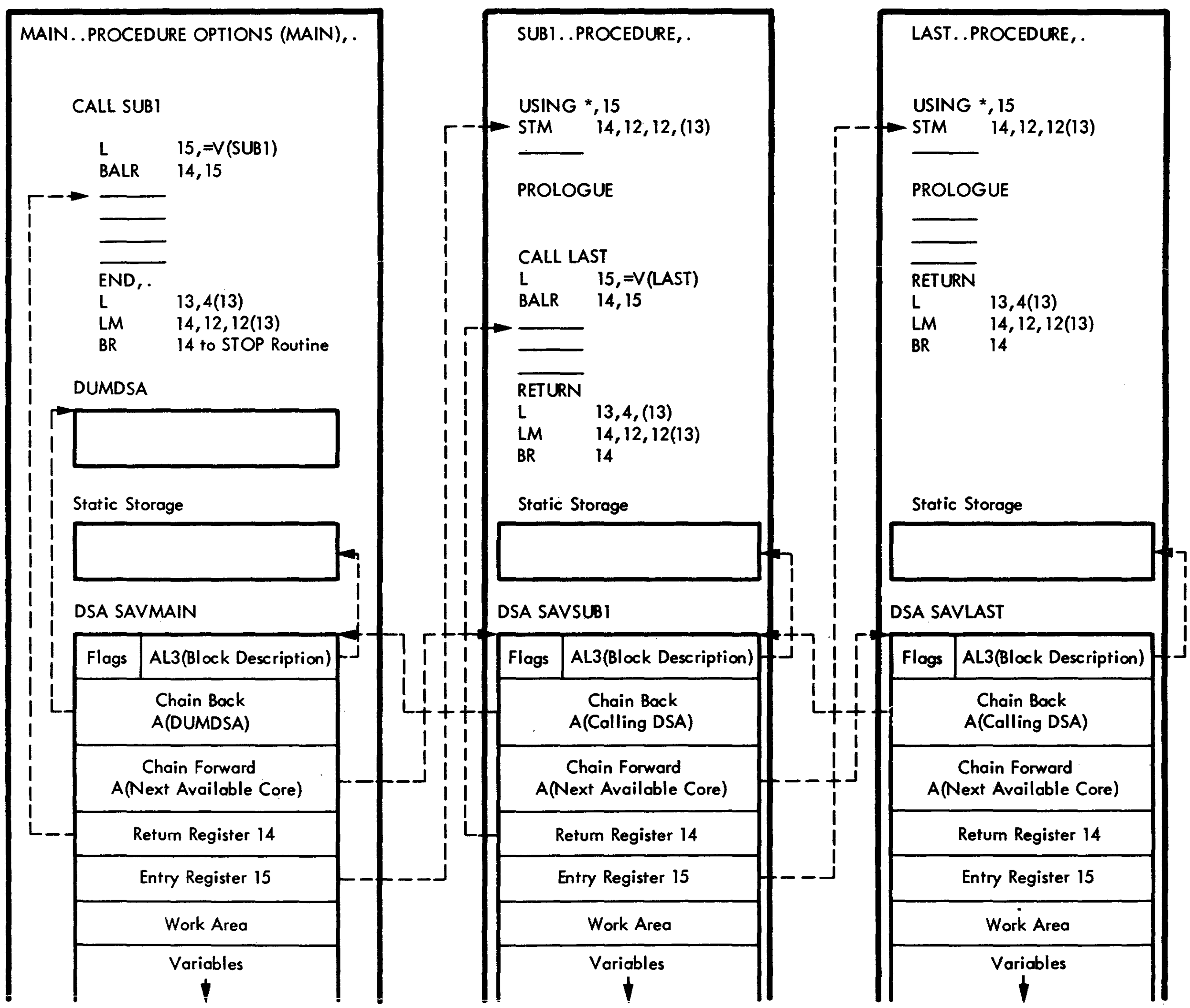


Figure 32. DSA Chaining

Data Storage Requirements

The storage requirements for data depend on the following two factors:

1. The storage required for the data itself.

2. The storage required for the data descriptor. (The data descriptor is required whenever the compile-time data description is to be used in the object program.)

Data Descriptors

A data descriptor may describe more than one data item. Only one data descriptor is required for a group of data items that have identical (either explicitly or implicitly declared) attributes, e.g., for individual variables of identical attributes or for array elements. Thus, the statement

```
DECLARE (A, B, C(21), D) FIXED DECIMAL
  (5,2), (E, F, G) PICTURE '$99.99';
```

requires only two descriptors: one describing A, B, the 21 C's, and D, and one describing E, F, and G. Constants (except those used in output lists), label variables, label constants, or pointer variables do not require a descriptor.

A data descriptor and, therefore, storage in the object program is required only if the pertinent data item is used in a conversion or I/O library subroutine.

Fixed decimal Float decimal Fixed binary Float binary Sterling constants	Coded	Arithmetic
Fixed decimal Float decimal Sterling	Numeric (picture-specified)	
Character Bit Picture-specified character	String	Non-arithmetic
Label	Label	
Pointer	Pointer	

Figure 33. Types of Data Items

Data Items

Figure 33 shows the types of data items that require storage. In the following text, the storage requirements for each of these items are specified and illustrated by means of examples. The storage requirements given in these examples pertain to the data only. Unless otherwise stated, references to coded arithmetic and string data apply to both variables and constants. Other data types will have constants and variables explicitly differentiated in regard to storage requirements.

CODED ARITHMETIC DATA

Binary Fixed

Default precision: 15 bits
Maximum precision: 31 bits
Storage requirements:

1. Descriptor
   3 bytes (if required)

2. Data
   4 bytes internal fixed-point regardless of declared or default precision. Scale factor must not be specified.

Figure 34 shows the storage requirements for the binary fixed data declared in the following sample statement:

```
DECLARE I(8,5), A FIXED BINARY(7),
  J STATIC, Z(3) FIXED BINARY(27);
```

DATA ITEM	DECLARED ATTRIBUTES	DEFAULT ATTRIBUTES	BYTES
I	Dimension (8,5)	FIXED BINARY Precision (15)	160
A	FIXED BINARY Precision (7)	None	4
J	STATIC	FIXED BINARY Precision (15)	4
Z	Dimension (3) FIXED BINARY Precision (27)	None	12
		TOTAL	180

Figure 34. Example of Binary Fixed Data

Decimal Fixed

Default precision: (5,0)
Maximum precision: (15,0)
Storage requirements:

1. Descriptor
   3 bytes (if required)
2. Data
   Packed decimal form --
   4 bits = 1/2 byte for each digit. The sign is always stored and requires 1/2 byte. The total storage required must be expressible in byte form, i.e., +5.2 requires 2 bytes (1/2 byte for the sign, 1 byte for the two digits, 1/2 byte padding).
   Scale factor range: 0 to 15 (if present).

Figure 35 shows the storage requirements for the decimal fixed data declared in the following sample statement:

```
DECLARE A FIXED, B(5,2,3) FIXED, I FIXED
  STATIC, Q FIXED(14,2);
```

DATA ITEM	DECLARED ATTRIBUTES	DEFAULT ATTRIBUTES	BYTES
A	FIXED	DECIMAL Precision(5,0)	3
B	Dimension (5,2,3) FIXED	DECIMAL Precision(5,0)	90
I	FIXED STATIC	DECIMAL Precision(5,0)	3
Q	FIXED Precision(14,2)	DECIMAL	8
		TOTAL	104

Figure 35. Example of Decimal Fixed Data

Binary Float

Default precision: 21 bits
Maximum precision: 53 bits
Storage requirements:

1. Descriptor
   2 bytes (if required)
2. Data
   Hexadecimal floating-point form (see the SRL publication IBM System/360, Principles of Operation, Order No. A22-6821).
   a. Short floating-point form (4 bytes) used for a precision of less than 22 bits.
   b. Long floating-point form (8 bytes) used for a precision of greater than 21 bits.

Figure 36 shows the storage requirements for the binary float data declared in the following sample statement:

```
DECLARE A BINARY, B BINARY(29), C(2,5)
  BINARY(16), D FLOAT BINARY(50);
```

DATA ITEM	DECLARED ATTRIBUTES	DEFAULT ATTRIBUTES	BYTES
A	BINARY	FLOAT Precision (21)	4
B	BINARY Precision (29)	FLOAT	8
C	Dimension (2,5) BINARY Precision (16)	FLOAT	40
D	BINARY FLOAT Precision (50)	None	8
		TOTAL	60

Figure 36. Example of Binary Float Data

Decimal Float

Default precision: 6 decimal digits
Maximum precision: 16 decimal digits
Storage requirements:

1. Descriptor
   2 bytes (if required)
2. Data
   a. Short form (4 bytes) used for less than 7 decimal digits.
   b. Long form (8 bytes) used for more than 6 decimal digits.

Figure 37 shows the storage requirements for the decimal float data declared in the following sample statement:

```
DECLARE A(5,3), B FLOAT(8),
  C DECIMAL(14), D;
```

DATA ITEM	DECLARED ATTRIBUTES	DEFAULT ATTRIBUTES	BYTES
A	Dimension (5,3)	DECIMAL FLOAT Precision (6)	60
B	FLOAT Precision (8)	DECIMAL	8
C	DECIMAL Precision (14)	FLOAT	8
D	None	DECIMAL FLOAT Precision (6)	4
		TOTAL	80

Figure 37. Example of Decimal Float Data

NUMERIC (PICTURE-SPECIFIED) DATA

Default precision: not applicable
Maximum length: after resolution of all replications, the picture-specified numeric field must not be greater than 32 characters. The number of possible picture-specified digit positions depends on whether the number is numeric fixed (15 digits) or numeric float (16 digits).
Storage requirements:

1. Descriptor
   a. Fixed-point data -- one byte for each picture character plus 8 to 20 bytes, with an average of 12 additional bytes (if required).
   b. Floating-point data -- one byte for each picture character plus 20 to 44 bytes, with an average of 24 additional bytes (if required).
   c. Numeric sterling data -- one byte for each picture character plus 4 bytes (if required).

2. Data
   One byte for each picture character except for M, V, K, and G.

Figure 38 shows the storage requirements for the numeric data declared in the following sample statement:

DECLARE A PICTURE '$99.99', B PICTURE '(8)9V(4)9', C PICTURE '.99K+99', D PICTURE 'ZZ99B9(2)B.9,99';

DATA ITEM	BEFORE REPLICATION RESOLUTION	AFTER REPLICATION RESOLUTION	BYTES
A	$99.99	Same	6
B	(8)9V(4)9	99999999V9999	12
C	.99K+99	Same	6
D	ZZ99B9(2)B.9,99	ZZ99B9BB.9,99	13
		TOTAL	37

Figure 38. Example of Numeric Data

STRING DATA

Character-String Data

Default precision: not applicable
Minimum length: 1 character
Maximum length: 255 characters
Storage requirements:

1. Descriptor
   2 bytes (if required)
2. Data
   1 byte per character

Figure 39 shows the storage requirements for the character-string data declared in the following sample statement:

DECLARE A(5) CHARACTER(20), B CHARACTER (111);

DATA ITEM	DECLARED ATTRIBUTES	BYTES
A	Dimension (5) CHARACTER (20)	100
B	CHARACTER (111)	111
	TOTAL	211

Figure 39. Example of Character-String Data

Bit-String Data

Default precision: not applicable
Minimum length: 1 bit
Maximum length: 64 bits
Storage requirements:

1. Descriptor
   2 bytes (if required)
2. Data
   1 byte for each group of 8 bits or part thereof. Packed format is not permitted.

Figure 40 shows the storage requirements for the bit-string data declared in the following sample statement:

DECLARE A BIT(12), B (11,7,2) BIT (1);

DATA ITEM	DECLARED ATTRIBUTES	BYTES
A	BIT (12)	2
B	Dimension (11,7,2) BIT (1)	154
	TOTAL	156

Figure 40. Example of Bit-String Data

Picture-Specified Character-String Data

Default precision: not applicable
Minimum length: 1 character
Maximum length: 255 characters
Storage requirements:

1. Descriptor
   2 bytes (if required)
2. Data
   1 byte per character

Figure 41 shows the storage requirements for the picture-specified character-string data declared in the following sample statement:

```
DECLARE A PICTURE '(105)X', B
  CHARACTER(105);
```

DATA ITEM	DECLARED ATTRIBUTES	BYTES
A	PICTURE '(105)X'	105
B	CHARACTER (105)	105
	TOTAL	210

Figure 41. Example of Both Character-String and Picture-Specified Character-String Data

LABEL DATA

Label Variables

Default precision: not applicable
Maximum precision: not applicable
Storage requirements: 8 bytes

Label Constants

Default precision: not applicable
Maximum precision: not applicable
Storage requirements: 8 bytes for each occurrence of the label in an assignment statement or in a GO TO statement referring to a label that is not contained in the block containing the GO TO statement. Label constants in R format items require 4 bytes. All other label constants do not require storage.

Figure 42 shows the storage requirements for the label data declared in the following sample statement:

```
DECLARE A LABEL, B(7) LABEL;
```

DATA ITEM	DECLARED ATTRIBUTES	BYTES
A	LABEL	8
B	Dimension (7) LABEL	56
	TOTAL	64

Figure 42. Example of Label Data

POINTER VARIABLES

Default precision: not applicable
Maximum precision: not applicable
Storage requirements: 4 bytes

Figure 43 shows the storage requirements for the pointer variable declared in the following sample statement:

```
DECLARE P PCINTER, A BASED (P) FLOAT;
```

DATA ITEM	DECLARED ATTRIBUTES	DEFAULT ATTRIBUTES	BYTES
P	POINTER	None	4
		TOTAL	4

Figure 43. Example of Pcinter Data

Data Storage Depending on Storage Class

STATIC and AUTOMATIC data require the same amount of storage. No storage is required for BASED data. However, accessing based variables by means of pcinters requires 4 extra bytes per reference compared with the other storage classes.

Storage of External Data

Each distinct EXTERNAL variable, array, or structure requires storage in multiples of 8 bytes, since padding to the next doubleword boundary is required if the length of the EXTERNAL data item is not 8 or a multiple of 8 bytes. Figure 44 shows the storage requirements of the EXTERNAL data declared in the following sample statement:

```
DECLARE (A BIT(2), B(3,2,3) CHARACTER(2),
  C CHARACTER(9), D FLOAT(14), E,
  F PICTURE '$99.99', G FIXED DECIMAL
  (13,2)) EXTERNAL;
```

VARIABLE	BYTE REQUIRED		
	DATA STORAGE	PADDING	TOTAL
A	1	7	8
B	36	4	40
C	9	7	16
D	8	0	8
E	4	4	8
F	6	2	8
G	7	1	8

Figure 44. Example of External Data Storage

Use of Constants in The Source Text

Constants may appear in the source text wherever an expression is permitted. In addition, they may appear as replication factors, upper bounds of a subscript range in the dimension attribute of an array, etc. Appearance and representation of constants in the object program depends entirely on their representation and context in the source program. Only the following three cases are of concern to the programmer:

1. If a constant appears in the source text as an argument in a function or subroutine procedure, its object-time representation is derived directly from the source-program representation. For example, the statement

   ```
   CALL A (1.5, 3.7E-4, 110011B);
   ```

   results in an object-time FIXED DECIMAL representation of the constant 1.5, a FLOAT DECIMAL (short float) representation of the constant 3.7E-4, and a FIXED BINARY representation of the constant 110011B.

   Note: If arguments are written as constants, these constants are transmitted to the called routine in coded form and with the precision derived from the source text representation. The called routine, in turn, assumes a certain internal representation of the argument as specified in the parameter declaration. The user must therefore ensure that base, scale, and precision of both arguments and parameters match. For instance, declaring the first parameter in the above example as FIXED (7,1) might lead to an object-time error because the called program assumes an argument that occupies 4 bytes, whereas the constant 1.5 occupies only 2 bytes.

2. If a constant appears in the source text as the upper bound of an array subscript, the appearance of this constant in the object program depends on how the expression used in this subscript position is employed in the remainder of the source text. At best, no constant appears at object time for any upper bound. In the most unfavorable case, a FIXED BINARY constant appears in the object program for every upper bound in the dimension attribute of the DECLARE statement. Thus,

   ```
   DECLARE A (5, 7, 2), B (9, 11);
   ```

   may result in, at most, five FIXED BINARY constants in the object program. At best, no object-time constant will appear for the five upper bounds in the source text.

3. An object-time constant is derived from each source-text constant of a certain base, scale, and precision. However, base, scale, and precision of the object-time constant depend entirely on the context in which it is used. For example, the statements

   ```
   DECLARE A BINARY;
   A = 1.7;
   ```

   cause the constant 1.7 to be stored in the object program in floating-point form, even though the source-text representation is fixed decimal. This shows that identically represented source-text constants may be converted at compile time into a number of different object-time constants (this does not apply to constants in DO iteration specifications). For instance, the following sample statements

   ```
   DECLARE A FIXED DECIMAL,
      B BINARY, C FIXED BINARY;
         A = 2;
         B = 2;
         C = 2;
   ```

   result in three different object-time representations of the single compile-time constant 2. On the other hand, constants of equal value, base, scale, and precision are stored only once in the object program unless NOOPT has been specified in the PL/I PROCESS card. When in doubt about constants which appear similar, e.g., 1.2E+7 as opposed to 12000000, the programmer should review the question of precision of arithmetic constants in the Subset language publication.

Data Storage Mapping

This section discusses the location of a variable in relation to other variables. The location of data with respect to the entire program is discussed in the section Program Overhead.

Boundary Requirements

In the object program, variables that are not part of a structure are grouped according to certain rules referred to as boundary requirements, which depend on the hardware configuration of the system used. For the System/360, the largest unit of storage is the "double word" (8 bytes), which must always be on a double-word boundary (double-word aligned). That is, the first byte of any double word in storage must be on an address divisible by 8. "Full words" (4 bytes) must be full-word aligned, i.e., the first byte of any full word in storage must be on an address divisible by 4. Bit strings, as another example, must be byte aligned, i.e., they may occur on any byte boundary. If any machine address divisible by 8 is chosen as arbitrary byte 0, the above boundary requirements can be reduced to the following:

- double-word aligned data may appear on any byte 0;
- full-word aligned data may appear on any byte 0, 4, 0, 4, etc.; and
- byte-aligned data may appear on any byte 0, 1, 2, 3, ... 7, 0, etc.

Storage Mapping — Element Data

To minimize padding between element data items, the DOS/TOS PL/I compiler gathers - as far as possible - all element data items that are subject to the same boundary requirements. This is done regardless of the point of declaration within the program.

The following discusses the possibilities of mapping elementary data items not contained in structures or arrays and should be understood as an introduction to the mapping of structures.

Much storage can be saved by economically arranging the individual data types. Consider the following example:

```
A BIT(2), B, C BIT(9), D;
```

The result of left-to-right storage allocation is illustrated in Figure 45.

The total storage requirement in this example is 16 bytes, of which 5 are used for padding.

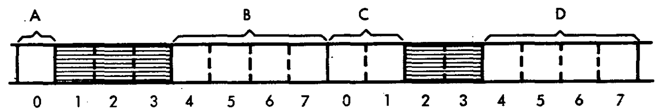


Figure 45. Storage Allocation Example 1

Rearranging the variables as follows:

```
A BIT(2), C BIT(9), B, D;
```

results in a reduction of the total storage requirements to 12 bytes with only one padding byte. Figure 46 illustrates the storage allocation.

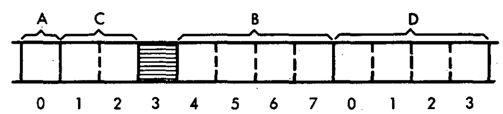


Figure 46. Storage Allocation Example 2

Finally, assume that the variables were rearranged as follows:

```
B, D, A BIT(2), C BIT(9);
```

This is the way in which the DOS/TOS PL/I compiler gathers elementary data items not contained in arrays or structures. The total storage requirements would be reduced to 11 bytes without any padding. The storage allocation is shown in Figure 47.

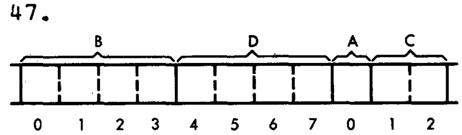


Figure 47. Storage Allocation Example 3

Storage Mapping — Arrays

The storage requirement of an array equals the sum of the requirements of the individual data items contained in the array. Bit-string data items are aligned on byte boundary. Thus, the storage requirement of the array declared in the statement

```
DECLARE A(5,4,3) BIT(9);
```

can be calculated as follows: The number of data items in the array is 5x4x3=60. Due to boundary alignment, each item requires 2 bytes. Total storage requirement: 2x60=120 bytes.

The individual items of an array are stored in major row sequence. For the above example, this means that the items are stored as follows:

```
A(1,1,1)
A(1,1,2)
........
A(5,4,2)
A(5,4,3)
```

Storage Mapping — Structures

To minimize padding, the DOS/TOS PL/I compiler gathers - as far as possible - all elementary data items that are subject to the same boundary requirements.

In the declaration of a structure, such gathering of data is not performed because a structure is regarded as one record, and the programmer might wish to predestine the relative position of every data item within that record, e.g., in a punched card. Thus, the statement below results in the storage allocation illustrated in Figure 48. The total storage requirement is 12 bytes, including 3 padding bytes.

```
DECLARE 1 A ALIGNED, 2 B, 2 C BIT(1), 2 D;
```

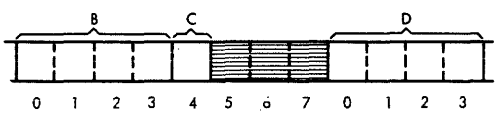


Figure 48. Storage Allocation Example 4

In this example, structure A, which has the unused 3 bytes between C and D, can be thought of as a record without any editing descriptors for the components B, C, and D. It should not be thought of as a bit string because this might lead the programmer to erroneously assume that the first bit of the byte following C is the first bit of D.

Logical Depth Concept

In the following discussion, the term "logical depth" is used to describe the level number of a minor structure or elementary data item relative to the level of the major structure. A minor structure or elementary data item can have a high level number but be at a relatively low logical depth. For instance, in the following sample declaration:

```
DECLARE 1 A,
          15 B,
          15 C,
             95 D,
             95 E,
          15 F,
             31 G,
             31 H,
                45 I,
                45 J,
                   54 K,
                   54 L;
```

structure J has components at logical depth 5 although the level number is 54. The logical depth of these components is greater than that of the components of structure C (3), even though their level number (54) is not as high.

When mapping a major structure, first map all minor structures at greatest logical depth n. Then continue with mapping the minor structures at logical depth n-1. The components that form the minor structure at logical depth n-1 consist of:

1. elementary items at logical depth n, and

2. minor structures at logical depth n, which have already been mapped.

After mapping the minor structures at logical depth n-1, proceed by mapping all minor structures at logical depth n-2. Again, the components that form the minor structure at logical depth n-2 consist of:

1. elementary items at logical depth n-1, and

2. minor structures at logical depth n-1, which have already been mapped and contain the mapped structures at logical depth n.

Continuing this process leads to the major structure, which is at logical depth 1. Mapping of the major structure is done by joining the components at logical depth 2. These components consist of:

1. elementary items logical depth 2, and

2. minor structures at logical depth 2, which have already been mapped and contain the mapped structures at logical depth 3. These, in turn, contain the mapped structures at logical depth 4, etc.

The storage mapping of structures is done according to the set of rules listed below. In the mapping process, a component (or a group of partially mapped components) may be shifted to minimize the padding that may be required between the component and the component to be appended. The opportunity or potential for such shifting depends on the stringency level of the element to be appended. The amount of shifting that is permissible

Variable Type	Stored Internally as	Storage Requirement[1] (in Bytes)	Alignment Requirement	Explanation	Stringency Level
BIT(n)[2]	One byte for each group of 8 bits (or part thereof)	CEIL $\frac{n}{8}$			
CHARACTER(n)	One byte per character	n			
PICTURE	One byte for each PICTURE character except M,V,K,G	Number of PICTURE characters other than M, V, K, and G	Byte	Data may begin on any byte	1
DECIMAL FIXED (w, d)	1/2 byte per digit plus 1/2 byte for sign	CEIL $\frac{w+1}{2}$			
BINARY FIXED (w)	Binary integer				
BINARY FLOAT (w) w < 22	Short floating point	4	Full-word	Data must begin on byte 0 or 4	
DECIMAL FLOAT (w) w < 7					2
LABEL	---	8			
POINTER	---	4	Full-word (right-adjusted)	Data must begin on byte 0 or 4	
BINARY FLOAT (w) 21 < w < 54	Long floating point	8	Double-word	Data must begin on byte 0	3
DECIMAL FLOAT (w) 6 < w < 17					

[1]See Storage of External Data for data declared with attribute EXTERNAL.
[2]Structures containing bit strings must have the attribute ALIGNED because the default attribute (UNALIGNED) is not permitted in the PL/I Subset language.

Figure 49. Summary of Data Alignment Requirements and Stringency Levels

is determined by the alignment requirements of the element(s) to be shifted.

Both the stringency level number and the alignment requirements for the individual data items are shown in Figure 49.

Structure Mapping Rules

1. Locate the first minor structure of the greatest logical depth. (See Figure 50, part A. The declaration shown is used throughout the figure.)

2. Begin the map with the first element of this minor structure. The map begins on byte zero (See Figure 50, part B).

3. Append the next element of the minor structure at the first following byte position where it may be legally placed. This byte position is determined by the alignment requirement of the element to be appended. (See Figure 50, part B.)

4. Owing to the alignment requirement, some unused space (padding) may result between the first and the appended element. The preceding element may then be shifted to the right provided the alignment requirement of that element is still satisfied after the shifting. If no shifting or only a partial shifting is permissible, the padding remains there permanently. (See Figure 50, part B.)

5. The elements so mapped are now permanently joined and may be considered a single element. The alignment requirement of the joined items is that of the item of higher stringency level.

6. Repeat rules 3 and 4 for all remaining elements of the minor structure. (See Figure 50, part B.)

7. Repeat rules 2 through 6 for all minor structures of the same logical depth. Map all minor structures individually. (See Figure 50, part C.)

8. Repeat rules 2 through 7 for the minor structures of the next higher logical depth. Elementary items are appended according to rules 3 and 4. Minor structures are appended beginning at the byte position they had when they were previously mapped. Padding between the two elements, if any, is removed by

   a. shifting the succeeding element as far to the left as its alignment requirement permits, and

   b. shifting the preceding element as far to the right as its alignment requirement permits.

   Any padding that remains after these two shifting processes remains there permanently. (See Figure 50, part D.)

9. Continue this repetitive process until all minor structures are mapped. (See Figure 50, part E.)

10. Map the major structure as if mapping a minor structure. (See Figure 50, part F.)

11. If the shifted structure does not begin on byte zero, pad to the left until byte zero is reached. This is the physical beginning of the structure. However, the name of the major structure still points to the first component of the structure.

12. The first element of the structure must begin on byte zero of the structure being mapped if the structure is a based variable and the pointer variable associated with it appears in the SET clause of a READ or LOCATE statement. In this case, the user must make sure that the structure begins on byte zero. Padding, if required, is best done with a dummy variable of the CHARACTER type. (See Figure 50, part G.)

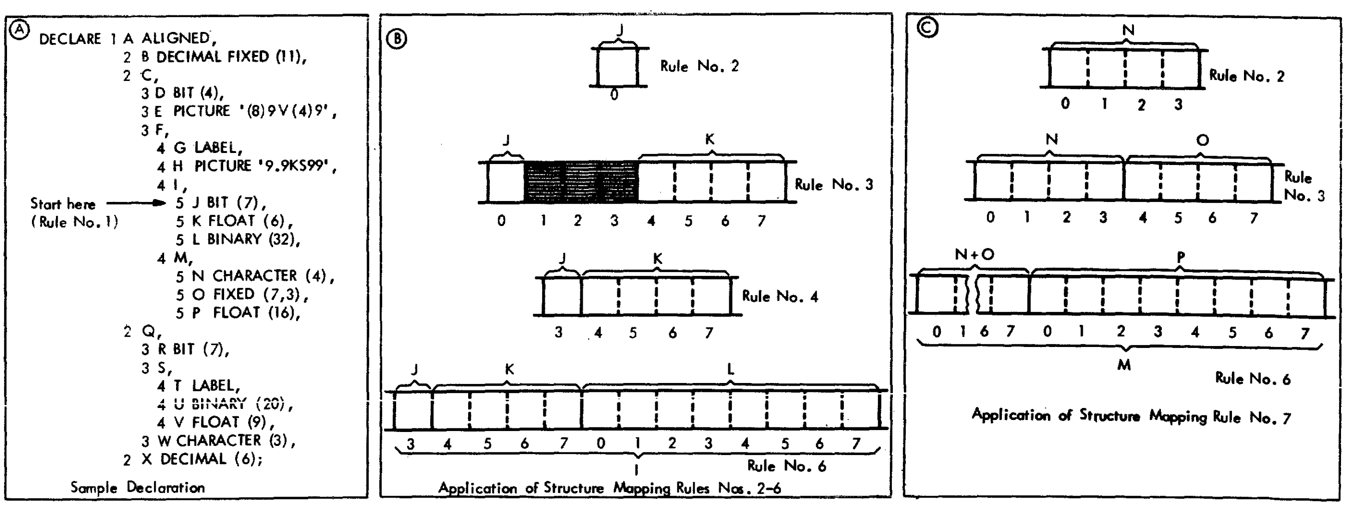


Figure 50. Example of Structure Storage Mapping (Part 1 of 2)

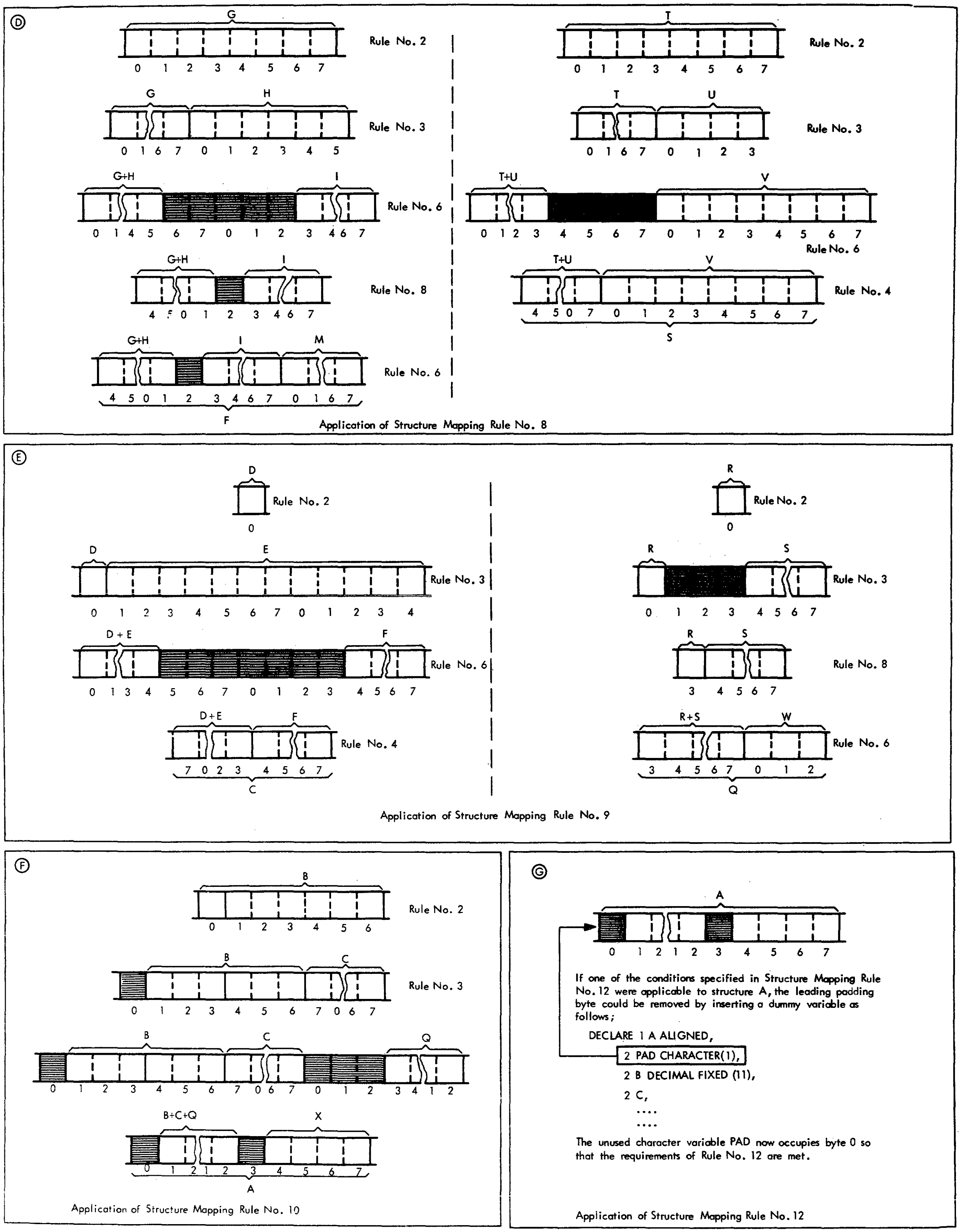


Figure 50. Example of Structure Storage Mapping (Part 2 of 2)

Subroutine Storage Requirements

Three types of subroutines may be required in a program:

1. Conversion subroutines.

2. Subroutines called by built-in function names, pseudo variables, and other implied subroutine calls.

3. Subroutines called by I/O statements.

Conversion Subroutines

Conversion subroutines are required in the object program when certain conversions are implicitly requested in the source text. For example, the statements

```
   DECLARE A FIXED BINARY, B FIXED, C
    BINARY;
        A = B + C;
```

imply that B is to be converted to binary float before being added to C, and that the sum is to be converted to fixed binary before being stored in A.

The 18 conversion subroutines (see Appendix A) can perform every kind of data conversion permitted in the PL/I Subset language. Appendix B lists all possible combinations of data conversion and shows which subroutines are required to perform such conversions. For instance, the conversion from numeric float to numeric fixed decimal requires subroutines 4, 5, and 12. Subroutine 5 converts from numeric float to an internal intermediate form. Subroutine 4 converts from this internal intermediate form to coded fixed decimal. Subroutine 12 converts from coded fixed decimal to numeric fixed decimal.

Note: In some cases it may happen that no subroutine is used at object time although the condition for its inclusion was satisfied. In these cases, the user has overestimated his storage requirements.

Average Conversion Requirements

A system used for scientific purposes will normally use subroutines 1, 2, 7, 8, 9, 10, and possibly 17 and 18, with a total storage requirement of approximately 2K for an average program.

A system used for commercial purposes will most likely use subroutines 11 and 12 with a total storage requirement of approximately .7K for an average program.

Built-in Functions, Pseudo-Variables, And Other Implied Subroutine Calls

Certain built-in functions and pseudo-variables require an object-time subroutine for proper functioning. Some of the built-in functions only allow float arguments. If an argument is not in this form, it is converted before the subroutine is activated.

The source text operator ** is an implicit request for an exponentiation subroutine and, depending on the attributes of the arguments, six different subroutines could be required.

All information required for this type of subroutines is listed in Appendix C.

Depending on the specific arguments, some functions that are marked IL may or may not require subroutines. For instance, a fixed first argument in the FIXED function would not require a subroutine, whereas a float first argument most probably would. However, the subroutine used is a conversion subroutine rather than a function subroutine.

The object-time subroutines are cataloged in the relocatable library. The programmer can find the module name in the entry-points column. If a module has more than one entry point, the module name is written first.

Note: For some mathematical functions, the programmer may be interested in details such as error statistics and algorithms. For such details refer to the SRL publication IBM System/360 Operating System, PL/I Library Computational Subroutines, Order No. GC28-6590. The DOS/TOS PL/I compiler uses the same algorithms as the OS PL/I compiler. Where applicable, the respective internal names of the OS PL/I compiler subroutines are given in parentheses in the rightmost column of Appendix C.

Special Note Regarding Compatibility

Certain built-in functions available in the full PL/I language are not available in the PL/I Subset language. Thus, if the name of a user-written function procedure happens to be the same as that of an unavailable built-in function, the user-written function procedure is called if the program was compiled by means of the DOS/TOS PL/I compiler because the built-in function of that name is not available. However, if this program were compiled by means of the OS PL/I compiler, the built-in function of that name -- which, in this case, is available -- would be called. For example:

```
A:  PROCEDURE;
   .
   .
   X = REAL(Y);
   .
   .
   END;
```

REAL is a function procedure. If this procedure is compiled by means of the OS PL/I compiler, the built-in function REAL is called. Therefore, user-written function procedures should be named in such a manner as to avoid these complications.

Subroutines Called by I/O Statements

Subroutines may be called by I/O source statements for use at object time. The library subroutines that may be called are listed and described in Appendix D.

Care should be taken that any subroutine called by an I/O statement does not itself contain an I/O statement, a PUT/GET STRING statement, or invoke another subroutine containing such a statement.

I/O Storage Requirements

This section provides the information that allows the user to determine the amount of storage required for I/O purposes at object time. Object-time core storage is required

1. as a function of the file declaration itself, and
2. by library subroutines called by I/O statements, such as GET, PUT, etc.

The library subroutines called by I/O statements are listed in Appendix D.

File Declarations

Each file declaration requires four items:

1. Buffers (if required)
2. DTF table
3. Appendage
4. IOCS logic module

The first three items are unique to each declaration. The fourth may be used by various file declarations.

BUFFERS

The number of buffers and the corresponding storage requirements directly derive from the file declaration.

For files other than REGIONAL or INDEXED, the buffer size is equal to the block size specified in the F, V, or U option. Thus, 80 bytes are required with the option F(80). If, in addition, the option BUFFERS(2) is used, the storage requirements for the buffers of this file are doubled. The total storage required for such files equals the sum of the storage requirements for all buffers used for all these files.

<u>Note</u>: No buffer storage is required if the F or U option is used with unbuffered files.

Additional buffer storage (8 * number of extents) is set aside for REGIONAL files.

For REGIONAL(3) files the key length must be added to the buffer length.

The buffer storage requirements for indexed files can be calculated according to the following formulas:

1. <u>Indexed sequential input and update</u>

   unblocked: recsize+2* keylength+10
   blocked: MAX(blocksize,keylength+10+recs ize)

2. <u>Indexed sequential output</u>

   blocksize+keylength+8+recsize
   [+keylength if unblocked]

3. <u>Indexed direct update</u>

   recsize [+keylength if unblocked]
   [+ADDBUFF if specified]
   [+MAX(8+keylength+blocksize,
   8+keylength+10+recsize) if ADDBUFF
   not specified]
   [+INDEXAREA if specified]

4. <u>Indexed direct input</u>

   keylength+MAX(blocksize,10+recs ize)
   [+INDEXAREA if specified]

DTF TABLE

The DTF (Define The File) table is required for each declaration. The function of the DTF table is (together with the appendage) to allow communication between the object program produced from I/O source statements and the DTF program. The DTF program in turn communicates with the operating system for physical device control.

The DTF table has a fixed length for each I/O device type. Figure 51 shows the storage requirements for the individual DTF tables.

The number of DTF tables is equal to the number of files. The total storage required for all DTF tables is, therefore, equal to the sum of their individual storage requirements. Thus, an object program using three printers and five buffered, blocked-record, magnetic tape files would require

3 x 48 + 5 x 112 = 704 bytes of storage

for DTF tables.

A DTFCD table is generated for each card device. Figure 52 shows the PL/I attributes and the corresponding DTFCD parameters.

Declaration Specified by File	Storage Requirements in Bytes		
Card dev. INPUT	56		
Card dev. OUTPUT	48		
2540,OUTPUT	136		
2520,OUTPUT	56		
Printer	48		
Unbuffered magnetic tape	48		
Magnetic tape, other than unbuffered, with the option	INPUT	OUTPUT	UPDATE
F	112	104	-
V	128	120	-
U	112	104	-
Regional (1)*			
with VERIFY	-	256	264
without VERIFY	216	216	216
Regional (3)*			
with VERIFY	-	328	336
without VERIFY	216	288	288
Indexed direct*	300	-	556**
with INDEXAREA*	324	-	580**
Indexed sequential*	284	252	284
Note: 4 x extentnumber must be added to all values given for indexed files			
Consecutive disk*			
Unbuffered	152	152	152
F	136	160	160
V	152	176	192
U	152	168	192
DTFDI	240	240	240
* Not permitted for TOS. ** Add keylength to this value.			

Figure 51. Storage Requirements for DTF Tables

A DTFPR table is generated for each printer. Figure 53 shows the PL/I attributes and the corresponding DTFPR parameters.

A DTFMT table is generated for each magnetic tape drive. Figure 54 shows the PL/I attributes and the corresponding DTFMT parameters.

A DTFSD table is generated for each disk file with the CONSECUTIVE option. Figure 55 shows the PL/I attributes and the corresponding DTFSD parameters.

PL/I ATTRIBUTES	DTFCD PARAMETERS
Blocksize in F option	BLKSIZE
Logical device address in MEDIUM option	DEVADDR
Dev. type in MEDIUM opt.	
2540	DEVICE=2540
1442	DEVICE=1442
2520	DEVICE=2520
2501	DEVICE=2501
Function attribute	
INPUT	TYPEFILE=INPUT EOFADDR
OUTPUT	TYPEFLE=OUTPUT SSELECT=2
F (blocksize)	RECFORM=FIXUNB
BUFFERS option	
BUFFERS(1)	IOAREA1
BUFFERS(2)	IOAREA1 IOAREA2 IOREG=(2)
2540, OUTPUT	CRDERR=RETRY
Control character for RECORD I/O	
CTLASA	CTLCHR=ASA
CTL360	CTLCHR=YES

Figure 52. PL/I Attributes and Corresponding DTFCD Parameters

PL/I ATTRIBUTES	DTFPR PARAMETERS
Blocksize in F option	BLKSIZE
Logical device address in MEDIUM option	DEVADDR
Dev. type in MEDIUM opt.	
1403	DEVICE=1403
1404	DEVICE=1404
1443	DEVICE=1443
1445	DEVICE=1445
F (blocksize)	RECFORM=FIXUNB
BUFFERS Option	
BUFFERS(1)	IOAREA1
BUFFERS(2)	IOAREA1 IOAREA2 IOREG=(2)
USAGE attribute	
STREAM	CTLCHR=ASA
RECORD	PRINTOV=YES
CTLASA	CTLCHR=ASA
CTL360	CTLCHR=YES

Figure 53. PL/I Attributes and Corresponding DTFPR Parameters

PL/I ATTRIBUTES	DTFMT PARAMETERS
Blocksize in F, V, U option	BLKSIZE
Recsize in F option	RECSIZE
Logical device address in MEDIUM option	DEVADDR
F, V, U option F (blocksize) F (blocksize, recsize) V (maxblocksize) U (maxblocksize)	 RECFORM=FIXUNB RECFORM=FIXBLK IOREG=(2) RECFORM=VARBLK IOREG=(2) RECFORM=UNDEF
BUFFERS option BUFFERS(1) BUFFERS(2)	 IOAREA1 IOAREA1 IOAREA2 IOREG=(2)
Function attribute INPUT OUTPUT INPUT) UNBUFFERED OUTPUT	 TYPEFLE=INPUT EOFADDR TYPEFLE=OUTPUT TYPEFLE=WORK EOFADDR
V (maxblocksize) OUTPUT	VARBLD=(3)
INPUT, V and F not UNBUFFERED	WLRERR
BACKWARDS	READ=BACK
LEAVE	REWIND=NORWD
NOLABEL option NOLABEL without NOLABEL	 FILABL=NO FILABL=STD
U option, BACKWARDS	IOREG=(2)
INPUT	ERROPT=Library routine
U, other than UNBUFFERED	RECSIZE=(4)

Figure 54. PL/I Attributes and Corresponding DTFMT Parameters

A DTFDA table is generated for each disk file with the REGIONAL option. Figure 56 shows the PL/I attributes and the corresponding DTFDA parameters.

PL/I ATTRIBUTES	DTFSD PARAMETERS
Blocksize in F, V, U option	BLKSIZE
Recsize in F option	RECSIZE
Device type in MEDIUM option	DEVICE= 2311 2314 2321
F, V, U option F (blocksize) (blocksize, recsize) V (maxblocksize) U (maxblocksize)	 RECFORM=FIXUNB RECFORM=FIXBLK IOREG=(2) RECFORM=VARBLK IOREG=(2) RECFORM=UNDEF
BUFFERS option BUFFERS(1) BUFFERS(2)	 IOAREA1 IOAREA1 IOAREA2 IOREG=(2)
Function attribute INPUT OUTPUT UPDATE INPUT) UNBUFFERED OUTPUT UPDATE UNBUFFERED	 TYPEFLE=INPUT EOFADDR TYPEFLE=OUTPUT TYPEFLE=INPUT UPDATE=YES EOFADDR TYPEFLE=WORK DELETFL=NO EOFADDR EOFADDR
V (maxblocksize) OUTPUT	VARBLD=(3)
VERIFY	VERIFY=YES
- - -	ERROPT=Library routine
INPUT or UPDATE, F and V	WLRERR
U, other than UNBUFFERED	RECSIZE=(4)

Figure 55. PL/I Attributes and Corresponding DTFSD Parameters

DTF APPENDAGE

The DTF appendage, like the DTF table, consists of information derived from the file declaration. It also allows communication between the object program produced from I/O source statements and the DTF program. The length of the appendage is exclusively determined by the presence of a single attribute or option. If the declaration

1. contains the INDEXED option, the appendage length is 40 bytes;
2. contains the REGIONAL option, the appendage length is 56 bytes;
3. contains the BUFFERED, STREAM, or UPDATE attribute, the appendage length is 24 bytes;
4. contains the PRINT attribute or is for SYSLST, the appendage length is 32 bytes;
5. does not apply to one of the file types listed under items 1 through 4, the appendage length is 16 bytes.

The number of appendages is equal to the number of files. The total storage required for appendages is equal to the sum of their individual storage requirements.

PL/I ATTRIBUTES	DTFDA PARAMETERS
Blocksize in F option	BLKSIZE
Device type in MEDIUM option	DEVICE= 2311 2314 2321
F (blocksize)	RECFORM=FIXUNB
BUFFERS(1)	IOAREA1
Function attribute and organization option	
INPUT, REGIONAL(1)	TYPEFLE=INPUT READID=YES
OUTPUT, REGIONAL(1)	TYPEFLE=OUTPUT WRITEID=YES
UPDATE, REGIONAL(1)	TYPEFLE=INPUT READID=YES WRITEID=YES
INPUT, REGIONAL(3)	TYPEFLE=INPUT READKEY=YES KEYARG KEYLEN
OUTPUT, REGIONAL(3)	TYPEFLE=OUTPUT AFTER=YES KEYLEN
UPDATE, REGIONAL(3)	TYPEFLE=INPUT READYKEY=YES WRITEKY=YES KEYARG KEYLEN AFTER=YES
VERIFY	VERIFY=YES
- -	SEEKADDR
- -	ERRBYTE
- -	XTNTXIT=IJKTXRM
- -	CONTROL=YES

Figure 56. PL/I Attributes and Corresponding DTFDA Parameters

PL/I ATTRIBUTES	DTFIS PARAMETERS
INPUT SEQUENTIAL or	TYPEFLE=SEQNTL ICAREAS IOREG=(2) ICROUT=RETRVE KEYARG[1]
INPUT DIRECT	TYPEFLE=RANDOM IOAREAR ICREG=(2) ICROUT=RETRVE KEYARG (separate)
OUTPUT SEQUENTIAL	IOAREAL WORKL (only if blocked) IOROUT=LCAD
UPDATE DIRECT	TYPEFLE=RANDOM IOAREAL[2,4] WORKL[4] ICAREAR[2] IOREG=(2) IOROUT=ADDRTR KEYARG[1,3]
Device type	DEVICE=2311, 2314, or 2321
VERIFY or device type = 2321	VERIFY=YES
F(a) F(a,b)	RECFORM=FIXUNB RECFORM=FIXBLK NRECDS RECSIZE
KEYLENGTH OFLTRACKS INDEXMULTIPLE EXTENTNUMBER KEYLOC INDEXAREA ADDBUFF HIGHINDEX 2311 2314 2321	KEYLEN CYLOFL MSTIND=YES DSKXTNT KEYLOC INDAREA INDSIZE INDSKIP IOSIZE HINDEX=n

[1] Separate for blocked
[2] ICAREAL and IOAREAR may be one and the same area
[3] Same as WORKL if unblccked
[4] ADD separate

Figure 57. PL/I Attributes and Corresponding DTFIS Parameters

A DTFIS table is generated for each disk file with the INDEXED option. Figure 57 shows the PL/I attributes and the corresponding DTFIS parameters.

A DTFDI table is generated for Stream files or buffered Record files if

1. the logical address specifies SYSIPT, SYSLST, or SYSPCH in the MEDIUM option,

2. records are of fixed length and unblocked, and the record size (n) is

   not greater than 80 for SYSIPT
   not greater than 81 for SYSPCH
   not greater than 121 for SYSLST and

3. for output files either

   CTLASA (RECORD OUTPUT files) or
   PRINT (STREAM OUTPUT files)

   is specified.

Figure 58 shows the PL/I attributes and the corresponding DTFDI parameters.

PL/I ATTRIBUTES	DTFDI PARAMETERS
Device address in MEDIUM option	DEVADDR=SYSxxx
BUFFERS(1) BUFFERS(2)	IOAREA1 IOAREA1 IOAREA2 IOREG=(2)
SYSIPT	EOFADDR=name ERROPT=name WLRERR=name
Recsize in F option	RECSIZE=...

Figure 58. PL/I Attributes and Corresponding DTFDI Parameters

IOCS LOGIC MODULE

The IOCS logic module uses the information obtained from the DTF table and the appendage, to communicate between the object program and the DOS/TOS control program. Different IOCS logic modules are used depending on the options and attributes specified in the file declaration. Files having the same options and attributes use the same IOCS logic module. For instance, any number of file declarations, each of which refers to a double-buffered input file using a 2540 card reader, would generate a requirement for one single IOCS logic module only.

The device type is the principal factor in determining which IOCS logic module is to be used. In Figures 59 through 64, the individual modules are therefore grouped according to device types. The storage required for each module is stated in bytes.

Card	One Buffer		Two Buffers	
Files	Input	Output	Input	Output
2540	96	192	128	216
1442	100	74	132	116
2520	96	80	128	124
2501	96	--	128	--

Figure 59. IOCS Logic Modules for Card Reading and Punching Devices

Printer Files			
STREAM		RECORD	
1 Buffer	2 Buffers	1 Buffer	2 Buffers
196	220	118	152

Figure 60. IOCS Logic Modules for Printers

Tape Files	Buffered			Unbuffered
	F	U	V	
Backwards	738	556	--	318
All others	690	564	762	

Figure 61. IOCS Logic Modules for Magnetic Tape Units

If both BACKWARDS and non-BACKWARDS modules are used in the same program, only the BACKWARDS module is included.

Disk Files	Consecutive				Regional	
	Unbuffered	Buffered				
		F	V	U	1	3
Input	682	546	746	618	392	392
Output	682	574	1166	730	392	696
Update	722	910	1255	1062	392	696

Figure 62. IOCS Logic Modules for Disk Units (other than INDEXED Files)

Disk Indexed Files	Input	Output	Update	
			Blocked	Unbl.
Sequential	1086	803	1086	1086
Direct	990	--	2948	2752
with INDEXAREA	1138	--	3162	2966
with ADDBUFF	--	--	3220	2936

Figure 63. IOCS Logic Modules for Disk Units (INDEXED Files)

	BUFFERS(1)	BUFFERS(2)
Input	308	368
Output	643	723

Figure 64. IOCS Logic Module for DTFDI Files

EXAMPLES

The following examples show the storage requirements for buffers, DTF table, appendage, and IOCS logic module.

Example 1

DECLARE PUNCHF FILE OUTPUT ENVIRONMENT (F(80) MEDIUM (SYSPCH, 2540));

```
    Buffers               80 bytes
    DTF table            136 bytes
    Appendage             24 bytes
    IOCS logic module    192 bytes
                         ---------
        Total            432 bytes
```

Example 2

DECLARE PRINTF FILE STREAM OUTPUT PRINT ENVIRONMENT (CONSECUTIVE F(121) BUFFERS (1) MEDIUM (SYSLST, 2400));

```
    Buffers              121 bytes
    DTF table            240 bytes
    Appendage             32 bytes
    IOCS logic module    690 bytes
                         ---------
        Total           1083 bytes
```

Example 3

DECLARE TAPEFF FILE RECORD UNBUFFERED ENVIRONMENT (U(512) MEDIUM (SYS004, 2400) LEAVE NOLABEL);

```
    Buffers                0 bytes
    DTF table             48 bytes
    Appendage             16 bytes
    IOCS logic module    318 bytes
                         ---------
        Total            382 bytes
```

Example 4

DECLARE TAPEBF FILE RECORD BACKWARDS UNBUFFERED ENVIRONMENT (U(512) MEDIUM (SYS004, 2400) LEAVE NOLABEL);

```
    Buffers                0 bytes
    DTF table             48 bytes
    Appendage             16 bytes
    IOCS logic module    318 bytes
                         ---------
        Total            382 bytes
```

Example 5

DECLARE DISK1F FILE STREAM INPUT ENVIRONMENT (F(1739) BUFFERS (2) MEDIUM (SYS001, 2311));

```
    Buffers             3478 bytes
    DTF table            136 bytes
    Appendage             24 bytes
    IOCS logic module    546 bytes
                        ----------
        Total           4184 bytes
```

Example 6

DECLARE DSKF FILE RECORD UPDATE BUFFERED ENVIRONMENT (F(1024, 256) BUFFERS (1) MEDIUM (SYS002, 2311));

```
    Buffers             1024 bytes
    DTF table            160 bytes
    Appendage             24 bytes
    IOCS logic module    910 bytes
                        ----------
        Total           2118 bytes
```

Example 7

DECLARE DSKR3F FILE RECORD OUTPUT DIRECT KEYED ENVIRONMENT (REGIONAL (3) F(800) MEDIUM (SYS003, 2311) KEYLENGTH (9))

```
    Buffers              809 bytes
    8x3 extents (default) 24 bytes
    DTF table            288 bytes
    Appendage             56 bytes
    IOCS logic module    696 bytes
                        ----------
        Total           1873 bytes
```

Example 8

DECLARE DSKR1F FILE RECORD UPDATE DIRECT KEYED ENVIRONMENT (REGIONAL (1) F(600) MEDIUM (SYS004, 2311));

```
    Buffers              600 bytes
    8x3 extents (default) 24 bytes
    DTF table            216 bytes
    Appendage             56 bytes
    IOCS logic module    392 bytes
                        ----------
        Total           1288 bytes
```

Example 9

```
DECLARE TAPERF FILE RECORD INPUT BUFFERED
ENVIRONMENT (V(2048) BUFFERS (2) MEDIUM
(SYS005, 2400));
```

Buffers	4096 bytes
DTF table	128 bytes
Appendage	24 bytes
IOCS logic module	762 bytes
Total	5010 bytes

Example 10:

```
DECLARE INDSQI FILE RECORD INPUT KEYED
ENVIRONMENT ( F(800,80) MEDIUM (SYS011,
2314) INDEXED KEYLENGTH(10) EXTENTNUMBER(
3) INDEXMULTIPLE KEYLOC(15));
```

Buffers	800 bytes
DTF table	296 bytes
Appendage	40 bytes
IOCS logic module	1086 bytes
Total	2222 bytes

Example 11:

```
DECLARE INDDUP FILE RECORD UPDATE DIRECT
KEYED ENVIRONMENT (F(800,80) MEDIUM
(SYS012,2321) INDEXED KEYLENGTH(12) VERIFY
EXTENTNUMBER(2) OFLTRACKS(3) KEYLOC(23)
ADDBUFF(1688));
```

Buffers	1768 bytes
DTF table	576 bytes
Appendage	40 bytes
IOCS logic module	3220 bytes
Total	5604 bytes

Note: If all of the file declarations shown in these examples were to appear in the same program, the total storage requirements would be less than the sum of the individual storage requirements because, in a few cases, different file declarations would use the same IOCS logic module.

System Units

SYSPRINT

The storage required for the DTF table, appendage, and IOCS logic module for SYSPRINT is 416 bytes for TOS and 424 bytes for DOS. If DOS allows a 2311 as SYSLST, 688 bytes are required.

SYSIN

The storage required for the DTF table, appendage, and IOCS logic module is 192 bytes for TCS and 216 bytes for DOS. If DOS allows a 2311 as SYSIPT, 408 bytes are required.

Note: If SYSIN and SYSPRINT are used in one program, the storage required for both is 568 bytes for TOS and 600 for DOS. The storage requirement is 920 bytes for DOS if a 2311, 2314, or 2319 is permitted for SYSIPT or SYSLST.

Program Overhead

Object-program overhead derives from the following two sources:

1. The DOS/TOS Supervisor, the size of which is installation-dependent.

2. The general PL/I overhead area, which exists as a function of the PL/I source text. This area comprises the following four parts:

   a. The static storage area.
   b. The dynamic storage area.
   c. The block prologue.
   d. The PL/I control module.

Static Storage Area

Static storage is required by the seven items listed below. (Note that internal blocks require only the static storage listed under items 5 - 7.)

1. A constant basis of 132 bytes.

2. All variables in any block declared with the attribute STATIC.

3. Constants used in the source text.

4. Four bytes for
   a. each library subroutine explicitly or implicitly used in the source text;
   b. each reference to a procedure that is external to the procedure under construction; and
   c. each distinct data item contained in any block and declared with the attribute EXTERNAL.

5. A communications area of 4 bytes.

6. An entry table with a minimum length of 4 bytes. If the block is a procedure, an additional entry of 4 bytes is made for each ENTRY statement in the block.

7. An entry of 8 bytes is made for the occurrence of each <u>different</u> condition in any ON statement internal to the block.

Since items 1, 5, and 6 are always required, the minimum static storage area required is 140 bytes, even for the most trivial procedure. For example,

```
A:  PROCEDURE OPTIONS (MAIN);
    END;
```

<u>Examples of Calculating Static Storage Requirements</u>

The following procedure:

```
A:  PROCEDURE OPTIONS (MAIN);
    DECLARE B FIXED BINARY STATIC;
    C:  PROCEDURE;
        D:  ENTRY;
        RETURN;
        END;
    E:  BEGIN;
        DECLARE I STATIC;
        I=1101B;
        END;
        F:  ENTRY;
    END;
```

consists of the blocks A, C, and E. The static storage requirements of the individual blocks are discussed in terms of the items 1 through 7 listed above.

<u>Block A</u>

1.	132-byte basis	132 bytes
2.	Two variables with the STATIC attribute	8 bytes
3.	One constant	4 bytes
4.	Communications area	4 bytes
5.	Entry table of 4 bytes minimum plus 4 bytes for entry point F	8 bytes
	TOTAL	156 bytes

<u>Block C</u>

1.	Communications area	4 bytes
2.	Entry table	8 bytes
	TOTAL	12 bytes

<u>Block E</u>

1.	Communications area	4 bytes
2.	Entry table	4 bytes
	TOTAL	8 bytes

Consider another external procedure A that contains no other blocks. It uses 400 bytes of static data storage (variables and constants). It requires five library subroutines explicitly and three library subroutines implicitly.

Three procedures external to A are referred to in procedure A. Six variables are declared with the attribute EXTERNAL. The procedure has seven secondary entry points and contains six ON statements, of which four have differing conditions. External procedure A would require the following static storage:

1.	132-byte basis	132 bytes
2.	STATIC variables	
		400 bytes
3.	Constants	
4.	a. 8 library subroutines	32 bytes
	b. 3 procedures external to A	12 bytes
	c. 6 EXTERNAL variables	24 bytes
5.	Communications area	4 bytes
6.	Entry table	32 bytes
7.	Four ON statements with differing conditions	32 bytes
	TOTAL	668 bytes

Finally, consider a third external procedure W that contains two other procedures, X and Y. Procedure Y contains a BEGIN block Z.

W uses 400 bytes of static data storage, X and Y each use 100, and Z uses 200 bytes. Procedure W requires 3 library subroutines, X requires 2, Y requires 5, and Z requires 13. The library subroutines used in blocks W, X, and Y are all different. The 13 subroutines used by Z comprise 3 that are required by other blocks. No procedure external to W is referred to, and there is no EXTERNAL data. Procedure W has 5 ENTRY statements, X has 2, and Y has 3. There are no ON statements in W, 2 ON statements with identical conditions in X, 3 ON statements with differing conditions in Y, and no ON statement in Z.

The static storage requirements for the individual blocks are as follows:

Block W

1.	132-byte basis	132 bytes
2.	STATIC variables	
		800 bytes
3.	Constants	
4.	A total of 20 library subroutines	80 bytes
5.	Communications area	4 bytes
6.	Entry table	24 bytes
	TOTAL	1040 bytes

Block X

1.	Communications area	4 bytes
2.	Entry table	12 bytes
3.	One ON statement	8 bytes
	TOTAL	24 bytes

Block Y

1.	Communications area	4 bytes
2.	Entry table	16 bytes
3.	Three differing ON conditions	24 bytes
	TOTAL	44 bytes

Block Z

1.	Communications area	4 bytes
2.	Entry table	4 bytes
	TOTAL	8 bytes

The total static storage required by external procedure W thus amounts to

1040 + 24 + 44 + 8 = 1116 bytes.

Dynamic Storage Area

Each blocks has its own dynamic storage area. The dynamic storage area is zero when the block is not active. The length of the dynamic storage area when the block is active is determined by the following five items:

1. Data with the attribute AUTOMATIC, either declared or by default.

2. A communications area of 80 bytes.

3. Four bytes for each different parameter to be transmitted to this block.

4. Working storage area I:
   This area is used to store intermediate results of arithmetic expressions. The length of this area is a function of the complexity of the source text. For a program with arithmetic data only, the average length of this area is approximately 36 bytes. However, if the expressions contain character strings, the length increases with the length of the character strings.

5. Working storage area II:

   This area is used to store expressions contained in DO loops. DO statements may be of either one of the following three forms:

a. DO var=expr-1,expr-2,...,expr-n; For such DO statements, the expressions are developed and stored directly in the variable so that no additional storage is required.

b. DO var=expr-1 TO expr-2; or DO variable=expr-1 BY expr-2;

   16 bytes are required for each DO statement of this form, regardless of the number of iteration specifications in each statement.

c.
```
                   TO      BY
   DO var=expr-1  expr-2  expr-3;
                   BY      TO
```

   24 bytes are required for each DO statement of this form, regardless of the number of iteration specifications in each statement.

The information required to determine which iteration specification is being operated upon is also stored in working storage area II. Each DO statement with more than one iteration specification requires additional bytes to service all iteration specifications. Thus, each DO statement requires zero, 16, or 24 bytes for storing expressions within iteration specifications, plus 8 bytes if there is more than one iteration specification for the DO statement.

Example of Calculating Dynamic Storage Requirements

Assume a procedure consists of the external procedure A, which contains the internal procedures B and C. Internal procedure C contains the BEGIN block D. A and B each have 400 bytes of AUTOMATIC data, C has 200, and D has 100 bytes of AUTOMATIC data. Procedures A, B, and C have only one entry point (their primary entry point), and each procedure has a list of five parameters. Only coded arithmetic data is used. The dynamic storage requirements of the individual blocks are then as follows:

Block A

1.	Data	400 bytes
2.	Communications area	80 bytes
3.	Parameter storage	20 bytes
4.	Working storage area I,	36 bytes
5.	Working storage area II (depends on complexity of DO's)	96 bytes
	TOTAL	632 bytes

Block B

1.	Data	400 bytes
2.	Communications area	80 bytes
3.	Parameter storage	20 bytes
4.	Working storage area I, approx.	36 bytes
5.	Working storage area II, approx.	32 bytes
	TOTAL	568 bytes

Block C

1.	Data	200 bytes
2.	Communications area	80 bytes
3.	Parameter storage	20 bytes
4.	Working storage area I, approx.	36 bytes
	TOTAL	336 bytes

Block D

1.	Data	100 bytes
2.	Communications area	80 bytes
3.	Working storage area I, approx.	36 bytes
4.	Working storage area II, approx.	32 bytes
	TOTAL	248 bytes

The total requirement for dynamic storage at a given moment depends on which blocks are simultaneously active. The total storage required is the sum of the dynamic storage areas for the active blocks. In the above example, this is a minimum of 632 bytes. If all blocks are active simultaneously, the dynamic storage requirements amount to 1784 bytes.

The Block Prologue

The prologue is a set of instructions generated for a PROCEDURE, ENTRY, or BEGIN statement. The generated instructions vary depending on the statement. The minimum prologue is 52 bytes. The maximum is approximately 140 bytes. The minimum prologue is used whenever the block is a BEGIN block. In all other cases, the average is approximately 60 bytes per prologue. A secondary entry point with 12 arguments results in the maximum of 140 bytes.

The PL/I Control Routine

The PL/I control routine is a library subroutine, which is always required in storage for PL/I programs. It is responsible for the interaction of the individual PL/I program components. Some of its functions are listed below:

1. Dynamic storage allocation.
2. Hardware interrupt servicing.
3. Handling of ON conditions.
4. Constructing diagnostic messages.
5. Terminating execution.
6. Transmitting communications information from block to block.
7. Providing library work space.

The PL/I control routine is fixed in length (approximately 1500 bytes) and is present only once in a PL/I program, regardless of the complexity of blocking structures, the number of external procedures, and depth of overlaying.

Note: In the discussion of the program overhead, it was shown where the STATIC and AUTOMATIC data will be. In all further references, the term "overhead" is used for the actual overhead without data and without the DOS/TOS control program.

Source Text And Object Program

After having estimated the storage requirements of (1) data, (2) library subroutines, (3) file declarations, and (4) overhead contained in the program, the user can determine what part of the total storage capacity is left for the remaining part of the program. The remaining part mainly consists of (1) in-line instructions produced directly from the source text and (2) calling sequences to subroutines for those operations that cannot be done in line.

What instructions are produced from the source text can be shown by a simple example.

```
DECLARE A FIXED DECIMAL;
     .
     .
     A = B * C + D;
     .
     .
```

The instructions produced from the assignment statement might be as follows:

- In-line instruction to load B into some register.
- In-line instruction to multiply C (floating-point multiplication) with the contents of this register.
- In-line instruction to add D (floating-point) to the contents of this register.
- Calling sequence(s) to convert the contents of this register to fixed decimal form.
- In-line instruction to store the result in A.

Calling sequences can be avoided in some cases, e.g., in the example shown above by giving A the attributes FLOAT DECIMAL instead of FIXED DECIMAL. To save storage, the user should, therefore, write his programs in such a manner as to avoid unnecessary calling sequences.

The above example shows that a series of instructions is generated for a single PL/I statement. The number of generated instructions depends on the form and complexity of the respective statement. The number of instructions generated for a source-text DO statement, for instance, depends on the complexity of the expressions within an iteration specification, the number of options chosen, and the number of iteration specifications. However, the following average values can be assumed:

1. In a purely scientific environment, the average PL/I source statement generates ten 4-byte instructions.
2. In a purely commercial environment, the average PL/I source statement generates seven 4-byte instructions.
3. These average values are considerably increased by an excessive use of conversions of base or scale and GET and PUT statements in either scientific or commercial environments.
4. Parameters as well as BASED and EXTERNAL data require 4 bytes in addition to the storage requirements of the data item.

Thus, if 5000 bytes are available for the object program, the user may assume that approximately 125 PL/I statements (scientific environment) or 178 PL/I statements (commercial environment) can be accommodated in this area. If the program exceeds this number of statements, the user must either shorten the function of the program or use the overlay feature. (Refer to the section Overlay.)

Note: If listing of source-program statement numbers in case of execution-time errors is requested (by specifying STMT in the PL/I PROCESS card), the additional storage requirements are 4 bytes for each time the statement number appears in the object-program listing.

Problem Analysis Example

A tape system that has a storage capacity of 16K is used for maintaining files. The problem program consists of 3 phases. Phase 1 reads transaction cards (one 80-column card per transaction) and sorts, edits, and writes the contents of these transaction cards on a magnetic tape file. Phase 2 reads the old master file, a transaction card, and writes a new master file record. Both of these operations involve magnetic tapes for old and new master records. An exception report is written, if necessary, on a fourth magnetic tape. Phase 3 takes the exception file and prepares it with appropriate headings.

In the following example, only the storage requirements for phase 2 are examined.

FILE DESCRIPTION

Old Master File: Unblocked, 320-character records of fixed length.

New Master File: Unblocked, 320-character records of fixed length.

Transaction File. Unblocked 80-character records of fixed length.

Exception File: Unblocked 100-character records of fixed length.

DATA ASSUMPTIONS

Due to the requirements of temporary storage, arithmetic statements, etc., 50 variables and constants are used in addition to the data read from and written into files. All data is describable in terms of pictures and character strings; no data is read or written in packed mode.

OTHER ASSUMPTIONS

1. Each file has only one buffer.
2. The data is processed in its respective buffer by use of the READ SET or LOCATE SET statements.
3. The program can be written in one block.
4. The problem does not necessitate inter-phase communication.
5. If conversions from numeric fixed to coded fixed become excessive, the user will convert the data items once and use the coded fixed form for subsequent computations.

Storage Requirements

The storage requirements are as follows:

1. Data
   a. Data read from, or written into, files are accounted for in buffers.

   b. 30 variables (XXXX.XX) 120 bytes
      20 constants (XXX.XX) 60 bytes

   c. Descriptors approximately 150 bytes

   TOTAL approx. 330 bytes

2. Non-I/O Subroutines

   Numbers 11 and 12

   TOTAL 640 bytes

3. File Descriptions

   a. Buffers - 820 bytes

   b. DTF tables - 368 bytes

   c. Appendages - 96 bytes

   d. IOCS logic modules - 690 bytes

   TOTAL 1974 bytes

4. I/O Subroutines

   Number 6

   TOTAL 652 bytes

5. Overhead

   a. Static - approx. 160 bytes

   b. Dynamic - approx. 150 bytes

   c. Prologue - approx. 60 bytes

   d. PL/I control - approx. 1500 bytes

   TOTAL approx. 1870 bytes

6. DOS/TOS Control Program

   approx. 6150 bytes

   GRAND TOTAL approx. 11,616 bytes

This means that approximately 4,770 bytes of storage are available for the actual program, so that the approximate number of PL/I statements that would fit into storage is 160.

After having programmed the problem, the user would determine whether or not he can change the buffering to allow for faster transaction processing. If the data read and/or written are changed into packed form, the buffer requirements are reduced, and the non-I/O subroutines of 640 bytes would not be required. This would allow for approximately 30 additional PL/I statements.

Overlay

If certain parts of an object program are not required in storage throughout its execution and never simultaneously required in storage, the same storage area can be used to store these parts to reduce the overall requirements of the program.

Each part of the program that will reside in storage only for a fraction of the execution time is referred to as an overlay. The MAIN procedure must not be used as an overlay. Each overlay as well as any portion of the program that resides in storage throughout the execution is referred to as a phase. A phase consists of one or more external procedures.

The PL/I subset does not provide direct overlay facilities. However, overlays can be performed by using the library subroutine OVERLAY that provides a link to the operating system which, in turn, loads the actual overlay. (Refer to the SRL publications describing the DOS/TOS control and service programs.) The statement calling the overlay must be coded as follows:

```
[label:] ... CALL OVERLAY
   (character string expression - max.
    length 8)
```

For example, LINK: CALL OVERLAY ('PHASE5');

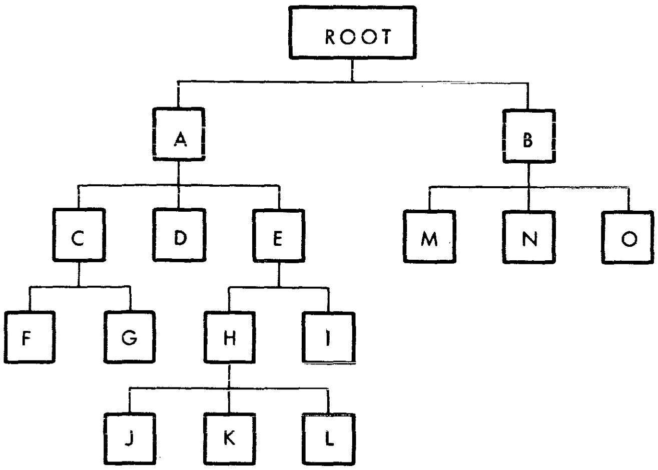
The overlay call activates the OVERLAY subroutine and transmits the name of the phase to be fetched to the control program. The control program locates this phase on the external medium. The phase is then loaded into storage. It must not overlay the fetching procedure. Finally, control is returned to the fetching procedure.

Rules For Using Overlay

The following 17 rules should be observed when using overlay calls:

1. After the phase has been entered in storage, it must be activated by means of a call to the procedure name or any of its entry points.
2. The phase name is independent of the procedure name. It is assigned by means of a PHASE card during processing by the Linkage Editor.
3. A fetching phase (i.e., a phase activating an overlay) may have been fetched into storage by a preceding fetching phase. A series of successive fetching phases is referred to as a tree structure (see Figure 65). The principal fetching phase of a tree structure is referred to as the root. A phase within the tree structure which is not a fetching phase is referred to as a leaf.
4. A fetching phase may fetch any phase lower than itself in the tree structure, provided the fetched phase is on the same branch as the fetching phase.
5. If a phase fetches a phase more than one level below it, an empty space is left in storage for each phase between the fetching and the fetched phase.
6. The root cannot be overlaid. It resides in storage throughout the execution of the problem program.
7. A phase may be activated at any time after it has been fetched, provided it has not been destroyed.
8. Fetching a phase already fetched into storage causes a new copy of that phase to be fetched into storage. All variables of that phase which are in static storage have no known value.
9. Data to be known in more than one phase may be given the EXTERNAL attribute or be transmitted through argument lists of the CALL statement. External names that are to be common to more than one phase below the root level must be declared to be external both in the affected phases and in the root. For larger volumes of data, the use of the EXTERNAL attribute generally requires less storage than argument transmission. Where the argument names change, argument transmission is normally more economical than giving the data the EXTERNAL attribute.
10. External names of procedures to be fetched must be unique (see Figure 65.)
11. A library subroutine is incorporated in every phase in which it is used if
    a. the subroutine is used in a procedure below the root level; and
    b. that subroutine is not in the root. The multiple appearance of

the subroutine can be avoided by incorporating it in the root through the use of an INCLUDE statement during link-editing so that it appears only in the root.



Note: The ROOT phase may fetch any phase, A through O. Phase A may fetch any phase, C through L. Phase B may fetch any phase, M through O. Phase C may fetch phases F and G. Phase E may fetch any phase, H through L. Phase H may fetch phases J through L. Phases D, M, N, O, F, G, I, J, K, and L are leaves.

Figure 65. Schematic Representation of a Tree Structure

Note: Care should be taken if relocatable modules that are not PL/I library subroutines are to be included into more than one phase by the autolink feature. For details, refer to the SRL publications describing the DOS/TOS system control and system service programs.

12. If many phases from different branches of the tree structure activate the same procedure, this procedure may be incorporated in the root in a manner similar to the inclusion of subroutines (see rule 11).

13. If (1) the declaration of a file is made internal to some phase which is not the root, (2) this file is opened in this phase, and (3) the phase is about to be overlaid with a phase from another branch of the tree structure, the user must close this file before it is destroyed. This restriction does not apply if the file is declared both in the root and in a lower phase.

    Note: If the PL/I standard files are used (by a GET or PUT statement) in a phase other than the root, these files must either be used in the root phase, too, or in a phase that will not be further overlaid. Another possibility is to include the corresponding modules in the root by means of the Linkage Editor control statements

```
INCLUDE IJKSYSA (for PUT)
INCLUDE IJKSYSI (for GET)
```

    In all other cases, the standard files cannot be closed, and an error will occur at End-of-Job.

14. If the object-time diagnostic messages are to include the numbers of the source statements causing the errors, STMT must be specified in the PROCESS card for at least the first external procedure contained in the root phase.

15. The time to find and transfer a phase to core storage requires between 200 and 600 msec for DOS, depending on the phase length. A 10K phase, for example, would require approximately 350 msec.

16. The time required to find and transfer a phase to core storage for TOS depends on the physical location of the phase on SYSLNK.

17. Different modules to be included from the relocatable library may be identical except for one or more additional entry points in one of these modules. If the module without the additional entry point(s) is contained in the root phase, calling of the module with the entry point(s) in overlay phases will result in an error during link-editing.

    For instance, the PL/I library routines IJKTSTM and IJKTLCM have the following entries:

Module Name	IJKTSTM	IJKTLCM
Entry Names	IJKTSTM IJKTSTN IJKTSTR	IJKTSTM IJKTSTN IJKTSTR IJKTLCM

    (IJKTSTM is used for stream I/O, IJKTLCM is used for stream I/O with COLUMN or LINE.)

    If IJKTSTM is contained in the root phase, calling of IJKTLCM in an overlay phase will result in an error during link-editing. To avoid such errors, the module containing the additional entry (IJKTLCM in this case) must be included in the root phase by means of an INCLUDE statement.

Overlay Example

Assume that some program consists of one external procedure, which is a single block. Compilation of this procedure on a system with a storage capacity of 16K produces an object program that requires 20K. The storage requirements for the individual parts of the program are as follows:

DOS/TOS control program	- 6K
Overhead	- 2K
Data	- 2K
Subroutines including logical IOCS	- 5K
Object program	- 5K

Actually, the program requires only 19K under the assumption that 1K of data is automatic and 1K is static. However, 20K is required when the data is allocated.

In order to make the object program run on a system with a storage capacity of 16K, it is segmented into 8 phases. The root, which is located behind the DOS/TOS control program, contains the MAIN procedure and the subroutines. Thus, the root plus the DOS/TOS control program may require 11K plus the overhead and program requirement of 2K, i.e., a total of 13K. Since the PL/I control program is in the root phase, the total overhead for the non-root phases is approximately .5K.

This remaining overhead increases slightly because there are now 8 separate blocks, each of which with its own overhead. The allotment of this remaining overhead may result in .25K per block. Due to these changes, the program logic must be slightly changed and extended to allow for the overlaying. This brings the requirement for the object program to about .7K per phase. Since each phase requires less than 1K and the root plus the control program requires 15K, the program will now run on a system with a storage capacity of 16K. The root will fetch the first phase (named PHSE1) and activate it. Control is then returned to the root, and the second phase (named PHSE2) is fetched and activated. This process is repeated until the eighth phase has been executed. This completes the processing of one block of input data, and the process is then repeated. The names of the procedures shown below are A for the root and B1, B2, ..., B8 for the phases.

```
A:PROCEDURE OPTIONS (MAIN);
  DECLARE (data items) EXTERNAL;
  ON ENDFILE (file-name) action;
 BEGIN:  CALL OVERLAY ('PHSE1');
         CALL B1;
         CALL OVERLAY ('PHSE2');
         CALL B2;
         .
         .
         CALL OVERLAY ('PHSE8');
         CALL B8;
         GO TO BEGIN;
         END
   B5:PROCEDURE;
   DECLARE (data items) EXTERNAL;
   .
   .  source text
   .
   RETURN;
   END;
```

For DOS, the additional time required per block of input data when using the overlay feature is approximately 4 seconds. For TOS, the additional time required depends on the number and order of the phases. In the above example, the time increase is about the same for DOS and TOS.

Processing of Overlays by The Linkage Editor

All phases of one program are processed by the Linkage Editor program in one single job step. Therefore, only one // EXEC LNKEDT statement must be given for a multi-phase program. Each phase requires one PHASE statement, which must immediately precede the input for this phase. The ENTRY statement, if used, must be the last statement in the input stream to be written on SYSLNK. A multi-phase program must contain one external procedure with the option MAIN. This external procedure must appear in the physically first phase, i.e., in the root phase.

If programs that contain overlays are to be processed by the Linkage Editor program, a PHASE statement of either one of the following three formats must be used:

1. PHASE phasename,ROOT
   This format must be used for the root phase. It must be the first PHASE statement in the input stream.

2. PHASE phasename,*
   This format of the PHASE statement causes the subsequent phase to be loaded beginning at the next double-word boundary. The use of this statement is recommended for the second phase.

3. PHASE phasename,symbol
   Symbol is either a previously-defined phase name or an entry name appearing in a previous phase (except in the root phase). This format of the PHASE statement causes the next phase to be loaded beginning at the address of the symbol.

The syntax rules for the PHASE statement are as follows:

1. A phase name must be from 5 to 8 characters long.

2. All phase names of a program must be identical in their leftmost four characters.

   Note: Different programs (tree structures) must differ in the first four characters of their phase names in order to avoid incorrect storage allocation.

3. The phase names must be identical to the values of the character-string expressions (except for blanks on the right-hand side) that are used as arguments in the OVERLAY statement.

When link-editing multiphase foreground programs, the ACTION statement with the operand F1 or F2 must be used because, otherwise, the PHASE card for the first phase could not have the ROOT operand. The first three characters of the phase names of a multiphase foreground program should be FGP to have them retrieved faster from the core-image library.

```
| |// JOB MYOVLAY
| |// OPTION LINK
| |   PHASE OVLAY1,ROOT
| |// EXEC PL/I
| |   RT:PROCEDURE OPTIONS (MAIN);
| |   RU:ENTRY
|1|      CALL OVERLAY ('OVLAY2');
| |      .
|2|      CALL OVERLAY ('OVLAY3');
| |      .
|3|      CALL E;
| |      .
| |      END;
| |/*
|4|   INCLUDE JKLM
|5|   PHASE OVLAY2,*
| |   INCLUDE
| |   deck XYZ
| |/*
|6|   PHASE OVLAY3,OVLAY2
|+|   INCLUDE MYPROG
| |// EXEC PL/I
| |   E:PROCEDURE;
| |   .
| |   END;
| |/*
|7|   ENTRY RU
| |// EXEC LNKEDT
|8|// EXEC
| |/&
```

Figure 66. Sample Program to be Processed by the Linkage Editor

Figure 66 shows a sample program to be processed by the Linkage Editor. The numbers at the left-hand margin are not part of the coding; they serve as reference to the explanations only.

Explanation

1 Causes loading of phase OVLAY2.

2 Causes loading of phase OVLAY3.

3 Activates procedure E in phase OVLAY3. It is assumed that phase OVLAY3 has been loaded previously and has not been destroyed, (for example, by reloading phase OVLAY2).

4 The module JKLM that is cataloged in the relocatable library is to be used in OVLAY2 and OVLAY5. Therefore, it is included in the ROOT phase by an INCLUDE statement.

5 This statement causes three actions:

   a. It signals that the input stream of OVLAY1 is terminated.

   b. The modules that are contained in the relocatable library and required for OVLAY1 are retrieved from the library by the autolink feature in order to complete OVLAY1.

   c. Phase OVLAY2 is loaded beginning at the first double-word boundary following the last module of OVLAY1.

6 This statement causes three actions:

   a. It signals that the input stream of OVLAY2 is terminated.

   b. The library modules that are required for phase OVLAY2 and not contained in the ROOT phase (OVLAY1) are retrieved from the library by the autolink feature.

   c. The starting point of OVLAY3 is determined to be the same as that for OVLAY2.

7 This statement causes four actions:

   a. It signals that the input stream for the program is terminated.

   b. The library modules that are required for phase OVLAY3 and not contained in the ROOT phase (OVLAY1) are retrieved from the library by the autolink feature.

   c. RU is determined to be the starting point for the execution of the program.

   d. The starting point of the dynamic storage area is determined to begin on the first double-word boundary following OVLAY2 or OVLAY3, whichever is longer.

8 Fetches OVLAY1 and transfers control to entry point RU. Note that only the ROOT phase is loaded by // EXEC.

+ See PL/I Procedures Contained in the Relocatable Library below.

The structure of the resolved overlay scheme of the above example is shown in Figure 67.

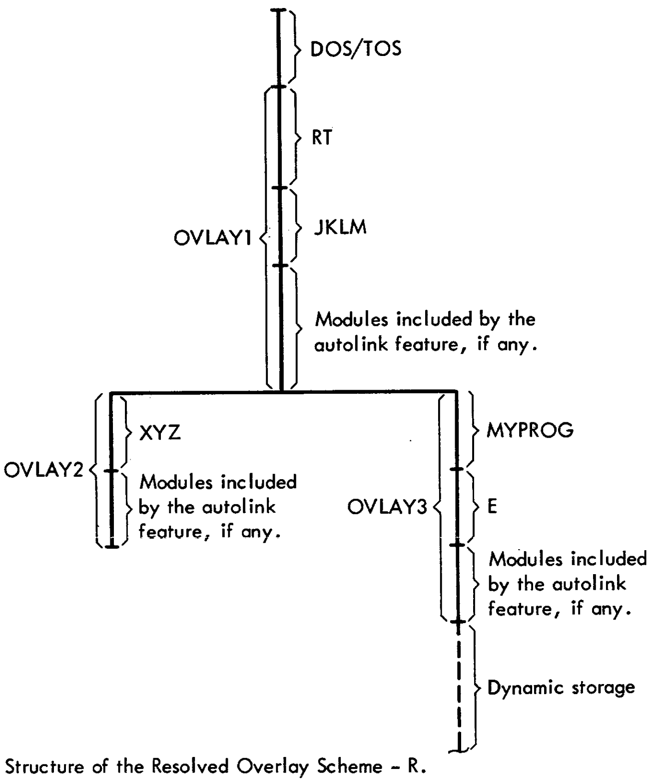


Figure 67. Structure of the Resolved Overlay Scheme

PL/I Procedures Contained in The Relocatable Library

Precompiled PL/I procedures may be incorporated in the relocatable library by using the DOS/TOS MAINT service program. A module is retrieved from the library and incorporated in the object program by the autolink feature when the name of the module is specified for the first time either in a PL/I function reference or in a CALL statement.

No module is retrieved from the library if only secondary entry points are referred to in the calling procedure(s). In this case, a statement of the format

```
INCLUDE module-name
```

is required to include the module in the object program. On the other hand, incorporation by the autolink feature can be suppressed for a specific module by referring only to secondary entries of that module. To obtain the same result as by calling the primary entry point, the programmer may insert a statement of the format

```
ENTRY secondary-entry-name
```

immediately behind the PROCEDURE statement of the external procedure.

Note: Although this description covers most of the applications of the overlay scheme, the reader should study the section covering the Linkage Editor program in the SRL publications that describe the DOS/TOS system control and service programs.

Program Listings

Source Program Listing

All source program cards are listed if the LIST option is in effect. Each card is printed as one line. The source statements are numbered sequentially starting at 1. The statement number is printed in print positions 1 through 6 of the line where the statement begins (right-aligned). In case a line contains more than one statement, only the number of the first statement is printed. However, since the remaining statements are counted, the next line again gives the correct statement number.

Note: If comments or character strings are not correctly opened or closed in the source text, unpredictable diagnostic messages may be produced. Also, the source statement numbering will be erratic.

If the source statement contains any error(s), the statement number is used in the corresponding diagnostic message to clearly identify the statement in error. The diagnostic messages are listed in Appendix F.

Column 1 of PL/I source program cards must always be blank. If column 1 of a source card contains any character, print positions 7 through 20 of the corresponding line in the source program listing -- i.e., the gap between the statement number column and the source statement column plus column 1 of the source card -- are filled with asterisks to indicate this error. Columns 73 through 80 are ignored and may contain any information.

Symbol Table Listing

If the SYM option is specified, all symbols used in PL/I source programs are listed in the symbol table. The format of the symbol table is shown in Figure 68.

The symbol table is listed even if NOSYM was specified in case a declaration contains an error or an external name is too long.

The programmer is advised to examine the symbol table listing after the first compilation of a procedure to detect erroneously declared identifiers and identifiers that may have been incorporated by default rules as the result of mispunching.

The attributes ALIGNED or UNALIGNED, if specified for a major structure, are printed together with the elements of the structure, unless an opposite attribute has been explicitly declared for a particular element.

Print Positions	Contain
1-31	user-defined name
33-36	internal representation
38-39	block number
41	block level number
43-49	one of the attributes ARRAY, STRUCT., ENTRY, or BUILTIN*
51-53	logical structure level*
55-61	one of the attributes ARITHM., STRING, LABEL, PCINTER, FILE, or PICTURE*
63-69	one of the attributes DECIMAL, BINARY, ALIGNED, UNAL., CONST., or VARIAB.*
71-75	one of the attributes FIXED, FLCAT, BIT, CHAR., or STERL*
77-81	the precision or length*
83-88	one of the attributes STATIC, AUTOM., BASED, PARAM., or DEFIN.*
90-92	one of the attributes INT or EXT
* if applicable	

Figure 68. Format of the Symbol Table Listing

Any error detected during compilation in the declaration of the symbols is identified in the symbol table. In this case, only the source program symbol, one of the messages listed in Figure 69, three asterisks, and the code pertaining to the message appear in the respective line of the listing.

Message 12 appears with the first comparand only. Comparison starts with the innermost block and proceeds either on the same nesting level according to the block sequence of the program, or to the block with the next higher nesting level.

Example:

```
OUT:  PROCEDURE;
      DECLARE E BINARY EXTERNAL;
      IN:  PROCEDURE;
           DECLARE E DECIMAL EXTERNAL;
           END IN;
      END OUT;
```

The message appears with the E in procedure IN.

Code	Message Text
01	SYNTACTICAL DECLARE ERROR.
02	CONFLICTING ATTRIBUTES.
03	PRECISION IS MISSING OR WRONG.
04	BASE VARIABLE ITSELF IS DEFINED OR BASED.
05	BASE OR POINTER INCORRECT.
06	ATTRIBUTES OF SECONDARY ENTRY CON-FLICT WITH THOSE OF PRIMARY ENTRY.
07	MULTI-DECLARED IDENTIFIER.
08	ENTRY RETURNS VALUE WITH CONFLICTING ATTRIBUTES.
09	INVALID STRUCTURE. (Any invalid element in a structure may invalid-ate the entire structure).
0A	ARRAY TOO LONG.
0B	STRUCTURE TOO LONG.
0C	POINTER IN BASED STRUCTURE.
0D	TOO MANY ARRAYS.
0E	INVALID PICTURE.
0F	STRUCTURE LEVEL DEEPER THAN EIGHT
10	NAME EXCEEDS 31 CHARACTERS IN LENGTH.
11	EXTERNAL NAME EXCEEDS 8 CHARACTERS IN LENGTH.
12	MULTIPLE DECLARATION OF EXTERNAL NAME INCONSISTENT.

Figure 69. Error Codes Used in the Symbol Table Listing

Cross-Reference Listing

If XREF is specified either in the OPTION statement or in the PL/I PROCESS statement a cross-reference listing will be provided containing the external names in alphabetic order as well as the internal names and the statement numbers of those statements in which the names appear. References to identifiers in DECLARE statements or to incorrectly declared identifiers are not printed.

Offset Table Listing

The offset table listing is produced if the SYM option is specified in the OPTION statement. The information is printed in four columns in hexadecimal notation.

Internal Name. A variable or constant is listed in the offset table if (1) it is declared in the source text and (2) it appears either in the automatic or static storage area, and (3) has a fixed offset relative to the beginning of the respective storage area.

Offset. This column gives the offset of the data item relative to the beginning of the automatic or static storage area for the corresponding block.

Type. This column indicates whether the data item is contained in static or in automatic storage.

Module Offset. This column gives the offset of the data item relative to the beginning of the module in which it appears. (Since the addresses in automatic storage are dynamically assigned, no offset relative to the beginning of the module can be given for automatic data.) The absolute address of the data item contained in static storage can be determined by adding the load address of the module (to be found in the Linkage Editor storage map) to the value given here.

External Symbol Table Listing

The external symbol table is produced if the SYM option is specified in the OPTION statement. It contains the following information:

```
column 1: SYMBOL - the external symbol
column 2: TYPE   - either SD, LD, or ER
column 3: ESID   - ESID number of control
                   section that is referred
                   to (for SD and ER)
column 4: ADDR   - begin address (for SD
                   and LD)
column 5: LENGTH - end address (for SD
                   only)
column 6: ESID   - ESID number of control
                   section that is referred
                   to (for LD)
```

Block Table Listing

The block table listing is produced if the SYM option is specified in the OPTION statement. The block table gives the number of the program block and the size of the corresponding DSA in hexadecimal notation.

Object Code Listing

The object code generated for a PL/I source program is listed following the offset table. The following should be noted:

1. All addresses and operands are printed in hexadecimal notation.

2. Length specifications in SS instructions are printed modulo 256 if one length is specified and modulo 16 if two lengths are specified.

3. Operands of the form X'nnn'(b) represent generated variables or constants. nnn is the displacement and b is the base register.

4. Operands of the form N'nnn', where nnn is greater than or equal to 100, represent internal names of declared items. (These can also be found in the symbol table.)

5. Operands of the form N'nnn', where nnn is less than 100, represent internal names of PL/I library subroutines.

6. Labels of the form L'nnn' represent internal names of declared or generated labels. (Only declared labels can be found in the symbol table.)

7. Operands of the form N'nnn' that appear in the instructions BC, BAL, or BCT represent internal names of either declared or generated labels.

8. A 'constant' of the form X'' has the same function as the assembler instruction EQU *.

9. An instruction of the form

   ```
   L'nnn' DC A(N'nnn')
   ```

   does not represent an address constant of itself. L'nnn', in this case, is the label of the constant, whereas A(N'nnn') refers to an entry point of that internal name in the program. For example, in the instruction

   ```
   L'0104' DC A(N'0104')
   ```

   L'0104' is the label of the constant defined by the DC. A(N'0104') refers to an entry point in the program that has the internal name.

10. If a statement is preceded by more than one label, all labels are equated to the one directly preceding the statement. For the statement:

    ```
    A: B: C: X = Y;
    ```

    the following code would be generated:

    ```
    L'   '  EQU  *     (for A)
    L'   '  EQU  *     (for B)
    L'   '  MVC  ....
    ```

11. The number of the source statement for which the object code is generated is printed at the end of the specific part of the object text. The statement number may appear more than once if the respective source statement was broken down into logical parts during compilation.

Statement Offset Listing

If LISTO is specified in the PROCESS card the statement numbers and the relative location of the end of each statement within the object module is printed. LISTO overrides LISTX, i.e., if LISTO and LISTX are specified, the LISTX option is ignored because the object code listing and the statement offset listing cannot be printed together.

Compile-Time Diagnostic Messages

Errors caused by non-observance of language rules or restrictions in the source text are detected by the compiler. A diagnostic message is printed for each detected error following the source listing. For a statement containing one or more errors, several diagnostic messages may be printed.

The individual diagnostic error messages are listed in Appendix F.

Object-Time Diagnostic Messages

Errors that occur during execution of PL/I programs cause the printing of an object-time diagnostic message. For the format of this message and for an explanation of the message codes that may be included in the message, refer to Appendix G.

Appendix A. Conversion Subroutines

No. and intern. name	Function	Reason for Inclusion in Object Program	Size(in Bytes)
1 IJKVECM	Converts input data from F or E notation to an internal intermediate form	F or E format has appeared in an input statement	404
2 IJKVCEM	Converts data from an internal intermediate form to F or E format in preparation for output	F or E format has appeared in an output statement	1024
3 IJKVPCM	Converts data in storage in coded fixed decimal form to an internal intermediate form	Coded fixed decimal expression appears in an output list or Coded fixed decimal data requires conversion to floating scale or binary base	68
4 IJKVCPM	Converts data from an internal intermediate form to coded fixed decimal form	Coded fixed decimal variable appears in an input list or Whenever a conversion to coded fixed decimal is required	214
5 IJKVFCM IJKVNPM	Converts data stored in numeric float form to an internal intermediate form	A numeric float variable appears in an arithmetic expression or in an output list	492
6 IJKVCFM IJKVPNM	Converts data in an internal intermediate form to internal numeric float	Numeric float variable appears in an input list or appears on the left side of an assignment symbol	680
7 IJKVBCM	Converts data in storage in fixed binary form to an internal intermediate form	Integer binary fixed expression appears in an output list	60
8 IJKVCBM	Converts data in an internal intermediate form to fixed binary form	Binary fixed variable appears in an input list	238
9 IJKVTCM	Converts data from coded floating point form (short or long word) to an internal intermediate form	Coded float expression or non-integer binary expression[1] appears in an output list or Coded float or non-integer fixed binary expression is assigned to a numeric decimal variable or a coded fixed decimal variable	320[2]
10 IJKVCTM	Converts data from an internal intermediate form to coded floating form (short or long)	Coded float variable appears in an input list or Conversion to coded float is required from either numeric data or coded fixed decimal	392[2]

No. and intern. name	Function	Reason for Inclusion in Object Program	Size(in Bytes)
11[4] IJKVNPM	Converts data from numeric fixed form to coded fixed decimal form[3]	Numeric fixed decimal number is used in an arithmetic expression or in an output list	368
12[5] IJKVPNM	Converts data from coded fixed decimal form to numeric fixed decimal form[3]	Numeric fixed decimal number appears on the left of an assignment symbol or in an input list	316
13 IJKVRPM	Converts from numeric fixed sterling to coded fixed decimal	Numeric sterling field is used in an arithmetic expression or in an output list	796
14 IJKVPRM	Converts from coded fixed decimal to numeric fixed sterling	Numeric sterling number appears on the left of an assignment symbol or in an input list	1252
15 IJKVGIM	Converts character string to bit string	Conversion to bit string from character string form is required -	254
16 IJKVIGM	Converts bit string to character string	Conversion to character string from bit string is required or a bit-string expression appears in an output list	148
17 IJKVBTM	Converts fixed binary data to coded float	Conversion from binary fixed to coded float is required	132
18 IJKVTBM	Converts coded float data to fixed binary	Conversion from coded float to fixed binary is required	228

[1]The only way for a non-integer fixed binary number to appear is if the result of a division of one fixed binary integer by another results in a non-integral value or by use of any of the built-in functions PRECISION, BINARY, or FIXED.

[2]Also requires a table of 128 bytes. Subroutines 9 and 10 require this table. If both subroutines appear, the table is in storage only once.

[3]Any picture data represented by [9...][V][9...][T] is converted to and from coded fixed decimal by a single in-line instruction and requires no subroutines.

[4]Subroutine 11 is a subset of subroutine 5. If 5 is present, 11 is not.

[5]Subroutine 12 is a subset of subroutine 6. If 6 is present, 12 is not.

Appendix B. Possible Combinations of Data Conversions

FROM \ TO		FORMAT ITEMS F	FORMAT ITEMS E	FORMAT ITEMS A	FORMAT ITEMS B	CODED FIXED DECIMAL	NUMERIC FIXED DECIMAL	CODED FLOAT	NUMERIC FLOAT	NUMERIC STERLING	FIXED BINARY	CHARACTER STRING	BIT STRING	LABEL	POINTER
FORMAT ITEMS	F	NP	NP	NP	NP	1,4	1,4, 12	1,10	1,6	1,4, 14	1,8	NP	NP	NP	NP
FORMAT ITEMS	E	NP	NP	NP	NP	1,4	1,4, 12	1,10	1,6	1,4, 14	1,8	NP	NP	NP	NP
FORMAT ITEMS	A	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	X	NP	NP	NP
FORMAT ITEMS	B	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	15	NP	NP
CODED FIXED DECIMAL		2,3	2,3	NP	NP	IL	12	3,10	3,6	14	IL	NP	IL	NP	NP
NUMERIC FIXED DECIMAL		2,3, 11	2,3, 11	NP	NP	11	11,12	3,10, 11	3,6, 11	11,14	11	IL	11	NP	NP
CODED FLOAT		2,9	2,9	NP	NP	4,9	4,9, 12	IL	6,9	4,9, 14	18	NP	18	NP	NP
NUMERIC FLOAT		2,5	2,5	NP	NP	4,5	4,5, 12	5,10	5,6	4,5, 14	5,8	IL	5,8	NP	NP
NUMERIC STERLING		2,3 13	2,3 13	NP	NP	13	12,13	3,10, 13	3,6, 13	13,14	13	IL	13	NP	NP
FIXED BINARY		2,7	2,7	NP	NP	IL	12	17	6,7	14	IL	NP	IL	NP	NP
CHARACTER STRING		NP	NP	X	NP	NP	NP	NP	NP	NP	NP	IL	15	NP	NP
BIT STRING		NP	NP	NP	16	IL	12	IL	6,7	14	IL	16	IL	NP	NP
LABEL		NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	IL	NP
POINTER		NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	IL

Legend: NP - Not permitted.

IL - Done directly in-line; no subroutine required.

X - Contained as part of edit-directed I/O package to be discussed in I/O chapter.

The numbers indicate the applicable conversion subroutines listed in Appendix A.

Appendix C. Built-In Functions, Pseudo-Variables, And Other Implied Subroutine Calls

No	Name	Argument(s)	Internal Name(s)	Size in Bytes	Restrictions and Additional Information
19	REPEAT	bit string	IJKRBKA IJKRBKB	292	Result must not exceed max. string length
		character string	IJKRGKM	84	
20	INDEX	bit string	IJKRBIM	292	
21		character string	IJKRGIM	108	
22	BOOL		IJKRBBM	424	
23	SUBSTR	character string	in-line	--	
		bit string	IJKVIIM	180	
24	UNSPEC	bit string	in-line	--	Argument must not exceed 8 bytes
26	DATE		IJKSDTM	58	
27	STRING		in-line	--	
28	ROUND	fixed binary fixed decimal	IJKRUBM in-line	148 --	
		float	in-line	--	
29	MAX/MIN	all fixed binary	IJKRMBX IJKRMBN	278	Argument with differing data attributes causes some of the data to be converted to one of the four permissible types. The choice depends on the element of the highest stringency level.
30		all fixed decimal	IJKRMPX IJKRMPN	386	
31		all short float	IJKRMSX IJKRMSN	132	
32		all long float	IJKRMLX IJKRMLN	172	
33	SIGN		in-line	--	
34	TRUNC	fixed binary	IJKRWBM	356	In-line code for TRUNC of fixed decimal data. IJKRWPM is used only for FLOOR and CEIL.
		fixed decimal	IJKRWPM	580	
		short float	IJKRWSM	236	
		long float	IJKRWLM	244	
35	FLOOR	Contained in TRUNC. Entry points are IJKRT..			
36	CEIL	Contained in TRUNC. Entry points are IJKRV..			

No	Name	Argument(s)	Internal Name(s)	Size in Bytes	Restrictions and Additional Information
37	MOD	fixed binary	IJKRSBM	200	
		fixed decimal	IJKRSPM	265	
		short float	IJKRSSM	184	
		long float	IJKRSLM	192	
38	PRECISION		in-line	--	
39	HIGH		in-line	--	
40	LOW		in-line	--	
41	FIXED				Attributes of arguments must permit conversion specified by built-in function name. No subroutine is called if argument is already in requested form. Appropriate subroutines 1-18 are used. Choice depends on attributes of argument and built-in function name. (See Appendix A.
42	FLOAT				
43	BINARY				
44	DECIMAL				
45	BIT				
46	CHAR				
47	SUM		in-line	--	
48	PROD		in-line	--	
49	ALL		in-line	--	
50	ANY		in-line	--	
51	ABS		in-line	--	
52	expr.1**expr.2	expr.1 fixed binary expr.2 integer constant	IJKREBM	92	$\|Result\| \le 2^{31}-1$ (IHEXIB)
53		expr.1 fixed decimal, expr.2 integer constant	IJKREPM	140	$\|Result\| \le 10^{15}-1$ (IHEXID)
54		expr.1 short float, expr.2 fixed binary with scale factor 0	IJKRESM	144	$\|Result\| \le 7.2x10^{75}$ (IHEXIS)
55		expr.1 long float, expr.2 fixed binary with scale factor 0	IJKRELM	152	$\|Result\| \le 7.2x10^{75}$ (IHEXIL)
56		expr.1 short float	IJKRXSA	152 (60,62)*	Expr.1 > 0; expr.2 not integer constant or fixed binary; $\|result\| \le 7.2x10^{75}$ (IHEXXS)
57		expr.1 long float	IJKRXLM	168 (61,63)*	Expr.1 > 0; expr.2 not integer constant or fixed binary; $\|result\| \le 7.2x10^{75}$ (IHEXXL)

No	Name	Argument(s)	Internal Name(s)	Size in Bytes	Restrictions and Additional Information
58	SQRT	short float	IJKQQSM	176	Argument = 0 or $2.4\text{x}10^{-78} \le$ argument $\le 7.2\text{x}10^{75}$ (IHESQS)
59		long float	IJKQQLM	160	Argument = 0 or $2.4\text{x}10^{-78} \le$ argument $\le 7.2\text{x}10^{75}$ (IHESSQL)
60	EXP	short float	IJKQASM	232	Argument $\le$ 174.6 (IHEEXS)
61		long float	IJKQALM	456	Argument $\le$ 174.6 (IHEEXL)
62	LOG/LOG10/LOG2	short float	IJKQLSA IJKQLSB IJKQLSC	272	Argument $\le 7.2\text{x}10^{75}$ (IHELNS)
63		long float	IJKQLLA IJKQLLB IJKQLLC	384	Argument $\le 7.2\text{x}10^{75}$ (IHELNL)
64	SIN/COS/ SIND/COSD	short float	IJKQSSD IJKQSSB IJKQSSC IJKQSSA	304	$\lvert$Radian Arg$\rvert < 2^{18}$xpi $\lvert$Degree Arg$\rvert < 2^{18}$x180 (IHESNS)
65		long float	IJKQSLD IJKQSLB IJKQSLC IJKQSLA	416	$\lvert$Radian Arg$\rvert < 2^{50}$xpi $\lvert$Degree Arg$\rvert < 2^{50}$x180 (IHESNL)
66	TAN/TAND	short float	IJKQTSB IJKQTSA	280	$\lvert$Radian Arg$\rvert < 2^{18}$xpi $\lvert$Degree Arg$\rvert < 2^{18}$x180 (IHETNS)
67		long float	IJKQTLB IJKQTLA	360	$\lvert$Radian Arg$\rvert < 2^{50}$xpi $\lvert$Degree Arg$\rvert < 2^{50}$x180 (IHETNL)
68	ATAN(X) ATAN(Y,X) ATAND(X) ATAND(Y,X)	short float	IJKQNSD IJKQNSB IJKQNSC IJKQNSA	400	$0 < \lvert X,Y \rvert \le 7.2\text{x}10^{75}$ (IHEATS)
69		long float	IJKQNLD IJKQNLB IJKQNLC IJKQNLA	536	$0 < \lvert X,Y \rvert \le 7.2\text{x}10^{75}$ (IHEATL)
70	SINH/COSH	short float	IJKQCSA IJKQCSB	208 (60)*	$\lvert$Arg$\rvert \le$ 174.6 (IHESHS)
71		long float	IJKQCLA IJKQCLB	288 (61)*	$\lvert$Arg$\rvert \le$ 174.6 (IHESHL)
72	TANH	short float	IJKQDSA	212 (60)*	$\lvert$Arg$\rvert \le 7.2\text{x}10^{75}$ (IHETHS)
73		long float	IJKQDLA	288 (61)*	$\lvert$Arg$\rvert \le 7.2\text{x}10^{75}$ (IHETHL)

No	Name	Argument(s)	Internal Name(s)	Size in Bytes	Restrictions and Additional Information
74	ATANH	short float	IJKQBSA	208 (62)*	|Arg| < 1 (IHEHTS)
75		long float	IJKQBLA	280 (63)*	|Arg| < 1 (IHEHTL)
76	ERF/ERFC	short float	IJKQRSB IJKQRSA	408 (60)*	|Arg| ≤ 7.62×10^{37} (IJEEFL)
77		long float	IJKQRLB IJKQRLA	776 (61)*	|Arg| ≤ 7.62×10^{37} (IHEEFL)
78	ADDR		in-line	--	
79	NULL		in-line	--	
80	ADD		in-line	--	
81	DIVIDE		in-line	--	
82	MULTIPLY		in-line	--	

*The subroutine whose number is given in parentheses is also used by this routine.

BUILT-IN FUNCTIONS CONTAINED IN THE FULL-SET LANGUAGE, BUT NOT IMPLEMENTED IN THE D-LEVEL COMPILER				
ALLOCATION	DATAFIELD	LBOUND	ONCHAR	ONSOURCE
COMPLETION	DIM	LENGTH	ONCODE	POINTER
COMPLEX	EMPTY	LINENO	ONCOUNT	POLY
CONJG	HBOUND	NULLO	ONFILE	PRIORITY
COUNT	IMAG	OFFSET	ONKEY	REAL
			ONLOC	STATUS

Appendix D. I/O Subroutines

Number	Name	Internal Name(s)	Description	Reason for Inclusion in Object Program	Bytes
1	Pagesize	IJKTPSM	Controls number of lines on printed page	The PAGESIZE option appears in an OPEN statement	72
2[1]	Stream Constructor I	IJKTSTM IJKTSTN IJKTSTR	Constructs a logical stream from physical record and vice versa	Always present for files declared with the STREAM attribute	674*
3[1]	Stream ConstructorII	IJKTLCM IJKTSTM IJKTSTN IJKTSTR	Same as Stream Constructor I except that LINE or COLUMN is used	Always present for files with the STREAM attribute, with format list containing LINE or COLUMN, or with PUT statement containing the LINE option	876*
4[2]	Format I	IJKTFDM	Associates a variable with its editing descriptor	GET/PUT FILE EDIT statement appears in source program	480
5[2]	Format II	IJKTGDI IJKTGDO	Same as Format I	GET/PUT STRING EDIT statement appears in source program	414
6	Consecutive Buffered Transmitter	IJKTCBM	Transmits data to/from the buffer from/to a record variable for consec. files	READ/WRITE/LOCATE/REWRITE statement is used for a consecutive buffered file	552*
7	Consecutive Unbuffered Transmitter	IJKTCUM	Transmits data directly from/to an external device directly to/from a record variable	READ/WRITE/REWRITE statement is used for a consecutive unbuffered file	252*
8	Regional Transmitter	IJKTRGM	Transmits data to and from a regional device via a hidden buffer	READ/WRITE statement is used for a regional file	398
9	Regional Extent I	IJKTXRM	Determines extent of regional file at open time and serves as file addressing routine to subroutine 8	A regional file exists for 2311 or 2314	356
10	Regional Extent II	IJKTXRN	Same as 9	A regional file exists for 2321	378
11	Indexed Sequential Transmitter	IJKTSIM	Transmits data to/from indexed data sets in seq. access access	READ/WRITE statement is used for indexed sequential file	652
12	Indexed Direct Transmitter	IJKTDIM	Transmits data to/from indexed data sets in direct access	READ/WRITE statement is used for indexed direct file	540

I/O Subroutines, Part 1 of 2

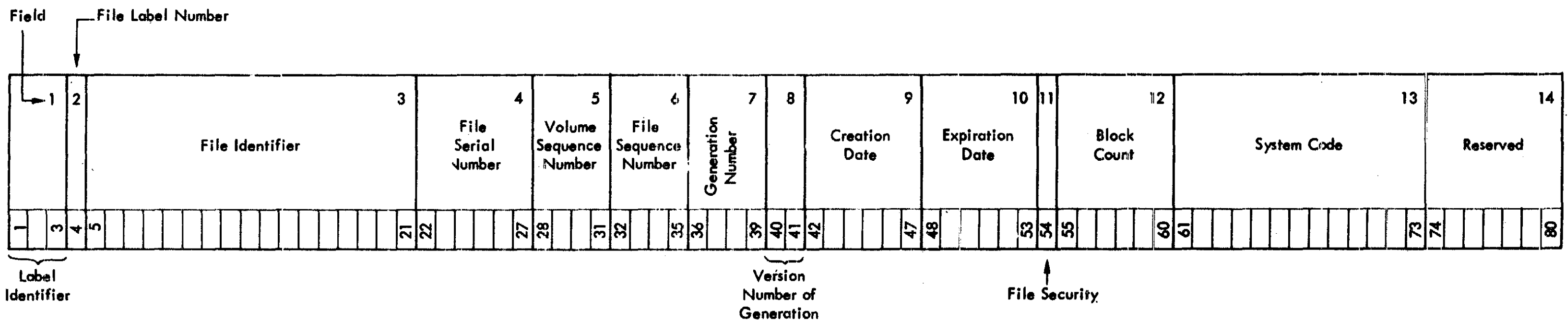
Number	Name	Internal Name(s)	Description	Reason for Inclusion in Object Program	Bytes
13	Display	IJKTDPD IJKTDPR	Handles DISPLAY statement and REPLY option	DISPLAY statement appears in source program	184
14	LIST-I/O	IJKTLIM	Handles list-directed input	GET [FILE/STRING] LIST	1068
15	LIST-I/O	IJKTLOM	Handles list-directed output	PUT [FILE/STRING] LIST	1076

[1]Subroutines 2 and 3 are never both used in any object program.

[2]Requires a 200-byte format scanner. May be required by either subroutine 4 or 5, but is present only once.

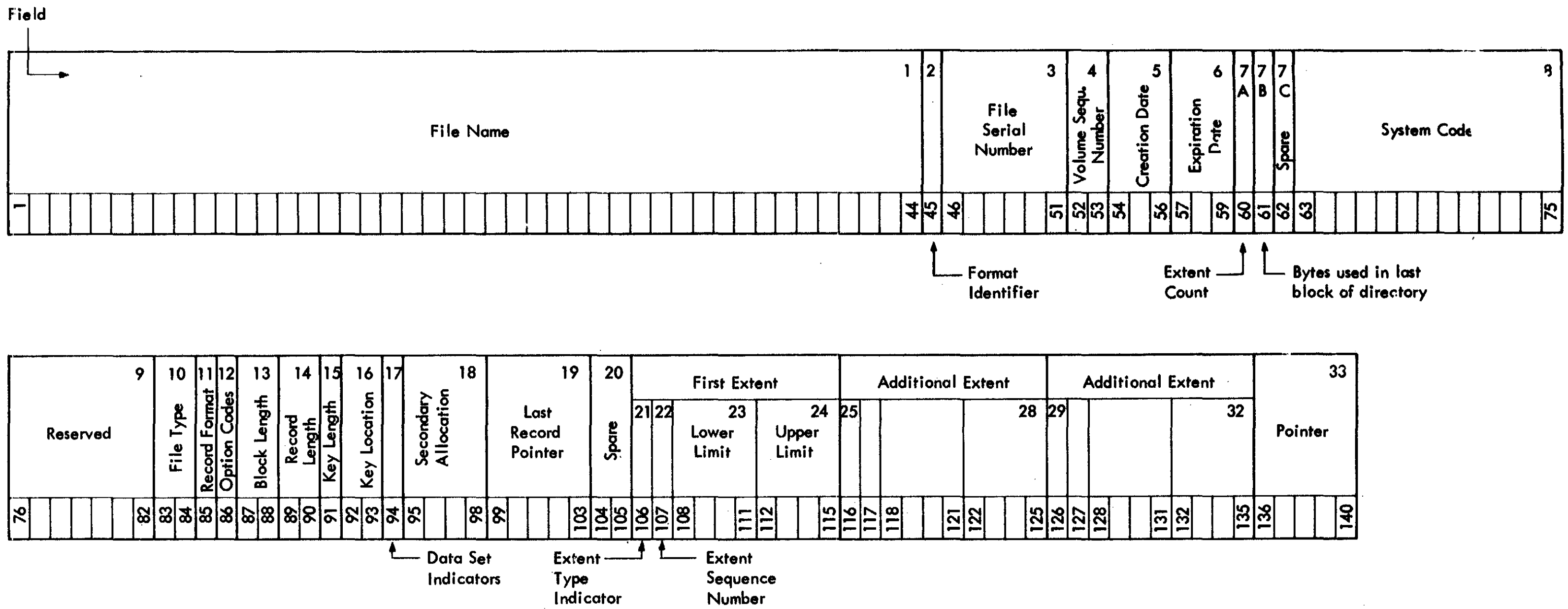
*Requires an additional subroutine of 100 bytes. May be required by several subroutines but is present only once.

I/O Subroutines, Part 2 of 2



The standard tape file label format and contents are as follows:

FIELD	NAME AND LENGTH	DESCRIPTION	FIELD	NAME AND LENGTH	DESCRIPTION
1.	<u>LABEL IDENTIFIER</u> 3 bytes, EBCDIC	Identifies the type of label HDR = Header -- beginning of a data file EOF = End of File -- end of a set of data EOV = End of Volume -- end of the physical reel	9.	<u>CREATION DATE</u> 6 bytes	Indicates the year and the day of the year that the file was created: Position / Code / Meaning 1 / blank / none 2 - 3 / 00 - 99 / Year 4 - 6 / 001 - 366 / Day of Year (e.g., January 31, 1965 would be entered as 65031)
2.	<u>FILE LABEL NUMBER</u>	Always a 1			
3.	<u>FILE IDENTIFIER</u> 17 bytes, EBCDIC	Uniquely identifies the entire file, may contain only printable characters.			
4.	<u>FILE SERIAL NUMBER</u> 6 bytes, EBCDIC	Uniquely identifies a file/volume relationship. This field is identical to the Volume Serial Number in the volume label of the first or only volume of a multi-volume file or a multi-file set. This field will normally be numeric (000001 to 999999) but may contain any six alphameric characters.	10.	<u>EXPIRATION DATE</u> 6 bytes	Indicates the year and the day of the year when the file may become a scratch tape. The format of this field is identical to Field 9. On a multi-file reel, processed sequentially all files are considered to expire on the same day.
5.	<u>VOLUME SEQUENCE NUMBER</u> 4 bytes	Indicates the order of a volume in a given file or multi-file set. The first must be numbered 0001 and subsequent numbers must be in proper numeric sequence.	11.	<u>FILE SECURITY</u> 1 byte	Indicates security status of the file. 0 = no security protection 1 = security protection. Additional identification of the file is required before it can be processed. (Not used by DOS/TOS)
6.	<u>FILE SEQUENCE NUMBER</u> 4 bytes	Assigns numeric sequence to a file within a multi-file set. The first must be numbered 0001.	12.	<u>BLOCK COUNT</u> 6 bytes	Indicates the number of data blocks written on the file from the last header label to the first trailer label exclusive of tape marks. Count does not include checkpoint records. This field is used in Trailer Labels.
7.	<u>GENERATION NUMBER</u> 4 bytes	Uniquely identifies the various editions of the file. May be from 0001 to 9999 in proper numeric sequence.	13.	<u>SYSTEM CODE</u> 13 bytes	Uniquely identifies the programming system.
8.	<u>VERSION NUMBER OF GENERATION</u> 2 bytes	Indicates the version of a generation of a file.	14.	<u>RESERVED</u> 7 bytes	Reserved. Should be recorded as blanks.



Format 1: This format is common to all data files on Direct Access Storage Devices.

FIELD	NAME AND LENGTH	DESCRIPTION
1.	FILE NAME 44 bytes, alphameric EBCDIC	This field serves as the key portion of the file label. Each file must have a unique file name. Duplication of file name will cause retrieval errors. The file name can consist of tree sections: 1. File ID is an alphameric name assigned by the user and identifies the file. Can be 1 - 35 bytes if generation and version numbers are used, or 1 - 44 bytes if they are not used. 2. Generation Number. If used, this field is separated from File ID by a period. It has the format Gnnnn, where G identifies the field as the generation number and nnnn (in decimal) identifies the generation of the file. 3. Version Number of Generation. If used, this section immediately follows the generation number and has the format Vnn, where V identifies the field as the version of generation number and nn (in decimal) identifies the version of generation of the file. Note: The Disk Operation System compares the entire field against the file-ID given in the DLBL statement. The generation and version numbers are treated differently by Operating System /360.
	The remaining fields comprise the DATA portion of the file label:	
2.	FORMAT IDENTIFIER 1 byte, EBCDIC numeric	1 = Format 1
3.	FILE SERIAL NUMBER 6 bytes, alphameric EBCDIC	Uniquely identifies a file/volume relationship. It is identical to the Volume Serial Number of the first or only volume of a multi-volume file.
4.	VOLUME SEQUENCE NUMBER 2 bytes, binary	Indicates the order of a volume relative to the first volume on which the data file resides.
5.	CREATION DATE 3 bytes, discontinuous binary	Indicates the year and the day of the year the file was created. It is of the form YDD, where Y signifies the year (0-99) and DD the day of the year (1-366).
6.	EXPIRATION DATE 3 bytes, discontinuous binary	Indicates the year and the day of the year the file may be deleted. The form of this field is identical to that of Field 5.
7A	EXTENT COUNT	Contains a count of the number of extents for this file on this volume. If user labels are used, the count does not include the user label track. This field is maintained by the Disk Operating System programs.

FIELD	NAME AND LENGTH	DESCRIPTION
7B	BYTES USED IN LAST BLOCK OF DIRECTORY 1 byte, binary	Used by Operating System/360 only for partitioned (library Structure) data sets. Not used by the Disk Operating System.
7C	SPARE 1 byte	Reserved.
8	SYSTEM CODE 13 bytes	Uniquely identifies the programming system. The character codes that can be used in this field are limited to 0-9, A-Z, or blanks.
9	RESERVED 7 bytes	Reserved
10.	FILE TYPE 2 bytes	The contents of this field uniquely identify the type of data file: Hex 4000 = Consecutive organization Hex 2000 = Direct-access organization Hex 8000 = Indexed-sequential organization Hex 0200 = Library organization Hex 0000 = Organization not defined in the file label.
11.	RECORD FORMAT 1 byte	The contents of this field indicate the type of records contained in the file: Bit Position / Content / Meaning 0 and 1 — 01 — Variable length records 10 — Fixed length records 11 — Undefined format 2 — 0 — No track overflow 1 — File is organized using track overflow (Operating System/360 only) 3 — 0 — Unblocked records 1 — Blocked records 4 — 0 — No truncated records 1 — Truncated records in file 5 and 6 — 01 — Control character ASA code 10 — Control character machine code 00 — Control character not stated 7 — 0 — Records have no keys 1 — Records are written with keys.
12.	OPTION CODES 1 byte	Bits within this field are used to indicate various options used in building the file. Bit 0 = If on, indicates data file was created using Write Validity Check. 1-7 = unused

FIELD	NAME AND LENGTH	DESCRIPTION
13.	BLOCK LENGTH 2 bytes, binary	Indicates the block length for fixed length records or maximum block size for variable length blocks.
14.	RECORD LENGTH 2 bytes, binary	Indicates the record length for fixed length records or the maximum record length for variable length records.
15.	KEY LENGTH 1 byte, binary	Indicates the length of the key portion of the data records in the file.
16.	KEY LOCATION 2 bytes, binary	Indicates the high order position of the data record.
17.	DATA SET INDICATORS 1 byte	Bits within this field are used to indicate the following: Bit 0 If on, indicates that this is the last volume on which this file normally resides. This bit is used by the Disk Operating System 1 If on, indicates that the data set described by this file must remain in the same absolute location on the direct access device. 2 If on, indicates that Block Length must always be a multiple of 7 bytes. 3 If on, indicates that this data file ist security protected; a password must be provided in order to access it. 4-7 Spare. Reserved for future use.
18.	SECONDARY ALLOCATION 4 bytes, binary	Indicates the amount of storage to be requested for this data file at End of Extent. This field is used by Operating System/360 only. It is not used by the Disk Operating System routines. The first byte of this field is an indication of the type of allocation request. Hex code C2 (EBCDIC B) blocks (physical records), hex code E3 (EBCDIC T) indicates tracks, and hex code C3 (EBCDIC C) indicates cylinders. The next three bytes of this field is a binary number indicating how many bytes, tracks or cylinders are requested.
19.	LAST RECORD POINTER 5 bytes, discontinuous binary	Points to the last record written in a sequential or partition-organization data set. The format is TTRLL, where TT is the relative address of the track containing the last record, R is the ID of the last record, and LL is the number of bytes remaining on the track following the last record. If the entire field contains binary zeros, the last record pointer does not apply.
20.	SPARE 2 bytes	Reserved
21.	EXTENT TYPE INDICATOR 1 byte	Indicates the type of extent with which the following fields are associated: HEX CODE 00 Next three fields do not indicate any extent. 01 Prime area (Indexed Sequential); or Consecutive area, etc., (i.e., the extent containing the user's data records.) 02 Overflow area of an Indexed Sequential file. 04 Cylinder Index or master index area of an Indexed Sequential file.

FIELD	NAME AND LENGTH	DESCRIPTION
		40 User label track area. 8n Shared cylinder indicator, where n=1, 2, or 4.
22.	EXTENT SEQUENCE NUMBER 1 byte, binary	Indicates the extent sequence in a multi-extent file.
23.	LOWER LIMIT 4 bytes, discontinuous binary	The cylinder and the track address specifying the starting point (lower limit) of this extent component. This field has the format CCHH.
24.	UPPER LIMIT	The cylinder and the track address specifying the ending point (upper limit) of this extent component. This field has the format CCHH.
25-28.	ADDITIONAL EXTENT 10 bytes	These fields have the same format as the fields 21 - 24 above.
29-32.	ADDITIONAL EXTENT 10 bytes	These fields have the same format as the fields 21 - 24 above.
33.	POINTER TO NEXT FILE LABEL WITHIN THIS LABEL SET 5 bytes, discontinuous binary	The address (format CCHHR) of a continuation label if needed to further describe the file. If field 10 indicates Indexed Sequential organization, this field will point to a Format 2 file label within this label set. Otherwise, it points to a Format 3 file label, and then only if the file contains more than three extent segments. This field contains all binary zeros if no additional file label is pointed to.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18		19		Field
DLBL-EXTENT Indicator	Filename	DA/IS Switch	File ID	Format ID	File Serial Number	Volume Sequence Number	Creation Date	Expiration Date	Reserved	Open Code	System Code	Volume Serial Number	EXTENT Type	EXTENT Sequence Number	EXTENT Lower Limit	EXTENT Upper Limit	System Unit Class	System Unit Order	2321 Lower Call	2321 Upper Call	Another EXTENT if DA or ISFMS
1	7	1	44	1	6	2	3	3	2	1	13	6	1	1	4	4	1	1	1	1	Bytes
0	1	8	9	53	54	60	62	65	68	70	71	84	90	91	92	96	100	101	102	103	Displacement

Field	Name	Description
1.	DLBL-EXTENT Indicator	X'80' = Next EXTENT on new pack. X'40' = Last EXTENT X'20' = Bypass EXTENT (SD), or number of EXTENTS (DA or ISFMS). X'10' = New value on same unit. X'08' = EXTENT limits omitted. X'04' = EXTENT converted to DASD address.
2.	Filename	
3.	DA/IS Switch	Same as field 1 except that only bits 4 and 5 are used for DA or ISFMS.
4.	File ID	File identifier including generation and version numbers. If field is missing on DLBL card, filename padded with blanks is inserted.
5.	Format ID	Numeric 1 is inserted.
6.	File Serial Number	Volume serial number from first EXTENT.
7.	Volume Sequence Number	Always initialized to X'0001'.
8.	Creation Date	Initialized with 3 bytes of X'00'.
9.	Expiration Date	If date is in the form YYDDD, it is converted to YDD. If date is in retention period form, 1 to 4 characters, the field is padded with binary zeros.
10.	Reserved	The retention period, if specified, is converted to a 2-byte number and inserted in this field.
11.	Open Code	DLBL type: S = Sequential D = Direct Access C or E = Indexed Sequential File Management System
12.	System Code	Initialized to contain: DOS/360 VER 3. This field is not processed by DOS.
13.	Volume Serial Number	Volume serial number for EXTENT.
14.	EXTENT-Type	Same codes as in Format 1 label: X'00' = Next three fields do not indicate any extent. X'01' = Prime area (ISFMS) or consecutive area, etc., (i.e., the extent containing the user's data records). X'02' = Overflow area of an ISFMS file. X'04' = Cylinder index or master index of an ISFMS file. X'40' = User label track area. X'8n' = Shared cylinder indicator, where n = 1, 2, or 4.
15.	EXTENT Sequence Number	Number of extents as determined by the EXTENT card sequence.
16.	EXTENT Lower Limit	Relative extent converted to the form HHnnT for // DLBL job control statement, or CCHH from // DLAB job control statement.
17.	EXTENT Upper Limit	Same as field 16, but for upper limit.
18.	System Unit Class System Unit Order	Device class and unit numbers.
19.	2321 Lower Call 2321 Upper Call	2321 EXTENT lower and upper limit bin numbers.

Note: For Sequential Disk files, a complete 104-byte block is repeated for each new EXTENT. For Direct Access and ISFMS files, only fields 13 through 18 are repeated for each EXTENT.

Format of DASD Label Information in Label Area Reserved by LABTYP Card

Appendix F. Compile-Time Diagnostic Messages

In the list of diagnostic messages below, the message text is preceded by the message number and the applicable severity code. Where necessary, the messages are followed by an explanation, an example, a description of the action taken by the system, and the response required from the user. Explanation, Example, and System Action are given only when the text of the message is not sufficiently self-explanatory.

When no User Response is stated, the user should assume that he must correct the error in his source program unless the action taken by the system makes it unnecessary for him to do so. However, even when system action successfully corrects an error, the user should remember that, if he subsequently recompiles the same program, he will get the same diagnostic message again unless he has corrected the source error.

The format of the diagnostic messages is as follows:

```
5xdddI  nnnn  C  comment
```

where:

x may be one of the following characters:

- A if compilation must be terminated as a result of a job-control device assignment or option error or

  if an error is detected in the PL/I PROCESS card.

- C if a logical error has been detected in a source statement.

- E if a syntactical error has been detected in a source statement.

- G if the source program is too long or causes a storage overflow.

ddd is the number of the error message. For a message that is also printed on the console, the number is composed of only two digits.

I indicates that the message is of information type and that no operator action is required.

nnnn is the number of the statement in which the error was detected. This number is given only in 5C and 5E type messages.

C is the severity code, which may be one of the following:

W = <u>Warning</u>.
This code indicates that the compiler suspects an error although the program is written in legal PL/I language. The compiler takes no further action.

E = <u>Error</u>.
The program is in error. However, the compiler has taken appropriate corrective action. Execution of the program will be successful if this corrective action was adequate.

S = <u>Severe Error</u>.
The program contains errors which the compiler cannot correct, but which do not prevent the compilation from being continued. Execution of the generated object program will be unsuccessful.

T = <u>Termination</u>.
The source program contains errors causing the compilation to be terminated. Compilation ends after the messages have been printed.

comment
is a compiler-generated explanation of the type of error.

The error messages are printed on the unit assigned to SYSLST if ERRS was specified in the Job Control OPTION statement or in the PL/I PROCESS card. The error list is followed by a message resulting from all detected errors. This message gives the action taken by the compiler.

If errors of the severity T are detected, the message is:

5E01I JOBSTEP PL/I TERMINATED. LINK OPTION RESET.

If no errors of the severity T, but errors of the severity S are detected, the message is:

5E02I LINK OPTION RESET.

Since in the case of severe errors no linkage editing is possible, the // EXEC LNKEDT statement, if any, is flagged as invalid by the Job Control message 1S1nD STATEMENT OUT OF SEQUENCE.

If only errors of the severity W or E are detected, the message is:

5E03I POSSIBLE ERRORS IN SOURCE PROGRAM.

Note: One or more of the following four diagnostic messages may appear after one of the messages 5C003I through 5C030I in order to give additional information. These four messages are printed without message numbers and severity codes.

CHARACTER MARKED BY ASTERISK IS NOT IN 60 CHAR. SET.

Note: This diagnostic message will only be printed for errors in DECLARE statements.

THE PRECEDING ERROR CONCERNS THE VARIABLE NAMED variable name

THE PRECEDING ERROR CONCERNS THE ATTR. FACTORIZATION BEGINNING WITH declare-statement item

'....' REPRESENTS CHARACTER STRING CONSTANT.

Explanation: Illegal use of character-string constant. Since external representation of the character-string constant is not available, the constant is replaced by four periods.

Example: DECLARE N PICTURE A'99999'. Due to the illegal character 'A' the string '99999' is not recognized as numeric picture but as character-string constant. The following messages will be issued where xx represents the statement number:

```
5C019I xx S  INVALID ATTRIBUTE(S) IGNORED..A'....'
             '....'  REPRESENTS CHARACTER STRING CONSTANT.
             THE PRECEDING ERROR CONCERNS THE VARIABLE NAMED N.
```

5A001I T NO COMPILER OUTPUT SPECIFIED IN OPTION STATEMENT.

5A002I T NOT THE SAME OR WRONG MEDIUMTYPES FOR SYS001, SYS002, SYS003.

Explanation: SYS001, SYS002, and SYS003 must be assigned to the same device type, i.e., either to magnetic tape drives, or to 2311 or 2314 DASD extents.

5A003I T PARTITION SIZE TOO SMALL FOR THE 12K VARIANT.

5A004I W ASTERISK IS NOT FOLLOWED BY BLANK. CARD IGNORED.

Explanation: Refers to PL/I PROCESS card. A plus sign is treated as an asterisk.

5A005I W ASTERISK AND BLANK(S) NOT FOLLOWED BY KEYWORD PROCESS.

Explanation: Refers to PL/I PROCESS card. A plus sign is treated as an asterisk.

5A006I W OPTION invalid option UNKNOWN. FOLLOWING TEXT IGNORED.

Explanation: Refers to PL/I PROCESS card.

5A007I W KEYWORD PROCESS NOT FOLLOWED BY BLANK. CARD INGORED.

Explanation: Refers to PL/I PROCESS card.

5A008I W PROCESS LIST TOO LONG. IGNORED IS invalid option

Explanation: Refers to PL/I PROCESS card.

5A009I W PROCESS LIST TOO LONG.

Explanation: Refers to PL/I PROCESS card.

5A010I W COMMA NOT FOLLOWED BY OPTION.

Explanation: Refers to PL/I PROCESS card.

5A011I W OPTION NOT FOLLOWED BY COMMA.

Explanation: Refers to PL/I PROCESS card.

5C003I E LEVELNUMBER OF STRUCTURE ITEM TOO HIGH. ASSUMED TO BE level number

Explanation: Level number must not be higher than 255.

5C004I S NO OPTIONS LIST WITH ENVIRONMENT ATTRIBUTE.

Example: DECLARE FIL FILE ENVIRONMENT INPUT;

5C005I S OPTION LIST NOT CLOSED BY). PARENTHESIS INSERTED AT END OF STATEMENT.

Explanation: This message concerns the ENVIRONMENT and the INITIAL attributes.

Example: DECLARE FIL FILE PRINT ENV(MEDIUM(SYSLST,1403) F(80) ;

5C006I S NO POINTER SPECIFIED FOR BASED ITEM.

Example: DECLARE VAR BASED;

5C007I S ERROR IN SPECIFICATION OF POINTER FOR BASED ITEM. IGNORED IS based data item

Examples: 1. DECLARE B BASED (A,D);
2. DECLARE C BASED (F(I));

5C008I S NO BASE SPECIFIED FOR DEFINED ITEM.

Example: DECLARE X DEFINED;

5C009I S ERROR IN SPECIFICATION OF BASE FOR DEFINED ITEM. IGNORED IS defined data item

5C010I S ERROR IN RETURNS LIST. IGNORED IS invalid elements

Example: DECLARE FUNCT ENTRY RETURNS (7);

5C011I E NO LENGTH SPECIFIED FOR STRING. LENGTH ASSUMED TO BE maximum value

5C012I S ERROR IN STRING LENGTH SPECIFICATION. IGNORED IS invalid element

Example: DECLARE CHARA CHARACTER (STU);

5C013I S ERROR IN PRECISION ATTRIBUTE. IGNORED IS invalid element

Example: DECLARE VAR FIXED (XYZ);

5C014I E VALUE OF ARRAY BOUND MUST NOT BE 0. ASSUMED TO BE 1.

5C015I E VALUE OF ARRAY BOUND TOO HIGH. ASSUMED TO BE maximum value

5C016I S ERROR IN DIMENSION ATTRIBUTE. IGNORED IS invalid element

Example: DELCARE A(7,I,J);

5C017I E RIGHT PARENTHESIS MISSING. CORRESPONDING LEFT ONE IGNORED BEFORE declare_statement item

5C018I S NESTING OF ATTRIBUTE FACTORIZATIONS TOO DEEP. DECLARATIONS FROM NESTING LEVEL 9 ON IGNORED

5C019I E INVALID ATTRIBUTE(S) IGNORED.. invalid attribute [,invalid attribute...]

5C020I E SYNTACTICALLY ILLEGAL CHARACTER(S) IGNORED.. ignored character(s)

Example: DECLARE PP FIXED $;

5C021I S DECL. TOO LONG. ITEMS EXCEEDING LIMIT ARE IGNORED BEGINNING WITH declare statement item

5C022I S NO NAME OR FACTORIZATION FOR LEVELNUMBER.. level number

Example: DECLARE 1 STR, 2, 3 STR1;

5C023I S NO INITIALIZATION WITH INITIAL ATTRIBUTE.

Example: DECLARE VAR INITIAL STATIC;

5C024I S LEVELNUMBER MUST NOT BE 0. ASSUMED TO BE 1.

5C025I E STRINGLENGTH MUST NOT BE 0. ASSUMED TO BE maximum value

5C026I E PRECISION TOO LARGE. SET TO 53.

5C027I E SCALEFACTOR TOO GREAT. ASSUMED TO BE maximum value

5C028I E STRINGLENGTH TOO GREAT. ASSUMED TO BE maximum value

5C029I E LIST OF INITIALIZATIONS NOT CLOSED BY). PARENTHESIS INSERTED AT END OF STATEMENT.

5C030I E NUMBER OF DIGITS IN PRECISION ATTRIBUTE MUST NOT BE 0. DEFAULT VALUE ASSUMED.

5C043I S TOO MANY DIGITS SPECIFIED FOR PICTURE VARIABLE. DEFAULT VALUE IN SYMTAB ASSUMED FOR variable name.

Explanation: The precision of the numeric-picture variable exceeds 15 or 16 digits for decimal fixed or decimal float, respectively. This would lead to an error at execution time. In the symbol-table listing, the default value is printed.

5C044I S SYNTAX ERROR IN INITIALLIST. NO INITIALIZATION OF variable name

Explanation: The INITIAL-list is composed of the following elements: constants, iteration-factors, left and right parentheses, and commas. Error number 44 will be issued if

- the succession of these elements is incorrect, or
- the constants or iteration-factors are incorrect.

Examples of incorrect succession:

1. INITIAL (1,2,)
2. INITIAL (1,(2,3))
3. INITIAL (1,(10) (2,3)4)

Examples of incorrect constants:

1. 1013B
2. 123E
3. 1.21.2L

Examples of incorrect iteration-factors:

1. INITIAL ((-3)0)
2. INITIAL ((0)(1,2))
3. INITIAL (10(1,2))

Moreover, message number 44 will be issued, if there is an illegal character within the INITIAL-list, e.g., INITIAL (2 * 3).

5C045I S NESTING DEPTH EXCEEDS 8. NO INITIALIZATION OF variable name

5C046I S ITERATION FACTOR NOT ALLOWED FOR SCALAR VARIABLE. NO INITIAL. OF variable name

Example: DECLARE Z FIXED INITIAL ((3)4);

5C047I S ITERATION FACTOR GREATER THAN 32K. NO INITIALIZATION OF variable name

5C048I S WRONG DATA TYPE. NO INITIALIZATION OF variable name

Explanation: This error message will be issued, if the type of a constant within the INITIAL-list is not compatible with the type of the variable to be initialized.

Example: DECLARE A DECIMAL FIXED INITIAL ('ABC');

5C049I S INITIAL VALUE IS NOT A LABEL CONST. WITHIN THE SCOPE OF LABEL VARIABLE. NO INITIAL. OF variable name

Explanation: The label constant is internal to a procedure or begin block internal to the block in which the label-variable is declared.

Example:
```
P:  PROCEDURE;
     DECLARE LAB LABEL INITIAL (L2);
      .
      .
      BEGIN;
      .
      .
L2:   END;
END P;
```

5C050I S MORE THAN ONE CONST. FOR SCALAR VARIABLE. NO INITIALIZATION OF variable name

Example: DECLARE Y INITIAL (3E + 01, 33 E + 2);

5C051I W TOO MANY CONSTANTS FOR ARRAY. EXCESS ONES IGNORED FOR array name

5C052I S INITIALLIST TOO LONG. INITIAL ATTRIBUTE IGNORED FOR variable name

5C053I T SYMBOL TABLE ERROR FOR INITIALIZED name

Explanation: This message occurs only if a STATIC structure containing elements with INITIAL attribute is multiply declared.

5C054I E ERROR IN F-OPTION OF FILE filename

5C055I E LEFT PARENTHESIS INSERTED IN FILE filename

5C056I E ILLEGAL ELEMENT IGNORED IN FILE filename

5C057I E RIGHT PARENTHESIS INSERTED IN FILE filename

5C058I S ILLEGAL USAGE OF REGIONAL OPTION. OPTION IGNORED IN FILE filename

5C059I W KEYED ATTRIBUTE INSERTED FOR DIRECT AND/OR INDEXED FILE filename

Explanation: Files with the attributes DIRECT and/or INDEXED must have the attribute KEYED.

5C060I T KEYLENGTH SPECIFICATION MISSING IN FILE filename

Explanation: KEYLENGTH must be specified in files having the KEYED attribute.

5C061I T ERROR IN KEYLENGTH SPECIFICATION FOR FILE filename

5C062I T ERROR IN BLOCKSIZE SPECIFICATION FOR FILE filename

5C063I E ERROR IN BUFFERS OPTION. BUFFERS(1) ASSUMED FOR FILE filename

5C064I E ERROR IN OFLTRACKS SPECIFICATION. OFLTRACKS IGNORED FOR FILE filename

5C065I T ERROR IN MEDIUM OPTION FOR FILE filename

5C066I T INVALID LOGICAL DEVICE NAME IN FILE filename

Example: DECLARE FILE2 FILE INPUT ENVIRONMENT (MEDIUM (SYSRDR, 2540) ...); SYSRDR is an invalid logical unit (choice must be made between SYSIPT and SYSnnn [nnn=001-222]).

5C067I T INVALID DEVICE TYPE SPECIFICATION IN FILE filename

Example: DECLARE FILE3 FILE...ENVIRONMENT (MEDIUM(...,2020)...);

5C068I T DEVICE TYPE OR FUNC. ATTR. CONFLICTS WITH LOG. DEVICE NAME IN FILE filename

Example: DECLARE FILE4 FILE INPUT ENVIRONMENT (MEDIUM (SYS001, 1403)...); Input from Printer 1403 impossible.

5C069I T CONFLICTING ATTRIBUTES AND/OR OPTIONS IN FILE filename

Examples:

1. DECLARE FILES FILE INPUT RECORD UPDATE ...;
2. DECLARE FILE6 FILE OUTPUT ENVIRONMENT (MEDIUM (SYS002, 1403) LEAVE NOLABEL F (81));

5C070I T INPUT, OUTPUT, OR UPDATE ATTRIBUTE MISSING IN FILE filename

5C071I E DIRECT ATTRIBUTE INSERTED FOR REGIONAL FILE filename

5C072I E NOLABEL OPTION INSERTED FOR UNBUFFERED TAPE FILE filename

5C073I T ENVIRONMENT ATTRIBUTE MISSING IN FILE filename

5C074I T MEDIUM OPTION MISSING IN FILE filename

5C075I T BLOCKSIZE NOT DIVISIBLE BY RECORDSIZE IN FILE filename

5C076I W RECORDSIZE OF RECORD NOT DIVISIBLE BY 8 IN FILE filename

Explanation: The record size must be divisible by 8 if blocked records are to be transferred by a READ SET or LOCATE statement.

5C077I W DIVISION OF BLOCKSIZE BY 8 DOES NOT YIELD REMAINDER OF 4 IN FILE filename

Explanation: If the V option is used, the record size of records to be transferred by a READ SET or LOCATE statement must yield a remainder of 4 after division by 8.

5C078I T BLOCKSIZE BEYOND DEVICE DEPENDENT LIMITS IN FILE filename

5C079I T F, U, OR V OPTION MISSING IN FILE filename

5C080I T MORE ERROR(S) IN FILE filename

Explanation: The maximum number of error messages issued for one file declaration is 7. If the file declaration contains more than 7 errors, this message is printed.

5C081I E INVALID ATTRIBUTE IGNORED IN FILE filename

5C082I E PRINT ATTRIBUTE ASSUMED FOR PRINTER AS PHYSICAL DEVICE IN FILE filename

5C084I T ERROR IN EXTENT NUMBER SPECIFICATION FOR FILE filename

5C085I E EXTENTNUMBER SET TO 3 IN DECLARATION OF FILE filename

5C086I S INVALID DEVICE TYPE SPECIFIED FOR HIGHINDEX IN FILE filename

Explanation: Only the device types 2311 and 2314 are allowed. 2321 may be specified if the device type in the corresponding MEDIUM option is also 2321.

System Action: The invalid device type is used for execution.

5C087I S NUMBER OF OFLTRACKS EXCEEDS DEVICE DEPENDENT LIMITS IN FILE filename

Explanation: The number n of overflow tracks specified in the OFLTRACKS option must be within the following limits:

$0 \le n \le 8$ for 2311
$0 \le n \le 18$ for 2314 and 2321

System Action: The value in error is used for execution.

5C088I S KEYLOC BEYOND RECORDSIZE LIMITS IN FILE filename

Explanation: The key location n specified in the KEYLOC option must be within the following limits:

$1 \le n \le$ record size - keylength + 1

The message is issued if $n >$ record size - keylength + 1. If $n = 0$ message 5C092I is printed.

System Action: The value in error is used for execution.

5C089I S ADDBUFF AREA LESS THAN MINIMUM OR GREATER THAN MAXIMUM IN FILE filename

Explanation: The number n of bytes specified in the ADDBUFF option must be within the following limits:

64 + block size + keylength $\le n <$ 32K

System Action: The value in error is used for execution.

5C090I S RECORDSIZE NOT GREATER THAN KEYLENGTH IN FILE filename

Explanation: For blocked records, the record size must be greater than the keylength. If KEYLOC is specified, this also applies for unblocked records.

System Action: The value in error is used for execution.

5C091I W RECORDSIZE EXCEEDS LIMIT FOR OVERFLOW RECORD IN FILE filename

Explanation: The lengths n of the records on the overflow tracks are restricted as follows:

$n \le 3605$ - keylength - 10 bytes for 2311
$n \le 7249$ - keylength - 10 bytes for 2314
$n \le 1984$ - keylength - 10 bytes for 2321

5C092I E INDEXAREA, ADDBUFF, HIGHINDEX OR KEYLOC OPTICN IGNORED IN FILE filename

Explanation: One of the options INDEXAREA, ADDBUFF, HIGHINDEX or KEYLOC is either not followed by a parenthesized specification or is followed by an invalid specification.

5C093I S INDEXAREA LESS THAN MINIMUM OR GREATER THAN MAXIMUM IN FILE filename

Explanation: The number n of bytes specified in the INDEXAREA option must not exceed the following limits:

$3 + (\text{keylength} + 6) \leq n < 32K$

System Action: The value in error is used for execution.

5C094I S MAX. NUMBER OF EXPL. DECLARED VARIABLES PER BLOCK REACHED WITH name

5C095I E MORE THAN ONE INITIAL ATTRIBUTE FOR variable name

System Action: Only the first INITIAL attribute is used.

5C096I E MORE THAN ONE DIMENSION ATTRIBUTE FOR variable name

System Action: Only the first dimension attribute is used.

5C097I E MORE THAN ONE LEVELNUMBER FOR STRUCTURE ITEM structure item name

System Action: The first level number is used.

5C098I E MORE THAN ONE PRECISION OR STRING LENGTH SPECIFIED FOR variable name

System Action: The first precision or length is used.

5C099I E MORE THAN ONE PICTURE ATTRIBUTE SPECIFIED FOR variable name

System Action: Only the first PICTURE attribute is used.

5C100I E MORE THAN ONE BASE OR POINTER SPECIFIED FOR variable name

Example: DECLARE NAME BASED(X) DECIMAL FIXED(7) BASED(Y);

5C101I E STRUCT. NOT START. WITH LEVELNUMBER 1, ASS. TO BE MAJOR STRUCT. NAME IS structure name

Example: DECLARE 2A, 2B, 2C; A is assumed to be the major-structure name.

5C102I E NON-FILETYPE ATTRIBUTES IGNORED FOR FILE filename

5C103I E NON-APPLICABLE ATTRIBUTE(S) IGNORED FOR STRUCTURE structure name

Example: DECLARE 1 A1 FIXED, 2B, 2C;

5C104I S INVALID INITIALIZATION IGNORED FOR variable name

Explanation: Initialization with INITIAL-attribute is conflicting with type or attributes of the variable.

Example: DECLARE E ENTRY INITIAL (SUBPRO);

5C105I E ALIGNMENT PERFORMED FOR BITSTRING bitstring-variable name

Explanation: Bit strings contained in structures and bitstring-arrays are aligned by the D-compiler.

5C106I E MORE THAN 12 DIFF. PARAMETERS TO BE PASSED TO OR FROM BLOCK NUMBER block number

Explanation: Number of parameters is limited to 12.

5C107I E TOO MANY DIGITS SPECIFIED IN PREC. ATTR. DEFAULT VALUE ASSUMED FOR variable name

5C108I E NO SCALE ALLOWED WITH FLOAT OR BIN FIXED. DFLT.PRECIS. ASSUMED FOR variable name

Explanation: A scale factor must not be specified within the precision attribute of BINARY FIXED or FLOAT variables. The whole precision attribute will be ignored and the default precision is assumed for that variable.

Examples:

Illegal:	Assumed:
BINARY FIXED (15,3)	BINARY FIXED (15)
BINARY FIXED (31,0)	BINARY FIXED (15)
DECIMAL FLOAT (3,2)	DECIMAL FLOAT (6)
DECIMAL FLOAT (6,0)	DECIMAL FLOAT (6)
BINARY FLOAT (53,8)	BINARY FLOAT (21)
BINARY FLOAT (53,0)	BINARY FLOAT (21)

5C109I E ENTRY INTO EXT. PROC. IS OF TYPE EXTERNAL. INTERNAL ATTR. IGN. FOR entry name

5C110I T MORE THAN 32K BYTES REQUIRED FOR ARRAY array name

5C111I T POINTER AND/OR BASE IDENT. NOT OR INCORRECTLY DECL. FOR ARRAY array name

Example: DECLARE U, BAS(10) BASED (U); U is not a pointer.

5C113I T REFERENCED VARIABLE OR RELATED BASE/POINTER INCORR. FOR ARRAY array name

Example: DECLARE 1 A, 2 (B(10),C), X(10) DEFINED B;
Defining on elements of structures is not allowed.

5C115I E REPLICATION FACTOR OF ZERO IGNORED IN INITIAL LIST OF variable name

5C116I E STRING CONSTANTS TRUNCATED ON RIGHT IN INITIAL LIST OF variable name

5C117I E EXPONENTS TRUNCATED ON RIGHT IN INITIAL LIST OF variable name

5C118I E FLOAT. CONSTANTS TRUNCATED ON RIGHT IN INITIAL LIST OF variable name

5C119I E ZERO ASSUMED FOR INVALID FLOAT. CONSTANTS IN INITIAL LIST OF variable name

5C120I E MAX. VALUE ASSUMED FOR INVALID FLOAT. CONSTANTS IN INITIAL LIST OF variable name

5C121I E STERLING CONSTANTS TRUNCATED ON RIGHT IN INITIAL LIST OF variable name

5C122I E BINARY FIXED CONSTANTS TRUNCATED ON RIGHT IN INITIAL LIST OF variable name

5C123I E DECIMAL FIXED CONSTANTS TRUNCATED ON RIGHT IN INITIAL LIST OF variable name

5C124I E RESULT OF CONST. CONV. UNDEF. DUE TO SIZE ERROR. CHECK INITIAL LIST OF variable name

5E001I T ILLEGAL CHARACTER IN LABEL PREFIX OR STATEMENT BEGINNING.

Examples: 1. LB1: +B2: LB3: ABC = 50;
Second label is not an identifier.

2. LAB: +BC = 50;
Statement begins with an illegal character.

System Action: The error statement is replaced by a dummy statement.

5E002I T STATEMENT TYPE CANNOT BE IDENTIFIED.

Explanation: An identifier at statement beginning is neither a statement identifier nor followed by the assignment symbol =.

Example: PUTT SKIP EDIT (B) (A); PUTT is not a statement identifier.

System Action: The error statement is replaced by a dummy statement.

5E003I T NESTING OF BLOCKS EXCEEDS 3 LEVELS.

Explanation: Implementation restriction. The depth of nested blocks is restricted to 3 levels. The external proceduce is the first level.

System Action: The flagged statement is replaced by the required number of END statements. The subsequent statements are ignored.

5E004I T NUMBER OF BLOCKS EXCEEDS 63.

Explanation: Implementation restriction. The total number of blocks in an external procedure (including the external procedure) must not exceed 63.

System Action: The flagged statement is replaced by the required number of END statements. The subsequent statements are ignored.

User Response: Reduce number of blocks in one compilation by generating external procedures.

5E005I T ILLEGAL CHARACTER FOUND IN IF-STATEMENT BEFORE 'THEN' IS DETECTED.

Example: IF A = 1; THEN GOTO LAB;

System Action: The incorrect IF statement is replaced by a dummy statement.

5E006I T NO LABEL IS PERMITTED BEFORE AN ELSE-CLAUSE.

Example: IF A = 1 THEN ...; LAB: ELSE B = 5;

5E007I T ELSE FOLLOWED BY INVALID UNIT.

Example: IF A = 1 THEN ...; ELSE 5 = B; where B is a correctly declared variable

System Action: The invalid ELSE clause is replaced by a dummy statement.

5E008I T DO-GROUP NESTING EXCEEDS 12 LEVELS.

Explanation: Implementation restriction. The maximum depth of a nested set of DO statements (including repetitive specifications in GET or PUT statements) is 12.

System Action: The flagged DO statement is replaced by a dummy statement and the following text is ignored.

5E009I T INVALID END STATEMENT.

Explanation: The keyword END is not followed by a semicolon or by the label of its associated PROCEDURE, BEGIN, or DO statement.

Example:
```
LAB:  PROCEDURE;
      .
      .
      END LAS;
```

5E010I T LOGICAL END OF PROGRAM DETECTED BEFORE END OF SOURCE TEXT.

Explanation: Text follows the logical end of the program. The programmer seems to have made an error in matching END statements with PROCEDURE, BEGIN, or DO statements.

System Action: All text following the flagged statement is ignored.

5E011I T MORE THAN ONE LABEL BEFORE PROCEDURE OR ENTRY STATEMENT.

Explanation: PROCEDURE and ENTRY statements must have one and only one label.

5E012I T NO LABEL BEFORE PROC. OR ENTRY STATEMENT. LABEL B INSERTED.

Explanation: PROCEDURE and ENTRY statements must have one and only one label.

System Action: The compiler inserts the label 'B:' before the flagged statement. This may cause further error messages (e.g., multiple declaration).

5E013I T FIRST STMNT NOT PROCEDURE STMNT. FOLLOWING TEXT IGNORED.

System Action: Further error messages may result (e.g., 5E012I and 5E015I).

5E014I T STATEMENT TOO LONG. STATEMENT TRUNCATED.

Explanation: Internal buffer overflow.

User Response: Subdivide statement and recompile.

5E015I T END OF SOURCE MODULE FOUND BEFORE LOGICAL END OF PROGRAM.

Explanation: Problem causing the error may be:

1. Missing final semicolon.

Example:
```
LAB:  PROCEDURE OPTIONS (MAIN);
           .
           .
      END
/*
```

2. Missing END statement(s).

Example:
```
LAB:  PROCEDURE OPTIONS (MAIN);
         .
         .
      DO I = 1 TO 5;
      END;
/*
```

5E016I T RIGHT PARENTHESIS MISSING IN THIS STATEMENT.

Example: A(2,3,1 = 15; where A is declared as a three-dimensional array.

5E017I T END OF SOURCE MODULE FOUND IN PARENTHESIZED LIST.

5E018I T ILLEGAL CHARACTERS IGNORED IN THE PROGRAM

Explanation: Any of the characters listed below in hexadecimal notation are ignored unless they are included in a character string. In a program containing such illegal characters, the compiler flags the first statement that is found to include one or more of these characters.

the illegal characters are:

B8	DA through DF
BA through BF	EA through EF
CA through CF	FA through FE

5E020I T ELEMENT IN PREFIX LIST IS NOT A LEGAL CONDITION NAME.

Explanation: The prefix list contains either an illegal condition name or no condition name at all.

Examples: 1. (   ): LAB: statement;

2. (ZERODIVIDE,+UNDERFLOW): LAB: statement;

3. (ZERODIDIVE, UNDERFLOW): LAB: statement;

System Action: The entire prefix list is ignored.

5E021I T NAME IN PREFIX LIST NOT FOLLOWED BY COMMA OR PARENTHESIS.

Examples: 1. (ZERODIVIDE UNDERFLOW): statement;

2. (OVERFLOW+CONVERSION): statement;

System Action: The entire prefix list is ignored.

5E022I T CONFLICTING CONDITION NAMES IN PREFIX LIST.

Example: (NOCONVERSION,CONVERSION): statement;

System Action: The conflicting names are ignored.

5E023I T COLON AFTER PREFIX LIST IS MISSING.

5E025I T RIGHT PARENTHESIS IS MISSING IN DATA OR FORMAT LIST

5E026I T MAIN PROCEDURE HAS INCORRECT OPTION LIST.

Explanation: For the D-level compiler, the option list of a main procedure is defined as

MAIN[,ONSYSLOG]

It must be enclosed in parentheses immediately followed by a semicolon. The problem causing the error may be:

1. Missing comma or right parenthesis.

   Example: TEST: PROCEDURE OPTIONS (MAIN;

2. Element in list which is not an identifier.

   Example: TEST: PROCEDURE OPTIONS (+AIN);

3. Identifier in list which is neither MAIN nor ONSYSLOG.

   Example: TEST: PROCEDURE OPTIONS (MIAN);

4. Option list not followed by semicolon.

5E027I T NESTING OF ATTRIBUTES EXCEEDS 8 LEVELS.

5E028I T INVALID DEFINED ATTRIBUTE IGNORED.

5E029I T ILLEGAL FACTORIZATION OF FILE ATTRIBUTES.

5E034I T TWO OR MORE IDENTICAL IDENTIFIERS IN ONE PARAMETER LIST.

5E040I T FIRST ARGUMENT IN SUBSTR PSEUDO-VARIABLE IS NOT A STRING VAR.

5E041I T MAJOR OR MINOR STRUCTURE IN IF STATEMENT.

5E042I T ARRAY IN ELEMENT-EXPRESSION OF IF-STATEMENT

5E043I W NO DATA FORMAT ITEM IN FORMAT LIST.

5E044I T ARRAY DECLARATION INCORRECT. FIRST BOUND OF ARRAY IS ZERO.

5E045I T EXTERNAL NAME(S) OF THIS PROGRAM LONGER THAN 8 CHARACTERS.

Explanation: See explanation of message 5E046I.

5E046I E EXTERNAL NAME(S) OF THIS PROGRAM LONGER THAN 6 CHARACTERS.

Explanation: Implementation restriction. The length of external identifiers must not exceed 6 characters. This also applies to names that are external by default such as filenames, names of external procedures, etc. If an identifier has 7 or 8 characters, the object program can still be executed but errors may possibly occur. If the external identifier is longer than 8 characters the compilation is terminated (message 5E045I is issued). The statement in error indicated in this message need not be the statement in which the error is detected.

5E047I T TOO MANY IDENTIFIERS IN THIS STATEMENT.

User Response: Subdivide statement and recompile.

5E049I T POINTER AND/OR BASE IDENTIFIER NOT OR INCORRECTLY DECLARED.

Examples:

```
1. DECLARE G CHARACTER (4);
   DECLARE K CHARACTER (4) BASED (G);
   K = 'TEST';
2. DECLARE P DECIMAL FLOAT POINTER;
   DECLARE A BASED (P);
   A = A+1;
```

In both examples, the third statement is flagged.

5E050I T ATTRIBUTE TABLE OVERFLOW. TOO MANY VARIABLES IN THIS STMNT.

User Response: For each element variable and for each structure element named in the statement, a table entry will be generated.

Subdivide statement and recompile.

5E051I T INVALID DEFINING

Example:

```
DECLARE 1 A,
          2 B DEFINED D,
          2 C;
DECLARE D;
B = 4;
```

The third statement causes the error message.

5E053I T OPERAND IN A GOTO STATEMENT IS NOT A LABEL.

Explanation: The operand in a GOTO statement must always be a label constant or an element label variable.

5E055I S ZERO-REPLICATION FACTOR FOR STRING CONSTANT IGNORED.

5E056I S STRING CONSTANT TOO LONG. TRUNCATED.

Explanation: Implementation restriction. The length of bit-string constants is restricted to 64 bits; the length of character-string constants is restricted to 255 characters.

System Action: Bit strings exceeding 64 bits and character strings exceeding 255 characters are truncated on the right.

5E057I E EXPONENT TOO LONG. TRUNCATED.

Explanation: Implementation restriction. The exponent subfield of a decimal floating point constant is restricted to 2 digits, and that of a binary floating point constant to 3 digits.

System Action: The exponent is truncated on the right.

5E058I E FLOATING-POINT CONSTANT TOO LONG. TRUNCATED.

Explanation: Implementation restriction. The length of binary floating-point data is restricted to 53 bits; the length of decimal floating-point data is restricted to 16 digits.

System Action: Decimal and binary floating-point constants exceeding 16 digits or 53 bits, respectively, are truncated on the right, and the exponents are increased by the number of digits or bits truncated.

5E059I E FLOATING-POINT CONSTANT TOO SMALL. SET TO ZERO.

5E060I E FLOATING-POINT CONSTANT TOO LARGE. MAXIMUM VALUE ASSUMED.

5E061I E STERLING CONSTANT TRUNCATED.

Explanation: The sterling constant is converted to and stored as decimal fixed-point pence. The converted constant must not exceed 15 significant digits.

System Action: The converted decimal fixed-point pence number is truncated on the right.

5E062I E BINARY FIXED-POINT CONSTANT TOO LONG. TRUNCATED.

Explanation: Implementation restriction. The length of binary fixed-point numbers must not exceed 31 bits.

System Action: The constant is truncated on the right.

5E063I E DECIMAL FIXED-POINT CONSTANT TOO LONG. TRUNCATED.

Explanation: Implementation restriction. The length of decimal fixed-point numbers must not exceed 15 digits.

System Action: The constant is truncated on the right.

5E064I E RESULT OF CONSTANT CONVERSION UNDEFINED DUE TO SIZE ERROR.

Explanation: The number of significant digits resulting from the constant conversion is greater than the precision specified for the target.

Example:
```
DECLARE X FIXED BINARY (10);
X = 2.444E5;
```

5E065I T TOO MANY CONSTANTS IN THIS COMPILATION.

Explanation: Internal buffer or constant-counter overflow.

5E067I E INVALID CHARACTER STRING. ONE BLANK ASSUMED.

Explanation: The apostrophe opening the character string is immediately followed by the closing apostrophe.

System Action: The compiler assumes the character string to consist of one blank.

5E068I T QUALIFIED NAME NOT DECLARED.

Example: LAB: PROCEDURE OPTIONS (MAIN);
STRUCT.SUB1 = 50;
END;

5E069I T REFERENCED VARIABLE OR RELATED BASE/POINTER INCORRECT.

Example: DECLARE A CHARACTER (3) BASED (P);
A = 'XYZ';

If P is not declared, the assignment statement causes the error message.

5E070I E A) HAS BEEN INSERTED IN ARGUMENT OR FORMAL PARAMETER LIST.

Example: CALL DYNDUMP (A,B ;

5E071I T UNSPECIFIED SYNTACTICAL ERROR.

Example: DO A = (B TO C BY D WHILE (E)); where A is a variable and B, C, D, E are valid expressions. The parentheses enclosing the specification of the DO statement are illegal.

5E072I T INTERNAL BUFFER OVERFLOW. (PROBABLY TOO MANY PARENTHESES).

User Response: Subdivide statement and recompile.

5E073I E ONE OR MORE) INSERTED TO OBTAIN A VALID EXPRESSION.

Example: DECLARE (A,B,C,D,E) DECIMAL FIXED;
A = B** (C+D*E ;

5E074I E ACTION FOR 5E073I MAY CAUSE ADDITIONAL ERROR MESSAGES.

5E075I T 2ND OPERAND IN DISPLAY STATEMENT INVALID.

Explanation: The second operand of the DISPLAY statement must be a character-string element variable enclosed in parentheses.

5E076I T SHILLING FIELD OF STERLING CONSTANT GREATER THAN 19.

5E077I T ERROR IN PARAMETER, OR SUBSCRIPT, OR ARGUMENT LIST.

5E078I S ILLEGAL FILENAME OR LABEL IDENT. IN ON, SIGNAL OR REVERT STMT.

5E079I T WHILE FOLLOWED BY INVALID EXPRESSION.

5E080I T 1ST OPERAND IN DISPLAY STATEMENT INVALID.

Explanation: The first operand in a DISPLAY statement must be an element expression enclosed in parentheses.

5E081I T INVALID OR MISSING CONDITION NAME.

Explanation: The keyword ON is not followed by a valid condition name and/or filename.

Examples:
1. ON +ONVERSION GOTO LAB;
2. ON CNVERSION GOTO LAB;
3. ON ENDFILE GOTO LAB; (filename missing)
4. ON ENDPAGE(?RATE)GOTO LAB; (invalid filename)

5E082I T INVALID OR MISSING OPERAND AFTER GOTO IN ON STATEMENT;

Explanation: The keyword GOTO in an ON statement is not followed by an identifier.

Examples:
1. ON CONVERSION GOTO;
2. ON CONVERSION GOTO +AB;

5E083I T UNSPECIFIED ERROR IN ON STATEMENT.

Explanation: The ON statement has the following format:

ON condition {SYSTEM;| ON-unit}

The compiler detected that the ON-condition is neither followed by the keyword SYSTEM nor by a valid ON-unit.

Example: ON CONVERSION +5;

5E084I T INVALID CALL STATEMENT.

Explanation: No identifier, especially no entry name, is following the keyword CALL.

Examples: 1. CALL +AB;
2. CALL;

5E085I T ERROR IN CLOSE LIST.

Explanation: The CLOSE statement has the following format:

CLOSE FILE (filename) [, FILE (filename)] ...;

Either the keyword CLOSE or one of the commas in the list is not followed by the keyword FILE.

Examples: 1. CLOSE FLE (OUT);
2. CLOSE (OUT);
3. CLOSE FILE (OUT), (IN);

5E086I T ERROR IN FILE OPTION

Explanation: Syntax error. The file option consists of the keyword FILE followed by the file name enclosed in parentheses.

Examples: 1. OPEN FILE (+-*);
2. OPEN FILE IN);
3. CLOSE FILE (IN ;

where IN is a valid file name.

5E087I T ERROR IN OPEN LIST.

Explanation: The OPEN statement has the following format:

OPEN FILE (filename) options group [,FILE (filename) options group]...;

Either the keyword OPEN or one of the commas in the list is not followed by the keyword FILE.

Examples: 1. OPEN FLE (IN);
2. OPEN (IN);
3. OPEN FILE (IN), (OUT);

5E088I T WRONG FILE OPTION IN READ, WRITE, OR REWRITE STMNT.

Explanation: The keyword READ, WRITE, or REWRITE is not followed by the keyword FILE.

5E089I T INVALID OR MISSING OPERAND IN PAGESIZE OPTION.

5E090I T NO SET OPTION IN LOCATE STATEMENT.

Explanation: The file option in a LOCATE statement is not followed by the keyword SET.

Examples: 1. LOCATE A FILE (OUT);
2. LOCATE A FILE (OUT) SE (P);

5E091I T INVALID OR MISSING OPERAND IN KEY OPTION.

Explanation: Syntax error. The KEY option must consist of the keyword KEY followed by a parenthesized expression representing a character string.

5E092I T INVALID FROM, FILE, OR INTO OPTION.

Explanation: Syntax error. FROM, FILE, or INTO is not followed by a valid operand, or the operand is not enclosed in parentheses.

Example: PUT FILE OUT EDIT (BUFFER) (A);

5E093I T INVALID OR MISSING OPERAND IN SET, STRING-, OR KEYTO OPTION.

Explanation: Syntax error. E.g., the SET option consists of the keyword SET followed by the name of a pointer variable enclosed in parentheses.

Examples: 1. LOCATE A FILE (OUT) SET (P1 ; where P1 is a pointer variable.
2. LOCATE A FILE (OUT) SET (1);

5E094I T INVALID OR MISSING OPERAND IN KEYFROM OPTION.

Explanation: The keyword KEYFROM must be followed by an element expression enclosed in parentheses.

5E096I T ERROR IN FORMAT LIST

Explanation: The error may be caused by:

1. Left parenthesis of one of the format lists is missing.
2. A left parenthesis or one of the commas in the list is neither followed by an iteration factor nor by a valid format item.
3. An iteration factor in the list is neither followed by a valid format item nor by a format list.

5E097I E MISSING) INSERTED IN FORMAT LIST.

5E098I T MISSING OR INVALID CONTROLVARIABLE IN DO-STATEMENT.

Example: DO C(5) = 1 TO 7;
The control variable C must not be subscripted.

5E099I T INVALID LINE, COLUMN, OR X FORMAT ITEM.

Explanation: Missing or invalid operand in a LINE, COLUMN, or X-format item.

Example: PUT SKIP EDIT (BUFFER) (X(5, A);

In the above example, the right parenthesis enclosing the operand of the X-format item is missing.

5E100I T INVALID R FORMAT ITEM.

Explanation: Missing or invalid operand in an R-format item.

5E101I T MISSING (IN E OR F FORMAT ITEM.

5E102I T MISSING INTEGER IN E OR F FORMAT ITEM.

5E103I T MISSING) IN E OR F FORMAT ITEM.

5E104I T COMMA MISSING AFTER 1ST INTEGER IN E FORMAT ITEM.

5E105I T BUILT-IN FUNCTION AS ARGUMENT OF PSEUDO-VARIABLE.

5E108I T INVALID OPTION LIST IN READ OR WRITE STATEMENT.

5E109I S MAIN PROCEDURE MUST NOT RETURN AN EXPRESSION VALUE.

5E110I S CHARACTER OR BIT EXPRESSION IS TOO LONG.

Explanation: The number of characters resulting from the evaluation of a character-string expression must not exceed 255. For bit-string expressions, the number of resulting bits must not exceed 64.

5E111I T DATA, OPTION, OR FORMAT LIST CONTAINS INVALID ITEM(S).

Examples:
1. PUT SKIP EDIT (BUFFER (A);
   Right parenthesis missing after BUFFER.
2. PUT EDIT SKIP (BUFFER) (A);
   The keyword EDIT must immediately be followed by the data specification.

5E112I T INVALID DATA ELEMENT.

5E113I T INVALID REPETITIVE SPECIFICATION.

5E114I S ENTRYPOINT IN THIS STATEMENT INVALIDLY DECLARED.

5E116I T MISSING OR WRONG BASED VAR. OR FILE OPTION IN LOCATE STMNT.

Explanation: Syntax error. The LOCATE statement has the following format:

LOCATE based variable FILE (filename) SET (pointer variable);

The based variable must be unsubscripted and must not be a minor structure or an element of a structure.

Examples:
1. LOCATE +1 FILE (OUT) SET (P1);
2. LOCATE A1 (OUT) SET (P1);

5E117I T INVALID EXPRESSION.

Explanation: The error may be caused by:

1. Missing operand.
2. Two infix operators not separated by operand.

5E118I E WARNING FOR INCORRECT PREFIX IN ENTRY STATEMENT.

5E119I T TOO MANY ENTRY POINTS AND/OR ON CONDITIONS IN BLOCK.

5E120I S ILLEGAL NULL STATEMENT IN ON-UNIT.

Explanation: The null on-unit must not be specified for the conditions CONVERSION, ENDFILE, and KEY.

5E121I T END OF INVALIDLY NESTED DO GROUP. NESTING EXCEEDS 12 LEVELS.

Explanation: Implementation restriction. The maximum depth of a nested set of DO statements (including repetitive specifications in GET or PUT statements) is 12. This message is issued as a follow-up to message 5E008I.

System Action: The flagged END statement is replaced by a dummy statement.

5E122I S ILLEGAL FILENAME IN ON CONDITION.

5E123I S ILLEGAL LABEL IDENTIFIER IN ON UNIT.

Example:
```
DECLARE C DECIMAL FIXED;
ON CONVERSION GOTO C;
```

5E124I E REVERT STATEMENT WITHOUT CORRESPONDING ON STATEMENT.

5E126I E INCORRECT NUMBER OF ARGUMENTS.

Example: B = SUBSTR(A, 1 1);
Due to a missing comma in the argument list, the compiler recognizes only two arguments.

5E127I E OPTIONS MAY NOT BE SPEC. FOR SUBPROCEDURES. OPTIONS IGNORED.

5E128I T BUILT-IN FUNCTION NAME IN INCORRECT CONTEXT.

Explanation: A built-in function name has explicitly been declared with the BUILTIN attribute, but is used in a non-function-reference context.

Example:
```
DECLARE ABS BUILTIN;
ABS = ABS + 1;
```

Note: Built-in functions without arguments or which have been declared contextually only are not concerned.

5E129I S CONVERSION OF ARITH. DATA TO BIT STRING YIELDS RESULT GT 31.

5E130I T INVALID KEY.

5E131I T MORE THAN 65534 VARIABLES AND/OR CONSTANTS.

Explanation: An internal overflow of the variable and constant counter of the compiler occurred.

5E132I T STACK OVERFLOW. (IF-NEST TOO DEEP).

Explanation: Implementation restriction: The maximum number of IF statements in a nest is 100.

5E133I T PROBABLY BAD IF-NEST.

5E134I T ELSE IMMEDIATELY FOLLOWS IF.

5E135I T ELSE IMMEDIATELY FOLLOWS ANOTHER ELSE.

5E137I T ILLEGAL STATEMENT USED AS UNIT IN AN IF STATEMENT.

Examples:
1. IF element expression THEN FORMAT (format-list);
2. IF element expression THEN unit-1 ELSE FORMAT (format-list);

The FORMAT statement is not permitted as unit in an IF statement.

5E138I T ELSE WITHOUT CORRESPONDING IF.

5E140I S INCORRECT SPECIFICATION OF CONSTANT ARGUMENT.

5E141I T TOO MANY STRUCTURES IN STRUCTURE ASSIGNMENT.

5E142I T NUMBER OF INTERMEDIATE RESULTS IS TOO BIG. STACK OVERFLOW.

5E143I T NON-IDENTICAL STRUCTURING IN STRUCTURE ASSIGNMENT.

5E144I T ARRAY USED IN INCORRECT CONTEXT.

5E145I T STRUCTURE USED IN INCORRECT CONTEXT.

5E146I T INVALID CONVERSION OR ILLEGAL COMBINATION OF DATA TYPES.
Example: P = A; where A is a character string and P is a pointer variable.

5E147I T NON-IDENTICAL NUMBER OF ARRAY ELEMENTS IN ARRAY-ASSIGNMENT.

5E148I T UNPERMITTED ASSIGNMENT TO FUNCTION VALUE.

Explanation: The left side of an assignment statement is a built-in function which is neither a STRING built-in function nor a pseudo variable.

5E149I S NUMBER OF ARGUMENTS IS GREATER THAN TWELVE.

5E150I T TOO MANY REPETITIVE SPECIFICATIONS.

Explanation: Implementation restriction. The number of iteration specifications must not exceed 50.

Example: DO I = 1 TO 2, 2 TO 3, 3 TO 4, ..., 51 TO 52;

System Action: The flagged DO statement is replaced by a dummy statement and the following text is ignored.

5E152I T PROCESSING OF STATEMENT TERMINATED. (TABLE OVERFLOW).

Explanation: An internal table overflow occurred during the processing of a DO statement.

Since the DO statement will be deleted from the text string, there will be a surplus END statement in the source program.

User Response: Subdivide statement and recompile.

5E153I T POINTER AS ELEMENT OF DATA LIST.

5E154I W POSSIBLE ERROR IN FORMAT ITEM IF USED FOR OUTPUT.

5E155I S INCORRECT ARGUMENT IN BUILT-IN FUNCTION OR PSEUDO-VARIABLE.

Example:
```
DECLARE (A,B) CHARACTER (2);
B = SUBSTR(A,5,4);
```

Since A and B are only two characters long, the arguments 5 and 4 in the argument list are invalid.

5E0156I S INVALID NUMBER OF DIMENSIONS.

Example: A (2 3,1) = 15; where A is declared as a three-dimensional array. The error is caused by a missing comma between the integers 2 and 3.

5E157I W ERROR IF USED FOR OUTPUT.

5E158I T ENTRY NAME OR LABEL ON LEFT SIDE OF ASSIGNMENT STATEMENT.

Example: LAB: N = 3; DO LAB = A TO B; where A and B are valid expressions.

5E159I T R FORMAT ITEM IN ITERATION LIST AT DEPTH GREATER THAN TWO.

5E160I T STATEMENT TOO LONG. STATEMENT DELETED.

Explanation: Internal buffer overflow.
User Response: Subdivide statement and recompile.

5E161I T TOO MANY IDENTIFIERS IN PROGRAM.

5E162I S CONTROL ITEMS NOT ALLOWED FOR THIS STATEMENT.

5E163I T NO LABEL DESIGNATOR IN REMOTE FORMAT ITEM.

5E164I E LABEL CONST. IN R FORMAT ITEM NOT INTERNAL TO CRRNT BLOCK.

Explanation: The R format item and the specified FORMAT statement must be internal to the same block.

5E165I S NO POINTER VARIABLE IN SET OPTION.

5E166I S INCORRECT RECORD VARIABLE.

5E167I W RECORD VARIABLE ON WRONG BOUNDARY.

Explanation: The variable is not on a double-word boundary. An error may occur if later a READ statement with the SET option is issued, and a similar variable is used.

5E168I S RECORD VARIABLE ON WRONG BOUNDARY.

5E169I S RECORD VARIABLE LENGTH NOT IN ACCORDANCE WITH RECORDSIZE.

5E170I S INCORRECT VARIABLE IN STRING OPTION.

5E171I T INCORRECT NAME IN FILE OPTION.

Explanation: File name not or incorrectly declared.

5E172I S STATEMENT NOT IN ACCORDANCE WITH FILE DECLARATION.

5E173I T INCORRECT ITEM IN DATA LIST.

5E174I T NO STRING VARIABLE IN SUBSTR PSEUDO-VARIABLE.

5E175I T FORMAT LIST TOO LONG.

Explanation: Internal buffer overflow.

5E176I S FORMAT STATEMENT NOT PRECEDED BY LABEL. STATEMENT DELETED.

Explanation: A FORMAT statement must be preceded by at least one label.

5E177I T TOO MANY FORMAT LABELS IN PROGRAM.

Explanation: Implementation restriction. The number of labels preceding FORMAT statements in one program is restricted to 127.

5E178I T NESTING OF ITERATION LIST IN FORMAT LIST TOO DEEP.

5E179I S REMOTE FORMAT ITEM IN FORMAT STATEMENT. STATEMENT DELETED.

Explanation: A FORMAT statement cannot contain an R format item.

System Action: The error statement is deleted from the text string.

5E180I S INCORRECT A,B FORMAT ITEM IN GET STATEMENT.

5E181I S VIOLATION OF FORMAT ITEM RESTRICTION.

5E182I W MOD (LENGTH OF RECORD VARIABLE,8) IS UNEQUAL TO FOUR.

Explanation: If the V option is used, the record size of records to be transferred by a READ SET or LOCATE statement must yield a remainder of 4 after division by 8.

5E183I S INCORRECT VARIABLE IN REPLY OPTION.

5E184I S WRONG VARIABLE IN SET OR KEYTO OPTION.

5E186I T TOO MANY REPETITIVE SPECIFICATIONS IN DATA SPECIFICATION.

5E187I S LENGTH OF RECORD VARIABLE GREATER THAN MAXBLOCKSIZE.

5E218I S ILLEGAL EXPRESSION IN ASSIGNMENT STATEMENT.

5E219I S MORE THAN TWELVE PARAMETERS IN PROCEDURE/ENTRY STATEMENT.

<u>System Action</u>: The parameter list is truncated on the right.

5E228I E CHARACTER STRING IN DISPLAY STATEMENT LONGER THAN 80 BYTES.

5E229I E EVALUATION OF OPTIM. SUBSCR. YIELDS DISPLACEM. GREATER 32K

<u>Explanation</u>: At least one subscripted variable in this statement is outside the declared bound of the array.

<u>Example</u>: The semantically wrong statement A(I) = A(I+35000); where A is declared as A(10), will cause this diagnostic message. This error is only detected if OPT is specified.

5E230I W IMPLEMENTATION DEFINED SUBROUTINE.

<u>Explanation</u>: This warning message will appear for each statement using one of the facilities DYNDUMP, OVERLAY, IJKTRON, IJKTROF, IJKEXHC.

5E231I E TOO MANY ARGUMENTS FOR IJKEXHC IN ONE BLOCK.

5E232I E INVALID ARGUMENT(S) FOR EXHIBIT CHANGED IGNORED.

5E233I E UNPERMITTED VALUE OF CONSTANT SUBSCRIPT(S).

<u>Explanation</u>: Constant subscript(s) too large. The absolute value of the displacement to the origin of the array is greater than 32767.

5E234I E NO SCALE FACTOR GIVEN IN BUILT-IN-FUNCT.

<u>Explanation</u>: Concerning the built in functions ADD, MULTIPLY, DIVIDE for fixed-scale arguments.

5E235I S INTERMED. RESULT IN ADD-FUNCT. TOO LONG. STATEMT. IGNORED

<u>Explanation</u>: Length of necessary working space (resulting from precision and scale of the arguments) greater than hardware defined limits (only for fixed scale arguments).

5E236I S INTERMED. SCALE-FACT. EXCEEDS PERMITTED RANGE

<u>Explanation</u>: The intermediate scale factor in the built-in-functions ADD, MULTIPLY, or DIVIDE is greater than 127 or less than -128 (only for fixed-scale arguments).

5E237I S EVEN PRECISION HERE NOT ALLOWED. CHOICE ODD TARGET PRECISION.

5E238I E TIME/DATE/OR NULL ASSUMED TO NAME PL/I BUILT-IN-FUNCTION

<u>Explanation</u>: Builtin functions without arguments should be explicitely declared with the BUILTIN attribute.

5E239I E UNKNOWN FUNCTION OR SUBROUTINE. ATTR. ENTRY ASSUMED

<u>Explanation</u>: Entry names must be explicitly declared with the attribute ENTRY.

5G01I PROGRAM BLOCK GREATER THAN 32K. COMPILATION TERMINATED.

5G02I SOURCE PROGRAM TOO LONG. COMPILATION TERMINATED.

5G03I STATIC STORAGE OVERFLOW. COMPILATION TERMINATED.

5G04I AUTOMATIC STORAGE OVERFLOW. COMPILATION TERMINATED.

5G05I MORE THAN 256 ESID NUMBERS NECESSARY. COMPILATION TERMINATED.

5G06I MORE THAN 65,534 VARIABLES AND/OR CONSTANTS. COMPILATION TERMINATED.

5G07I POSSIBLE RECURSIVE USE OF EXTERNAL PROCEDURE. COMPILATION TERMINATED.

5W01I SUCCESSFUL COMPILATION.

5W02I COMPILATION IN ERROR.

Appendix G. Object-Time Diagnostic Messages

Note: For a discussion of the program-checkout facilities of the compiler, refer to the section Program-Checkout Facilities.

The format of object-time diagnostic messages is as follows:

```
5L00I  ccqqqqqq aaaaaa ERROR nnnn
```

5L00I is a prefix to identify the message as a PL/I object-time message.

cc are two hexadecimal digits identifying the message, (see the message code list below),

qqqqqq are six hexadecimal digits qualifying the message code with the address of a file, if applicable. Otherwise six zeros.

aaaaaa are six hexadecimal digits specifying the address where the error was detected. If the error was detected in a library routine, aaaaaa is the address of the instruction that follows the call of the routine in the PL/I object program.

nnnn is the number of the source statement that caused the error. This number is printed only if STMT was specified in the PL/I PROCESS card. If STMT was not specified or when the compiler cannot determine the statement that caused the error, nnnn is set to 0000.

The messages are listed below by message code number (cc above).

Notes:
1. For errors not raising an ON-condition (other than ERROR), a message is printed for the specific error and the ERROR condition is raised. This applies to all errors with a message code higher than 10.
2. If SYSLST is not yet opened (for example, because of insufficient storage available for DSA), some of the messages may be printed only on the console.

LIST OF MESSAGE CODES

PL/I ON-Condition Comments

These object-time diagnostic messages are issued only if an enabled PL/I ON-condition is raised and no ON-unit is currently being executed for this condition.

01 OVERFLOW
02 UNDERFLOW
03 ZERODIVIDE
04 FIXEDOVERFLOW
05 SIZE
06 CONVERSION
09 ERROR
0A ENDFILE
0C TRANSMIT
0D KEY
0E RECORD

Only the last four conditions use the filename qualification.

With indexed-sequential files the ENDFILE condition will also be raised if a key higher than the last one on the file is requested. If the ENDFILE condition is not enabled for the file, the message 80 - NO RECORD FOUND - will be issued.

Hardware Interrupts

Severe programming errors might lead to program-check hardware interrupts during the execution of a PL/I program. These possible interrupts are identified by the following codes:

11 Operation Exception
12 Privileged-Operation Exception
13 Execute Exception
14 Protection Exception
15 Addressing Exception
16 Specification Exception
17 Data Exception
1E Significance Exception

Each of these exceptions is briefly discussed below.

11 Operation Exception. When an operation code is not assigned or the assigned operation is not available on the particular model, an operation exception is recognized. The operation is suppressed.

The instruction-length code is 1, 2, or 3.

12 Privileged-Operation Exception. When a privileged instruction is encountered in the problem state, a privileged-operation exception is recognized. The operation is suppressed.

The instruction-length code is 1 or 2.

13 Execute Exception. When the subject instruction of EXECUTE is another EXECUTE, an execute exception is recognized. The operation is suppressed.

The instruction-length code is 2.

14 Protection Exception. When the key of an instruction halfword or an operand in storage does not match the protection key in the PSW, a protection exception is recognized.

The operation is suppressed on a store violation, except in the case of STORE MULTIPLE, READ, DIRECT, TEST, AND SET, and variable-length operations, which are terminated.

Except for EXECUTE, which is suppressed, the operation is terminated on a fetch violation.

The instruction-length code is 0, 2, or 3.

15 Addressing Exception. When an address specifies any part of data, an instruction, or a control word outside the available storage for the particular installation, an addressing exception is recognized.

In most cases, the operation is terminated for an invalid data address. Data in storage remain unchanged, except when designated by valid addresses. In a few cases, an invalid data address causes the instruction to be suppressed - AND (NI), EXCLUSIVE OR (XI), OR (OI), MOVE (MVI), CONVERT TO DECIMAL, DIAGNOSE, EXECUTE, and certain store operations (ST, STC, STH, STD, and STE). The operation is suppressed for an invalid instruction address.

The instruction-length code normally is 1, 2 or 3; but may be 0 in the case of a data address.

16 Specification Exception. A specification exception is recognized when:

1. A data, instruction, or control-word address does not specify an integral boundary for the unit of information.

2. The R_1 field of an instruction specifies an odd register address for a pair of general registers that contains a 64-bit operand.

3. A floating-point register address other than 0, 2, 4, or 6 is specified.

4. The multiplier or divisor in decimal arithmetic exceeds 15 digits and sign.

5. The first operand field is shorter than or equal to the second operand field in decimal multiplication or division.

6. The block address specified in SET STORAGE KEY or INSERT STORAGE KEY has the four low-order bits not all zero.

7. A PSW with a nonzero protection key is encountered when protection is not installed.

The operation is suppressed. The instruction-length code is 1, 2, or 3.

17 Data Exception. A data exception is recognized when:

1. The sign or digit codes of operands in decimal arithmetic or editing operations or in CONVERT TO BINARY are incorrect.

2. Fields in decimal arithmetic overlap incorrectly.

3. The decimal multiplicand has too many high-order significant digits.

The operation is terminated. The instruction-length code is 2 or 3.

1E Significance Exception. When the result of a floating-point addition or subtraction has an all-zero fraction, a significance exception is recognized.

The operation is completed. The interruption may be masked by PSW bit 39. The manner in which the operation is completed is determined by the mask bit.

The instruction-length code is 1 or 2.

Housekeeping Errors

21 STORAGE OVERFLOW
There is not sufficient storage available for dynamic storage allocation.

22 INVALID LABEL
The label variable in a GOTO statement does not contain a valid label.

23 SECOND CALL OF MAIN
A procedure with the option MAIN is called by a PL/I program.

24 PARAMETER NOT ON DOUBLE-WORD BOUNDARY
Procedure expecting double-precision floating-point variable as parameter has been passed single-precision value.

25 INVALID SIGN CHARACTER
Incorrect character for sign position of PICTURE data containing T, I, or R in specification.

Mathematical and Arithmetical Subroutines (Short Arguments)

30 X LT 0 IN SQRT(X)

31 ABS(X) GE (2**18)*K IN SIN(X) OR COS(X) (K=PI) OR SIND(X) OR COSD(X) (K=180)

32 ABS(X) GE (2**18)K IN TAN(X) (K=PI) OR TAND(X) (K=180)

33 X TOC NEAR SINGULARITY IN TAN(X) or TAND(X)

34 Y=X=0 IN ATAN(Y,X)

35 X GR 174.6 IN SINH(X) OR COSH(X)

36 X GR 174.6 IN EXP(X)

37 X GR 1 IN ATANH(X)

38 X LE 0 IN LOG(X) OR LOG2(X) OR LOG10(X) OR X LE 0 AND Y NOT FIXED POINT (P,0) IN EXPRESSION X**Y

39 X=0, Y LE 0 IN X**Y

3A X=0, N=0 IN X**N

Mathematical and Arithmetical Subroutines (Long Arguments)

40 X LT 0 IN SQRT(X)

41 ABS(X) GE (2**50)*K IN SIN(X) OR COS(X) (K=PI) OR SIND(X) OR COSD(X) (K=180)

42 ABS(X) GE (2**50)*K IN TAN(X) (K=PI) OR TAND(X) (K=180)

43 X TOO NEAR SINGULARITY IN TAN(X) OR TAND(X)

44 Y=X=0 IN ATAN(Y,X)

45 X GR 174.6 IN SINH(X) OR COSH(X)

46 X GR 174.6 IN EXP(X)

47 X GR 1 IN ATANH(X)

48 X LE 0 IN LOG(X) OR LOG2(X) OR LOG10(X) OR X LE 0 AND Y NOT FIXED POINT (P,0) IN EXPRESSION X**Y

49 X=0, Y LE 0 IN X**Y

4A X=0, N=0 IN X**N

Other Built-in Functions

50 Y=0 IN MOD(X,Y)
Binary fixed arguments

51 Y=0 IN MOD(X,Y)
Decimal fixed arguments

52 Y=0 OR
ABS(X/Y) GT 7.2*10**75 IN MOD(X,Y)
Short floating-point arguments

53 Y=0 OR
ABS(X/Y) GT 7.2*10**75 IN MOD(X,Y)
Long floating-point arguments

54 MOD(X,Y) GE ABS(Y)
Short floating-point arguments

55 MOD(X,Y) GE ABS(Y)
Long floating-point arguments

MOD for floating-point arguments will be calculated as
a=X/Y; b=Y*a; MOD(X,Y)=X-b

If the exponent of X is so high that X+Y has the same value as X, then MOD(X,Y)=0; message 54 or 55 will be generated in such a case.

Input/Output Errors

61 FORMAT ERROR
Illegal combination of data list item and format list item.

62 END OF STRING
Attempt to read or write beyond the specified string in a GET EDIT or PUT EDIT statement with the STRING option.

63 ILLEGAL USE OF CONTROL FORMAT OR OPTION
An invalid PAGE, SKIP, LINE, or COLUMN format is specified for a file.

64 ILLEGAL USE OF STREAM FILE
Attempt to execute a disallowed GET EDIT or PUT EDIT statement for a STREAM file.

This error message may also occur if a program processes file labels, but the job-control LBLTYP card has been omitted in the job-control deck for the program.

65 ILLEGAL USE OF CONSECUTIVE BUFFERED FILE
Attempt to execute a disallowed READ, WRITE, REWRITE, or LOCATE statement for a CONSECUTIVE BUFFERED file.

This error message may also occur if a program processes file labels, but the job-control LBLTYP card has been omitted in the job-control deck for the program.

66 ILLEGAL USE OF CONSECUTIVE UNBUFFERED FILE
Attempt to execute a disallowed READ, WRITE, or REWRITE statement for a CONSECUTIVE UNBUFFERED file.

This error message may also occur if a program processes file labels, but the job-control LBLTYP card has been omitted in the job-control deck for the program.

67 ILLEGAL USE OF REGIONAL FILE
Attempt to execute a disallowed READ, WRITE, or REWRITE statement for a REGIONAL file.

This error message may also occur if a program processes file labels, but the job-control LBLTYP card has been omitted in the job-control deck for the program.

69 PAGE SIZE OPTION FOR NON-PRINT FILE

6A ILLEGAL USE OF INDEXED SEQUENTIAL FILE
Attempt to execute an invalid READ, WRITE, or REWRITE statement for an INDEXED SEQUENTIAL file.

6B ILLEGAL USE OF INDEXED DIRECT FILE
Attempt to execute an invalid READ, WRITE, or REWRITE statement for an INDEXED DIRECT file.

This error message may also occur if a program processes file labels, but the job-control LBLTYP card has been omitted in the job-control deck for the program.

6C INPUT DATA ELEMENT TOO LONG
Attempt to read an element of excessive length in a GET LIST statement.

6D TOO MANY CONCURRENT I/O ERRORS FOR STACK SIZE
Indicates that more than three files have WLR and/or TRANSMIT errors being handled at the same time.

6E FILE IN ERROR NOT IN STACK
Indicates that a file with WLR or TRANSMIT error flagged in the DTF appendage is not in the error file stack.
This message can also occur if the LBLTYP card has been omitted, thereby causing label data to overlay and set the appropriate bit in the DTF appendage.

6F ILLEGAL USE OF STREAM FILE
Attempt to execute a disallowed GET LIST or PUT LIST statement for a STREAM file.

70 ERROR DURING POSITIONING OF INDEXED SEQUENTIAL INPUT FILE
An error has occurred during the positioning to the record key specified in the KEY option of a READ statement.

71 ERROR DURING INITIALIZATION OF INDEXED SEQUENTIAL OUTPUT FILE
The cylinder index area is not large enough to accommodate all entries required to index each cylinder specified for the prime data area.

72 ERROR DURING INITIALIZATION OF INDEXED SEQUENTIAL OUTPUT FILE
The master index area is not large enough to accommodate all entries required to index each track of the cylinder index.

7B END OF STRING
Attempt to read or write beyond the specified string in a GET LIST or PUT LIST statement with the STRING option.

If the ERROR condition is raised as a result of System action for the KEY condition, one of the following messages may be printed to give a more specific description of the error that caused the KEY condition to be raised.

80 NO RECORD FOUND
The record to be retrieved by a READ KEY from an INDEXED file has not been found in the data file.

81 OVERFLOW AREA FULL
There is no more space available in the overflow area(s) for the record to be added to an INDEXED DIRECT file by a WRITE KEYFROM statement.

82 PRIME DATA AREA FULL
The prime data area has been filled while creating or extending an INDEXED SEQUENTIAL file by a WRITE KEYFROM statement.

83 DUPLICATE RECORD
The record being added by a WRITE KEYFROM STATEMENT to an INDEXED SEQUENTIAL or DIRECT file has a duplicate record key of another record in the file.

84 SEQUENCE CHECK
The record being written by a WRITE KEYFROM statement to an INDEXED SEQUENTIAL file is not in the sequential order required.

87 FORMAT ERROR IN INPUT
a) Delimiter is neither blank nor comma

b) Character B is missing in external format of a bit string

c) External format of data item is incompatible with internal declaration; for example:

External	Internal
Character string	Bit string
String data	Numeric, E, F-format

Appendix H. I/O Statement Format And On-Condition Checklist

VALID INPUT/OUTPUT STATEMENT FORMATS AND APPLICABLE ON-CONDITIONS	TYPE OF FILE	STREAM			RECORD																			
					SEQUENTIAL												DIRECT							
					CONSECUTIVE BUFFERED			CONSECUTIVE UNBUFFERED						INDEXED SEQ			REGIONAL (1)			REGIONAL (3)			INDEXED DIRECT	
		INPUT	OUTPUT, NOT PRINT	OUTPUT PRINT	INPUT	OUTPUT	UPDATE	INPUT DECLARED	OUTPUT DECLARED	UPDATE DECLARED	BACKWARDS, INPUT NOT DECLARED	INPUT/OUTPUT NOT DECLARED, INPUT	INPUT/OUTPUT NOT DECLARED, OUTPUT	INPUT	OUTPUT	UPDATE	INPUT	OUTPUT	UPDATE	INPUT	OUTPUT	UPDATE	INPUT	UPDATE
OPEN	FILE (filename)	O	O	O	M	M	M	M	M	M				M	M	M	M	M	M	M	M	M	M	M
	FILE (filename) INPUT										M	M												
	FILE (filename) OUTPUT												M											
	FILE (filename) PAGESIZE (n)			O																				
CLOSE	FILE (filename)	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
GET*	FILE (filename) EDIT (data) (format) [(data)(format)] . . .	O																						
	FILE (filename) LIST (data)	O																						
PUT*	FILE (filename) EDIT (data) (format) [(data)(format)] . . .		O	O																				
	FILE (filename) LIST (data)		O	O																				
	FILE (filename) PAGE [LINE(n)]			O																				
	FILE (filename){PAGE|LINE (n)|SKIP (n)}			O																				
	FILE (filename){PAGE|LINE (n)|SKIP (n)}EDIT (data)(format)[(data) (format)]			O																				
	FILE (filename){PAGE|LINE (n)|SKIP (n)} LIST (data)			O																				
READ	FILE (filename) INTO (variable)				O		O	O		O	O	O		O		O								
	FILE (filename) SET (pointer)				O		O																	
	FILE (filename) INTO (variable) KEY (expression)													O		O	O		O	O		O	O	O
	FILE (filename) INTO (variable) KEYTO (variable)													O		O								
REWRITE	FILE (filename)						O																	
	FILE (filename) FROM (variable)						O			O						O								
	FILE (filename) FROM (variable) KEY (expression)																		O			O		O
LOCATE	variable FILE (filename) SET (pointer)					O																		
WRITE	FILE (filename) FROM (variable)					O			O				O											
	FILE (filename) FROM (variable) KEYFROM (expression)														O			O	O		O	O		O
CONDITIONS WHICH MAY OCCUR	CONVERSION	O	O	O																				
	SIZE	O	O	O																				
	ENDFILE (filename)	O			O		O	O		O	O	O		O		O								
	ENDPAGE (filename)			O																				
	KEY (filename)													O	O	O	O	O	O	O	O	O	O	O
	RECORD (filename)				O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
	TRANSMIT (filename)	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O

Symbols used: M = Use of this statement is mandatory

O = For I/O statements: Use of this statement format is optional
For ON conditions: This condition may occur

* = Note that GET/PUT STRING is not an I/O statement and may be used without

Appendix I. File Attributes And Options

FILE ATTRIBUTES AND OPTIONS \ TYPE OF FILE	STREAM INPUT CARD	STREAM INPUT TAPE	STREAM INPUT DASD	STREAM OUTPUT NOT PRINT CARD/PRINTER	STREAM OUTPUT NOT PRINT TAPE	STREAM OUTPUT NOT PRINT DASD	STREAM OUTPUT PRINT PRINTER	STREAM OUTPUT PRINT TAPE	STREAM OUTPUT PRINT DASD	RECORD CONSECUTIVE BUFFERED INPUT CARD	BUFFERED INPUT TAPE FORWARDS	BUFFERED INPUT TAPE BACKWARDS	BUFFERED INPUT DASD	BUFFERED OUTPUT CARD/PRINTER	BUFFERED OUTPUT TAPE	BUFFERED OUTPUT DASD	BUFFERED UPD. DASD UPDATE	UNBUFFERED TAPE INPUT/OUTPUT	UNBUFFERED DASD INPUT/OUTPUT	UNBUFFERED DASD UPDATE	REGIONAL (1) INPUT (DASD ONLY)	REGIONAL (1) OUTPUT	REGIONAL (1) UPDATE	REGIONAL (3) INPUT	REGIONAL (3) OUTPUT	REGIONAL (3) UPDATE	INDEXED SEQUENTIAL INPUT	INDEXED SEQUENTIAL OUTPUT	INDEXED SEQUENTIAL UPDATE	INDEXED DIRECT INPUT	INDEXED DIRECT UPDATE
filename [1–6 characters]	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
FILE	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
RECORD										S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
STREAM	D	D	D	D	D	D	D	D	D																						
INPUT	S	S	S							S	S	S	S					E	E		S			S			S			S	
OUTPUT				S	S	S	D	D	D					S	S	S		E	E			S			S			S			
UPDATE																	S			S			S			S			S		S
SEQUENTIAL										D	D	D	D	D	D	D	D	D	D	D							D	D	D		
DIRECT																					S	S	S	S	S	S				S	S
KEYED																					S	S	S	S	S	S	S	S	S	S	S
BACKWARDS												S						O													
PRINT							S	S	S																						
BUFFERED										D	D	D	D	D	D	D	D										D	D	D		
UNBUFFERED																		S	S	S											
ENVIRONMENT (	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
MEDIUM (	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
SYSIPT,	C	C	C							C	C		C																		
SYSPCH,				C	C	C								C	C	C															
SYSLST,				C	C	C	C	C	C					C	C	C															
SYSnnn, [nnn = 000–222]	C	C	C	C	C	C	C	C	C	C	C	S	C	C	C	C	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
2501|2520|2540|1442)	S			C						S				C																	
1403|1404|1443|1445)				C			S							C																	
2400)		S			S			S			S	S			S			S													
2311|2314|2321)			S			S			S				S			S	S		S*	S*	S	S	S	S	S	S	S	S	S	S	S
U (maxblocksize)											C	C	C		C	C	C	C	C	C											
F (blocksize)	S	S	S	S	S	S	S	S	S	S	C	C	C	S	C	C	C	C	C	C	S	S	S	S	S	S	C	C	C	C	C
F (blocksize, recsize)											C	C	C		C	C	C										C	C	C	C	C
V (maxblocksize)											C		C		C	C	C														
BUFFERS (1)	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D														
BUFFERS (2)	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O														
CTLASA|CTL 360														O																	
LEAVE		O			O			O			O	O			O			O													
NOLABEL		O			O			O			O	O			O			S													
NOTAPEMK					O			O							O																
VERIFY						O			O							O	O		O	O		O	O		O	O		O	O		O
CONSECUTIVE	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D											
REGIONAL (1)																					S	S	S								
REGIONAL (3)																								S	S	S					
INDEXED																											S	S	S	S	S
KEYLENGTH (n) [n = 9 - 255 for REGIONAL (3); n = 1 - 255 for INDEXED]																								S	S	S	S	S	S	S	S
EXTENTNUMBER (n)**																					O	O	O	O	O	O	S	S	S	S	S
INDEXMULTIPLE																											O	O	O	O	O
HIGHINDEX ({2311|2314|2321})																											O	O	O	O	O
OFLTRACKS (n) [n = 0 – 8 for 2311; n = 0 – 18 for all other DASDs]																												O			O
KEYLOC (n) [1≤ n ≤ recsize–keylength +1]																											B	B	B	B	B
INDEXAREA (n)[n<32K]																														O	O
ADDBUFF (n)[74+ blocksize+keylength≤ n < 32K]																															O
)	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
EXTERNAL	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D

S = Attribute or option must be specified.
D = Default attribute or option if not specified.
O = Optional attribute or option. Specify if applicable.
C = Choice must be made between these options.
E = Must be specified here or in the OPEN statement (but not in both places).
B = Optional for unblocked files. The default value for blocked files is n = 1.

No entry is permitted where a blank appears.
* UNBUFFERED is not permitted for files residing on a 2321 Data Cell Drive.
**For INDEXED files, EXTENTNUMBER (n) must be specified [2 ≤ n < 256].
For REGIONAL files, EXTENTNUMBER (n) is optional [0< n < 256].

Appendix J. Default Attributes of Coded Arithmetic Variables

DECLARED ATTRIBUTES	DEFAULT ATTRIBUTES
DECIMAL FIXED	(5,0)
DECIMAL FLOAT	(6)
BINARY FIXED	(15)
BINARY FLOAT	(21)
DECIMAL	FLOAT (6)
BINARY	FLOAT (21)
FIXED	DECIMAL (5,0)
FLOAT	DECIMAL (6)
None - initial character I - N	BINARY FIXED (15)
None - all others	DECIMAL FLOAT (6)

Appendix K. Restrictions to The PL/I Subset Language

ALIGNED or UNALIGNED

Must not be specified for minor-structure names.

Arithmetic Constants

Any embedded blanks in arithmetic constants will be deleted from the number string and no error message will be given. However, embedded blanks in repetition-factor fields of PICTURE items are not deleted.

Arrays

The maximum number of arrays in a source module is 32.

Arrays of Structures

Arrays of structures are not implemented.

Attribute Factorization

The maximum attribute factorization depth is 8.

Binary Fixed-Point Data

Binary fixed-point numbers may have a length between 1 and 31 bits. This also applies to all intermediate results in binary fixed-point form.

Binary Floating-Point Data

Binary floating-point data may have a length between 1 and 53 bits.

Bit Strings

Bit strings may have a length between 1 and 64 bits. The default alignment attribute is not implemented; bit strings are aligned by the D-Compiler. A warning message is given if a bit string associated with the default alignment attribute occurs within a structure.

Blanks

Blanks embedded in arithmetic constants will be deleted (see also Arithmetic Constants).

Blanks between operators will also be deleted. E.g., X * * Y; will be interpreted as X**Y. Similarly, 'XXX' 'YYY' will be interpreted as 'XXX''YYY', resulting in a character-string value of XXX'YYY.

Blocks (of Program)

The size of any internal or external program block (exclusive of data) is restricted to 32K. The size of an external block plus all of its internal blocks (exclusive of data) must not exceed 64K.

The depth of nested blocks is restricted to 3. The external procedure counts as depth 1.

The total number of blocks in an external procedure (including the external procedure) must not exceed 63.

Blocksize Options

The block length must be at least 1 byte (18 bytes for magnetic tape files) and must not exceed 32,767 bytes. The device types and corresponding maximum block lengths are as follows:

2540		80
2540	(CTLASA, CTL360)	81
1442		80
1442	(CTLASA, CTL360)	81
2520		80
2520	(CTLASA, CTL360)	81
2501		80
1403	(PRINT attribute or CTLASA or CTL360)	133
1403	(no PRINT attribute)	132
1404	(PRINT attribute or CTLASA or CTL360)	133
1404	(no PRINT attribute)	132
1443	(PRINT attribute or CTLASA or CTL360)	145
1443	(no PRINT attribute)	144
1445	(PRINT attribute or CTLASA or CTL360)	114
1445	(no PRINT attribute)	113
2400	(no PRINT attribute)	32,767
2400	(PRINT attribute)	145
2311	(no key, no PRINT attribute)	3625
2311	(PRINT attribute)	145
2311	(including key)	3605
2314	(no key, no PRINT attribute)	7294
2314	(PRINT attribute)	145
2314	(including key)	7249
2321	(no key, no PRINT attribute)	2000
2321	(PRINT attribute)	145
2321	(including key)	1984

The block size option V must include the control fields for the blocks and records.

Only fixed-length unblocked records are permitted for STREAM files.

The block size options V and U and the F option with the record size option are permitted for magnetic tape files and disk files only.

Built-in Functions

String arguments must not be used in the ROUND built-in function.

Bit arguments must not be used with the UNSPEC built-in function.

Character Strings

Character strings may have a length between 1 and 255.

Compatibility with OS F PL/I

1. A GOTO statement which branches directly into an iterative DO loop will not be diagnosed as an error by the D Compiler, although such a statement is not allowed in the language, and is flagged as illegal by the F Compiler.
2. Certain statements are not recognized by the F Compiler (see DYNDUMP, IJKEXHC, ... in this Appendix).
3. The I/O ENVIRONMENT attributes are not recognized by the F Compiler.

Refer also to Appendix B. Upward Compatibility in the publication IBM System/360 DOS/TOS PL/I Subset Reference Manual, Order No. GC28-8202.

Conversion

Arithmetic to bit string:
The scale factor must be less than the precision.

Bit string to arithmetic:
The maximum length of the bit string to be converted is 31.

Data Storage

Static - internal:
The static storage for any external procedure (excluding external data) must be less than 64K.

Automatic:
The automatic storage area per block must be less than 64K.

Data aggregates:
Each individual data aggregate must be less than 32K.

Decimal Fixed-Point Data

Decimal fixed-point numbers may have a length between 1 and 15 digits. This also applies to all intermediate results in decimal fixed-point form.

Decimal Floating-Point Data

Decimal floating-point numbers may have a length between 1 and 16 digits.

DECLARE Statement

The length of a DECLARE statement is unrestricted; however, the length of one declaration-unit appearing in a DECLARE statement is restricted to

- 136 syntactical elements, if 10K bytes are available to the compiler, and to
- 2000 syntactical elements, if 46K bytes are available to the compiler.

One declaration-unit is delimited by

- the keyword DECLARE and a semicolon, or
- the keyword DECLARE and a first-level comma, or
- two first-level commas, or
- a first-level comma and a semicolon.

Each parenthesis, identifier, comma, attribute, and constant is counted as one syntactical element. A character-string constant in an INITIAL-list counts as two syntactical elements. Consider the following example:

```
DECLARE (X FIXED, D FLOAT) STATIC,
        (A INITIAL (7), B(10)) EXTERNAL,
        NAME CHARACTER (4) INITIAL
        ('ABCD');
```

The above DECLARE statement consists of three declaration-units, the first of which contains 8, the second 13, and the third 10 syntactical elements.

DEFINED Attribute

A bit class variable must not be a DEFINED item. The attributes for the DEFINED item and the base identifier will not be checked to determine whether they correspond to the rules for overlay defining.

Dimension Attribute

The maximum number of dimensions is 3. Each bound must be an unsigned integer less than 32,768. The dimension attribute may be factored.

DISPLAY Statement

The result in the message expression in the DISPLAY statement must not exceed 80 characters. If the REPLY option is used, the message must be followed by the EOB (End of Block) condition by pressing the appropriate keys. For an example see Appendix L. Programming Examples, "Using The Console For Communications".

DO Statement

The number of iteration specifications in a DO nest must not exceed 50.

The maximum depth of a nested set of DO statements is 12. For details on repetitive specification see GET Statement.

DYNDUMP, IJKEXHC, IJKTRON, IJKTROF, OVERLAY Names

The names DYNDUMP, IJKEXHC, IJKTRON, IJKTROF, and OVERLAY are not recognized by the OS PL/I compiler. Consequently, the CALL statement referring to one of these names will result in an unresolved external reference from the linkage editor under the OS PL/I compiler. Under the D-level compiler, a warning message is issued for each statement using one of these names.

END Statement

If a label follows the END statement, it must be the label of the nearest unmatched PROCEDURE, BEGIN, or DO statement. If a BEGIN or DO statement is preceded by more than one label, only the one closest to the statement identifier may be used with the END statement.

Exponent Subfield

The exponent subfield for decimal and binary floating-point constants is restricted to 3 digit positions for binary and 2 digits for decimal constants.

Files (unbuffered)

For unbuffered files the RECORD condition will not be raised for records of incorrect length, because for the implementation of unbuffered files the system work files have been used (compiler enters the DTFSD parameter TYPEFLE=WORK in the DTF table).

FORMAT Statement

Replication factors:
The replication factor in a FORMAT statement may range between 1 and 255.
The depth of nested replication factors in a format list of a FORMAT statement is limited to 2.

Format constants:
The format constants must be such that w, d, s, and p are decimal integer constants. Only p may be signed (positive or negative). The A, X, LINE, and COLUMN field widths must be less than 256. The B field width must be less than 65.
The E and F field width must be less than 33. This width includes the sign for output fields even when they are positive, i.e., written as a blank. A SKIP must be less than 4.
The exponent subfield for input data described by the E format specification is limited to 2 digit positions.
The exponent subfield for output data described by the E format specification is always written with 2 digit positions.

GET Statement

The replication factor in a format list in GET or PUT statements may range between 1 and 255.

The depth of nested replication factors in a format list of GET or PUT statements is restricted to 5. If the format list contains a remote format item that is contained in a replication nest, it must not be at a depth greater than 2.

The depth of a nested set of repetitive specifications as well as the total number of repetitive specifications in GET and PUT statements are restricted to 11.

Identifiers

The length of EXTERNAL identifiers must not exceed 6 characters. This also applies to names that are external by default, such as file names, names of external procedures, etc.

IF Nesting

The maximum number of IF statements in a nest is 100.

Implicit Declarations

The identifiers DATE, NULL, and TIME should always be declared explicitly. If they are not explicitly declared a warning message is issued, and the BUILTIN attribute is assumed.

INITIAL Attribute

The length of the INITIAL-list for a character-string array is restricted by the following formula:

```
NC * LE + 14 * NF < NI
```

where

NC = the number of constants in the INITIAL-list

LE = the length of one array element

NF = the number of iteration factors

NI = 1500 (if 10K are available to the compiler)
18000 (if 46K are available to the compiler)

Consider the following example:

```
DECLARE CH(10) CHARACTER(250) INITIAL
   ((3)(2)'A','B',(2)'C','D','E','F',
   'G','H');
```

The INITIAL-list in the above DECLARE statement contains eight constants and one iteration factor. String repetition factors (as in (2)'A' and (2)'C') are not counted. The length of one array element is 250.

Application of the above formula yields a result of 2014 which is in error if NI = 1500.

KEY Condition

The KEY condition will not be raised for REGIONAL files if an attempt is made to add a duplicate key by a WRITE statement.

Labels

The total number of labels for all remote FORMAT statements in an external procedure must not exceed 127. This restriction is independent of the size of the available background program area.

Since environmental information is assigned to a label variable during assignment, a static label variable must be initialized each time a procedure is activated.

List I/O

The statement PUT LIST(NULL); - where NULL is declared as the built-in function - will not be diagnosed as an error, but will be executed giving unpredictable output data.

Names

Internal names:
The maximum number of names in all DECLARE statements of a program block is 3048. The maximum number of names given all its attributes by default is 3048.

Note: The above restrictions are applicable only if the source program is compiled on a 16K system. The restrictions are eased considerably with the availability of additional core storage.

External names:
The number of external names must not exceed 255. Names of external structures count as two names. This restriction is independent of the size of the available background program area.

Note: The number 255 includes the names of all library subroutines used by this external procedure.

Total number of names:
The total number of distinct internal and external names in a source program must not exceed 32,000. This restriction is independent of the size of the available background

Nesting I/O Statements

While an I/O statement is active, no other I/O statement must be activated (GET and PUT STRING are considered I/O statements in this connection). Thus, in the following example the second PUT statement is not allowed since it is 'nested' in the first one.

```
PUT FILE (X) EDIT (FUNCT(PAR1,PAR2,....)
   (format list);
     .
     .
     .
FUNCT: PROCEDURE (PARA1,PARA2,....)
   RETURNS (CHAR(120));
DCL Y CHAR (120);
     .
     .
     .
PUT STRING (Y) EDIT (data list) (format
   list);
     .
     .
     .
RETURN (Y);
END FUNCT;
```

ON Statement

If the condition of the ON statement is CONVERSION, ENDFILE, or KEY, the action must not be the null statement. A prefix is not allowed in an ON statement.

ON ENDFILE must not be specified with default files. When a program uses an implicit file declaration, such as GET FILE LIST (A,B,C); it is not possible to use ON ENDFILE (SYSIN). Therefore, when the ENDFILE condition is raised, a message occurs, and the job is cancelled.

When a key error occurs in a WRITE statement, the KEY condition is raised during execution of the current statement or the next I/O operation.

The standard system action for FIXEDOVERFLOW is comment and raise the ERROR condition.

PAGESIZE Option

The default condition is the size specified by the line count of the system.

Parameters

The number of distinct parameters of a procedure must not exceed 12. The same parameter appearing in a number of parameter lists of the same procedure (one PROCEDURE statement and several ENTRY statements, each with parameter lists) is considered as only one parameter.

Entry name parameters must be explicitly declared with the ENTRY attribute.

PICTURE Attribute

A PICTURE specification must have at least one PICTURE character other than M, V, K, or G. Arithmetic pictures must not have more than 32 characters excluding M, V, K, and G. PICTURE character strings must not have more than 255 characters. A PICTURE character preceded by the replication factor k is considered as k PICTURE characters.

PICTURE Data

Data declared with the PICTURE attribute must not have more than 15 digit-characters for numeric fixed-point data and 16 digit-characters for the mantissa and two for the exponent of numeric floating-point data.

Pictures with the fill character * preceded or followed by one of the characters +, -, S, or $ cause these characters to be replaced by * when the variable has a value of zero. Similarly, CR or DB are replaced by **.

The picture character B is implemented as a conditional insertion character when used in conjunction with a drifting character.

Procedure Default Condition

The default condition for all procedures excluding built-in functions and library subroutines is IRREDUCIBLE. The default condition for all data is ABNORMAL in the DOS/TOS PL/I compiler.

The PL/I Subset language does not have the attributes REDUCIBLE, IRREDUCIBLE, NORMAL, and ABNORMAL. Therefore, the user should familiarize himself with these items if he wishes to run programs written in the PL/I Subset language under OS control. For details on these attributes see the SRL publication IBM System/360, Operating System, PL/I(F) Language Reference Manual, Order No. GC28-8201.

PROCEDURE Statement

The OPTIONS attribute permits an options list, the form of which is (MAIN [, ONSYSLOG]). The MAIN option specifies this procedure to be the initial procedure. The ONSYSLOG option specifies that all output as a result of action taken due to an ON condition is to be printed on the device assigned to SYSLOG. If both options are used, they must appear in the order given above. Procedures declared with the OPTIONS attribute cannot be called from other procedures.

Put Statement

Refer to GET Statement.

Qualified Names

If a qualified name is truncated on the right, the remaining part of the qualified name must be unique. For example, in the structure

```
DECLARE 1 ATR,
         2 A1,
          3 B1,
          3 B2,
           4 D1,
           4 D2,
         2 A2,
          3 B1,
           4 D3,
           4 D4,
          3 B3;
```

the qualification ATR.B1.D3 is not allowed since ATR.B1 is not unique. The correct qualification would be ATR.A2.B1.D3. Ambiguous names may not be flagged by the compiler, and the code produced for such ambiguous references is unpredictable.

Repetition Factor

A repetition factor must be an unsigned decimal integer. Its length is restricted to three digits. Its value must not exceed 255. The two examples below are in error:

```
DECLARE A PICTURE '(0010)X';
DECLARE B PICTURE ' (260)X';
```

No embedded blanks are allowed in the repetition factor. E.g. DECLARE C PICTURE

'(1 2)9'; is invalid. However, preceding or following blanks are allowed, as e.g. in DECLARE D PICTURE'(4)X';

Scale Factor

Declaration of a scale factor is permitted only with decimal fixed-point data. It may range between 0 and 15 and must be unsigned.

Statements

The total number of identifiers, constants, and delimiters (excluding insignificant blanks and comments) contained in a statement must not exceed 230.

The number of different identifiers and constants (excluding constants not contained in an expression) is limited to 90 for each statement.

Note: The above restrictions are applicable only if the program is compiled on a 16K system. Each additional 4K available to the compiler allows an equivalent increase.

Structure Declarations

The maximum logical depth of a structure is 8. The maximum level number is 255. The number of names in a structure is restricted to 62, if 10K are available to the compiler (766 if 46K are available). This includes the major-structure name, minor-structure name(s), and structure-element names.

Structures (level numbers)

Any embedded blanks in level numbers will be deleted from the number string during compilation and no error message will be given. Level numbers may only be factored for elements of a structure, i.e., if factorization occurs in a structure declaration, the corresponding items are recognized as structure elements.

For example, in the declaration

```
DCL 1 A,
      2(B,C,D),
      3(E,F,G);
```

B, C, D, E, F and G will all be assumed to be elements of structure A, and will be assigned the logical level 2.

In order to obtain the structure

```
DCL 1 A,
      2 B,
      2 C,
      2 D,
        3 E,
        3 F,
        3 G;
```

the declaration of D must be removed from the factorization brackets. H1

```
     TRIAL:  PROCEDURE OPTIONS(MAIN);
     DECLARE 1 Y,
(10)   2 HOUR  BINARY(31)FIXED,
       2 MIN   BINARY(31)FIXED,
       2 SEC   BINARY(31)FIXED,
       2 MILL  BINARY(31)FIXED;
     DO I=1 TO 50;
       CALL CLOCK (HOUR,MIN,SEC,MILL);
(11)   PUT EDIT ('HOUR','MIN','SEC','MULL',Y)
       (SKIP,X(3),A(4),X(2),A(3),X(2),A(3),
       X(2),A(4),SKIP,F(6),F(5),F(5),F(6));
     END;
       END TRIAL;

       CLOCK: PROCEDURE(HOUR1,MIN1,SEC1,MILL1);
(2)    DECLARE 1 X,
       2 AH CHARACTER (2),
       2 AM CHARACTER (2),
       2 AS CHARACTER (2),
       2 AMI CHARACTER(3),
       H BIT(16),
       M BIT(16),
       S BIT(16),
       MI BIT(24),
       Y BIT(16),
       B1 BIT(8),
       B2 BIT(8),
       B3 BIT(8),
(9)    HOUR1 BIN FIXED(31),
       MIN1 BIN FIXED(31),
       SEC1 BIN FIXED(31),
       MILL1 BIN FIXED(31);
(1)    STRING(X)=TIME;
(4)    Y= '0000111100001111'B ;
(3)    H =UNSPEC(AH );
       H= H & Y;
(5)    M =UNSPEC(AM );
       M=M & Y;
       S =UNSPEC(AS );
       S=S & Y;
     MI=UNSPEC(AMI);
     MI=MI  &  '000011110000111100001111'B ;
(6)          B1=SUBSTR(H,1,8);
(7)          B2=SUBSTR(H,9,8 );
(8)  HOUR1= B2 +B1 * 1010B ;
             B1=SUBSTR(M,1,8);
             B2=SUBSTR(M,9,8 );
     MIN1=B2 +B1 * 1010B;
             B1=SUBSTR(S,1,8);
             B2=SUBSTR(S,9,8 );
     SEC1=B2+B1 * 1010B;
             B1=SUBSTR(MI,1,8);
             B2=SUBSTR(MI,9,8 );
             B3=SUBSTR(MI,17,8);
     MILL1=B3+B2 * 1010B + B1 * 1100100B;
     END CLOCK;
```

Figure 70. Conversion of Numbers in Character Form to Binary Numbers

Conversion of Numbers in Character Form Into Binary Numbers

The example in Figure 70 (encircled numbers are used for reference purposes only) shows how numbers in character form may be converted into binary numbers. For this purpose the time is used.

Note, however, that the example shows machine-dependent programming and was chosen for illustration purposes only.

The current time is obtained with the TIME built-in function (1) which returns a character string of length nine, in the form hhmmssttt, where:

hh is the current hour of the day
mm is the number of minutes
ss is the number of seconds
ttt is the number of milliseconds in machine-dependent increments

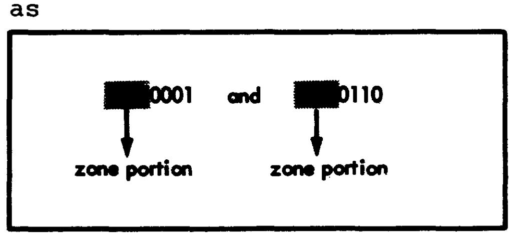
Through the use of the STRING pseudovariable the time (nine characters) is assigned piece by piece to the elements of X (2) (the lengths of the pieces being determined by the lengths of the elements in X).

Take, for example, the time

$$\overbrace{16}^{}\underbrace{}_{AH}\;\overbrace{23}^{AM}\;\overbrace{19}^{}\underbrace{}_{AS}\;\overbrace{080}^{AMI}$$

AM AMI
162319080 (4 P.M., 23 minutes, 19 seconds, and 80 milliseconds)
AH AS

To convert from character to binary, the UNSPEC built-in function (3) is used which returns a bit string that is the internal representation of a given value. Thus, the characters '1' and '6' would be represented as



To eliminate the zone portion of the characters, a mask is used ④ and "anded" with the binary representation of the characters '1' and '6' ⑤:

$$
\begin{array}{r}
1111000111110110 \\
\& \ 0000111100001111 \\
\hline
= \underbrace{00000001}_{1}\underbrace{00000110}_{6}
\end{array}
$$

Although the first byte of H now would contain binary one and the second binary six, the value of H would not be sixteen. To obtain an actual value of binary sixteen, the following is done:

The SUBSTR built-in function is used to (a) extract the first eight bits of H (which would be '00000001') which are assigned to B1 ⑥ and (b) extract the second eight bits ('00000110') which are assigned to B2 ⑦. Then the value of B1 is multiplied by ten (which would yield a value of ten) and the value of B2 (six) is added (giving a total of sixteen).

The result ⑧ must have a precision of 31 according to the precision rules for binary multiplication (⑨ and ⑩), if truncation is to be avoided.

The method that is used to convert the hours is also used to convert the minutes, seconds, and milliseconds. The results are returned to the invoking procedure which prints them in the following form ⑪ :

```
HOUR  MIN  SEC  MULL
 16   33    8   960
HOUR  MIN  SEC  MULL
 16   33    9    80
HOUR  MIN  SEC  MULL
 16   33    9   220
HOUR  MIN  SEC  MULL
 16   33    9   360
HOUR  MIN  SEC  mull
```

Storing And Retrieving Statistical Data

The example in Figure 71 (encircled numbers are used for reference purposes only) shows how volumes of statistical data that are too large to fit in core storage may be stored on disk and retrieved.

```
// JOB STATLAB           BEGIN OF JOB
// ASSGN SYS009,X'291'   LOGICAL DEVICE ADDRESS IS
*  ASSIGNED TO PHYSICAL DEVICE   (SEE ENV ATTRIBUTE)
// DLBL STATLI,'LAB 6 DATA FILE',O,SD
*  STATLI = FILENAME USED IN PROGRAM
*  'LAB 6 DATA FILE'=IDENTIFICATION OF DATA SET ON DISK
*  O MEANS THAT THE FILE MAY BE OVERWRITTEN ANY TIME
*  SD INDICATES A SEQUENTIAL DISK FILE
// EXTENT SYS009,PLIDO3,1,0,1800,40
*  SYS009 = LOGICAL ADDRESS OF EXTENT (SEE ENV ATTRIBUTE)
*  PLIDO3 = SERIAL NUMBER OF VOLUME TO WHICH EXTENT BELONGS
*  1 INDICATES A DATA AREA
*  0 = EXTENT SEQUENCE NUMBER
*  1800 = RELATIVE TRACK NUMBER (TRACK 0 OF CYLINDER 180]
*  40 = NUMBER OF TRACKS THAT MAY BE USED
// ASSGN SYS010,X'291'
// DLBL STATLA 'LAB 6 DATA FILE',O,SD
// EXTENT SYS010,PL1DO3,1,0,1800,40
// OPTION LINK,LIST,SYM,ERRS,60C,NODECK,LISTX, DUMP
// EXEC PL/I EXECUTION IS INITIATED

        STAT:     PROCEDURE OPTIONS (MAIN);
            DECLARE STATLI  FILE OUTPUT RECORD SEQUENTIAL ENVIRONMENT
            (F(80) MEDIUM (SYS009,2311));
            DECLARE STATLA FILE INPUT RECORD SEQUENTIAL ENVIRONMENT
            (F(80) MEDIUM (SYS010,2311));
      |     DECLARE  1  X,
 2    |         2  XX(10) FLOAT BINARY (21),
      |         2  YY(10) FLOAT BINARY (21);
            OPEN FILE (STATLI);
 1    |     DO I = 1 TO 500;
      |     DO J = 1 TO 10;
            IJ = 10  * ( I - 1);   IJ = IJ + J;
            XX(J) = IJ;  YY(J) = IJ;
            END;
 3    |     WRITE FILE (STATLI)  FROM (X);
            END;
            CLOSE FILE (STATLI);
            OPEN FILE (STATLA);
            DO J = 1 TO 500;
 4    |     READ FILE (STATLA) INTO (X);
 5    |     PUT EDIT (X) (SKIP,20 F(4,0));
            END;
            CLOSE FILE (STATLA);
            END;
```

Figure 71. Storing And Retrieving Statistical Data

The program creates 5000 values ① which are stored on disk in groups of 10 values each (② and ③). For this purpose a nested DO-group is used ① .

The same data is then retrieved again from disk ④ and printed in the form shown in Figure 72 ⑤ .

11	12	13	14	15	16	17	18	19	20	11
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1
1	2	3	4	5	6	7	8	9	10	1

Figure 72. Program Output of STAT

Creating And Retrieving a REGIONAL(1) File

The program shown in Figure 73 (encircled numbers are used for reference purposes only) shows how a REGIONAL(1) file ① may be created and retrieved from disk.

```
  // JOB CLEAR
  // ASSGN SYS011,X'291'
  // DLBL UOUT,'CLASS',000
  // EXTENT SYS011,PLID03,1,0,1500,5
2 // EXEC CLRDSK

  CLEAR DISK UTILITY
  UTILITY CONTROL CARDS
  // UCL B=(K=0,D=080),X'40',ON
  // END
  SPECIFIED PARAMETERS
    KEY LENGTH - 0
    DATA LENGTH - 080
3   FILL CHARACTER - X'40'
    OUTPUT PARAMETER - N
  ASSUMED PARAMETERS
    DEVICE TYPE - 2311
  RECORDS/TRACK - 25
   EXTENT     BB          LOWER    LIMIT            UPPER    LIMIT
  SEQ. NO.           C1   C2    H1    H2      C1   C2    H1    H2
   000      000     000  150   000   000     000  150   000   004
  END OF JOB

  // JOB REGIONL
  // OPTION LINK,SYM,LIST,ERRS,60C,NODECK,LISTX
  // ASSGN SYS011,X'291'
  // DLBL CLASS,'PL/I CLASS',000,DA
  // EXTENT SYS011,PLID03,1,0,1500,5
  // ASSGN SYS012,X'291'
  // DLBL CLASSI,'PL/I CLASS',000,DA
  // EXTENT SYS012,PLID03,1,0,1500,5
  // EXEC PL/I

  FTEST:  PROCEDURE OPTIONS(MAIN);
1 DECLARE CLASS FILE OUTPUT DIRECT RECORD ENVIRONMENT (REGIONAL(1)
4 F(80) MEDIUM(SYS011,2311)) KEYED;
  DECLARE CLASSI FILE INPUT DIRECT RECORD ENVIRONMENT (REGIONAL (1)
  F(80) MEDIUM(SYS011,2311)) KEYED;
5 DECLARE X CHARACTER (80);
6 DECLARE IREC PICTURE '(8)9';
  OPEN FILE (CLASS);
7 IREC=0;
  DO I = 1 TO 30;
  X='123456789';
      PUT EDIT(X,IREC) (A(20),F(10,0));
9 WRITE FILE (CLASS) FROM (X) KEYFROM (IREC);
      IREC=IREC+1;
  END;
  CLOSE FILE (CLASS);
8     IREC=0;
      PUT SKIP;
      OPEN FILE (CLASSI);
      DO I = 1 TO 30;
10 READ FILE (CLASSI) INTO (X) KEY  (IREC);
      PUT EDIT(X,IREC) (A(20),F(10,0));
      IREC=IREC+1;
      END;
      CLOSE FILE (CLASSI);
      END;
```

Job Control Statements to Clear And Preformat Disk Area

Figure 73. Creating And Retrieving a REGIONAL(1) File

Before a REGIONAL(1) file can be created, the extents used by the file must be preformatted by the DOS Clear Utility program ②, which creates dummy records that contain a string filled with user-defined characters ③. Only when the disk area preformatted, can the REGIONAL(1) file actually be created with the OUTPUT attribute ①.

Each record in the file has a length of 80 characters (④ and ⑤). The key ⑥ which must be declared as a numeric-character variable with the attributes PICTURE'(8)9' is not contained in the record; it is not written on disk, but only indicates the relative number of the record in the file. The first record in a REGIONAL(1) file always has the relative record number 0 (⑦ and ⑧). The key used to identify the record on disk must be specified in the KEYFROM option ⑨ of the WRITE statement, or in the KEY option ⑩ of the READ statement.

Printed, the records of the program shown in Figure 73 would look as shown in Figure 74.

```
123456789                         0123456789
123456789                         4123456789
123456789                         8123456789
123456789                        12123456789
123456789                        16123456789
123456789                        20123456789
123456789                        24123456789
123456789                        28123456789
123456789                         0123456789
123456789                         4123456789
123456789                         8123456789
123456789                        12123456789
123456789                        16123456789
123456789                        20123456789
123456789                        24123456789
123456789                        28123456789
```

Figure 74. Program Output of FTEST

Creating and Updating a Sequential Disk File

The three programs shown in Figures 75 through 77 (encircled numbers are used for reference purposes only) will create a file, update the file, and punch the file back onto cards. It is a sequential file. The input is also sequential.

The program shown in Figure 75 creates a sequential disk file 1 whose records are 80 bytes long. The input for this file is furnished by presorted cards 2. The records on the cards are just read 3 and written onto disk 4 as they are.

In the program shown in Figure 76 the data on disk created in the first program is updated 1. Input for any updates is furnished by other pre-sorted cards 2.

The information that is processed concerns college students, their credit hours, grade points, etc. 3. The updates reflect changes during a semester.

The update process is as follows: Student identification (ID) on card records 4 are compared 5 with student ID on disk records. If a disk record ID is smaller than the corresponding card ID, a new disk record is read in 6, 7. If the ID numbers are equal, the disk record is updated depending on the code of the card file record. The codes may be one of the following:

R -- concerns only regular hours and grade point changes (8, 9, 10)

F -- concerns only total hours attempted (11, 12, 13)

N -- concerns credit hours (14, 15, 16)

P -- concerns probation (17, 18, 19)

A -- concerns only attempt to change hours (19, 20, 21)

G -- concerns hours towards graduation (22, 23, 24)

When updated, the record is written back onto disk 25. ID errors 26 and data errors 27 are signalled by printing them out. The third program (shown in Figure 77) reads the data on disk (1 and 2) and punches them, as they are 3, into cards 4.

```
// JOB CUMBUILD
// DLBL IJSYS10, 'DATA-FILE'
// EXTENT SYS010,PLID03,1,0,1500,250
// ASSGN SYS010,X'291'
// OPTION LINK,SYM,LIST,ERRS,60C,NODECK,LISTX
// EXEC PL/I

        DISK1:  PROCEDURE OPTIONS (MAIN);
   1  |  DECLARE IJSYS10 FILE OUTPUT RECORD BUFFERED
      |     ENVIRONMENT(MEDIUM(SYS010,2311) F(80) BUFFERS(2));
   2  |  DECLARE CUMCARD FILE INPUT RECORD
      |     ENVIRONMENT(MEDIUM(SYSIPT,2540) F(80));
         DECLARE SEMGRADECARDS CHAR(80);
         ON ENDFILE(CUMCARD) GO TO STOP;
         OPEN FILE(IJSYS10),FILE(CUMCARD);
   3  |  READ:  READ FILE(CUMCARD) INTO (SEMGRADECARDS);
   4  |  WRITE FILE(IJSYS10) FROM (SEMGRADECARDS);
         GO TO READ;
         STOP:  CLOSE FILE(IJSYS10) ,FILE(CUMCARD);
         END;
```

Figure 75. Creation of Sequential Disk File

```
// JOB CUMUPDT
// OPTION LINK,SYM,ERRS,60C,NODECK,LIST,LISTX
// ASSGN SYS010,X'291'
// DLBL IJSYS10,'DATA-FILE'
// EXTENT SYS010,PLID03,1,0,1500,250
// EXEC PL/I

     DISK2:  PROCEDURE OPTIONS (MAIN);
 ①  | DECLARE IJSYS10 FILE UPDATE RECORD
    |    ENVIRONMENT(MEDIUM(SYS010,2311) F(80));
 ②  |    DECLARE CUMCARD FILE INPUT STREAM
    |    ENVIRONMENT(MEDIUM(SYSIPT,2540) F(80));
     DECLARE SIGN CHARACTER(1);
        DECLARE(ID,ALPHA)PICTURE '99999';
    | DECLARE 1 SEMGRADECARDS,
    |          2 SEMHOURSATTEMPTED            PICTURE '99V9',
    |          2 SEMHOURSOFNOCREDIT           PICTURE '99',
    |          2 SEMGRADEPOINTS               PICTURE '99V9',
    |          2 SEMGRADEPOINTAVE             PICTURE '9V999',
    |          2 TOTHOURSATTEMPTED            PICTURE '999V9',
    |          2 TOTHOURSOFNOCREDIT           PICTURE '99',
    |          2 TOTGRADEPOINTS               PICTURE '999V9',
    |          2 CUMULATIVEGRADEPOINTAVE      PICTURE '9V999',
    |          2 TOTHOURSTOWARDGRADUATION     PICTURE '999V9',
    |          2 SEMHOURSOFFORGGRADES         PICTURE '99',
 ③  |          2 MAJOR                        PICTURE '999',
    |          2 MARITALSTATUS                PICTURE '9',
    |          2 ADVISOR                      PICTURE '999',
    |          2 CLASSIFICATION               PICTURE '9',
    |          2 COLLEGE                      PICTURE '9',
    |          2 SEX                          PICTURE '9',
    |          2 STUDENTIDNUMBER              PICTURE '99999',
    |          2 STUDENTALPHANUMBER           PICTURE '99999',
    |          2 SEM                          PICTURE '9',
    |          2 YEAR                         PICTURE '99',
    |          2 AFFILIATION                  PICTURE '9',
    |          2 STUDENTNAME        CHARACTER(23),
    |          2 CONTROLCODE        CHARACTER(1);
        DECLARE FNAME CHAR(2),
              CNAME CHAR(2);
        DECLARE DUMMY CHAR(3);
        DECLARE OD PICTURE '99999' ONAME CHAR(2)
     OPEN FILE(IJSYS10),FILE(CUMCARD);
     /* THE CARDS FOR UPDATE ARE AS FOLLOWS - ALL CREDIT HOURS AND
 ⑩  |          GRADE POINTS ARE OF FORM R+3.0  +9.0  (LETTER, F9.0,F10.0)
    |          R CARDS ARE REGULAR HOURS, GRADE POINTS CHANGES.
 ⑬  |          F GRADE CARDS MAY BE POSITIVE OR NEGATIVE AND ONLY AFFECT TOTAL
    |          HOURS ATTEMPTED. NO-CREDIT CARDS ARE OF THE CHANGE NO HOURS
 ⑯  |          ONLY AND ARE OF THE FORM N3.0 (LETTER, F9.0) */
 ㉑  |    /*  A = CHANGE HOURS ATTEMPTED ONLY. */
    |    /*  G = CHANGE HOURS TO GRADUATION ONLY. */
 ㉔      /* FNAME - FIRST TWO LETTERS OF LAST NAME ON FILE.  */
        /* CNAME - FIRST TWO LETTERS OF LAST NAME ON CARD.  */
     /*   1234567890123456789012345678901234567890   */
     /*   12345AB   R+3.     +12.     F-6.     0.          */
     ON ENDFILE (IJSYS10) GOTO STOP;
 ④  |    GET EDIT (ID,CNAME,DUMMY) (COLUMN(1),F(5),A(2),A(3));
        ONAME = CNAME;
```

Figure 76. Updating Sequential Disk File (1 of 3)

```
         OD = ID;
      NEWCARD:
(6)   LOOP:  READ FILE (IJSYS10) INTO (SEMGRADECARDS);
         FNAME = SUBSTR (STUDENTNAME,1,2);
(5)      IF STUDENTIDNUMBER    = ID    THEN GO TO UPDATE;
(7)      ELSE IF STUDENTIDNUMBER    <  ID    THEN GO TO LOOP;
         ELSE DO;
            PUT EDIT (' STUDENT ID FROM FILE'  ,STUDENTIDNUMBER  ,
                  '  CARD ID NUMBER'  ,ID  ) (SKIP,A,F(6),A,F(6));
            PUT EDIT ('*************************') (A);
            GO TO CHECKNEXTCARD;
            END;
      /*  CHECK FIRST TWO LETTERS OF STUDENT LAST NAME */
      UPDATE:  IF FNAME ¬= CNAME THEN GO TO IDERROR;
         PUT EDIT (STUDENTNAME) (SKIP(2),A);
         PUT EDIT (' BEFORE UPDATE')(SKIP,A);
         PUT EDIT (' ID',ID,'  ALPHA',CNAME,'  TOT. HRS. ATT.',
            TOTHOURSATTEMPTED,    ' HRS. NC',
            TOTHOURSOFNOCREDIT,     ' GRADE PTS.', TOTGRADEPOINTS,
         ' HRS. TO GRAD.', TOTHOURSTOWARDGRADUATION
         ) (   A,F(6),A,A(3),A,F(5,1),A,F(5,1),A,F(4),A,F(4));
      /* SCAN REST OF CARD AND UPDATE FILE SINCE ID AND ALPHA VALID*/
      NEXTDATA:  GET EDIT (SIGN)  (A(1));
(8)      IF SIGN = 'R' THEN GO TO HOURS;
(11)     IF SIGN = 'F' THEN GO TO FGRADE;
         IF SIGN = 'N' THEN GO TO NOCREDIT;
(14)     IF SIGN = 'P' THEN GO TO PROBATION;
(17)     IF SIGN = 'A' THEN GO TO ATTHOURS;
         IF SIGN = 'G' THEN GO TO GRADHOURS;
(19)     IF SIGN ¬= ' ' THEN GO TO DATAERROR; ELSE GO TO CHECKNEXTCARD;
(22)  HOURS:  GET EDIT  (CREDITHOURS,GRADEPOINTS)  (F(9,0),F(10,0));
         TOTHOURSATTEMPTED = TOTHOURSATTEMPTED + CREDITHOURS;
(9)      TOTGRADEPOINTS = TOTGRADEPOINTS + GRADEPOINTS;
         TOTHOURSTOWARDGRADUATION = TOTHOURSTOWARDGRADUATION
            +CREDITHOURS;
         GO TO NEXTDATA;
(12)  FGRADE:                          GET EDIT (CREDITHOURS)  (F(9,0));
                                       TOTHOURSATTEMPTED = TOTHOURSATTEMPTED
              + CREDITHOURS;
         GO TO NEXTDATA;
(15)  NOCREDIT:  GET EDIT (CREDITHOURS)  (F(9,0));
         TOTHOURSOFNOCREDIT = TOTHOURSOFNOCREDIT + CREDITHOURS;
         TOTHOURSATTEMPTED=TOTHOURSATTEMPTED + CREDITHOURS;
         TOTHOURSTOWARDGRADUATION=TOTHOURSTOWARDGRADUATION+CREDITHOURS;
         GO TO NEXTDATA;
(18)  PROBATION:  CONTROLCODE='5';
         GET EDIT  (DUMMI)  (F(9,0));
         GO TO NEXTDATA;
(20)  ATTHOURS:  GET EDIT  (CREDITHOURS,GRADEPOINTS)  (F(9,0),F(10,0));
         TOTHOURSATTEMPTED = TOTHOURSATTEMPTED + CREDITHOURS;
         GO TO NEXTDATA;
```

Figure 76. Updating Sequential Disk File (2 of 3)

```
    GRADHOURS:  GET EDIT (CREDITHOURS,GRADEPOINTS)  (F(9,0),F(10,0));
23     TOTHOURSTOWARDGRADUATION = TOTHOURSTOWARDGRADUATION
             +CREDITHOURS;
25     GO TO NEXTDATA;
    CORRECTFILE:
             REWRITE FILE (IJSYS10) FROM (SEMGRADECARDS);
    BYPASS:
       PUT EDIT ('  AFTER UPDATE')(SKIP,A);
       PUT EDIT ('  ID',OD,'  ALPHA',ONAME,'  TOT. HRS. ATT.',
             TOTHOURSATTEMPTED,     ' HRS. NC',
             TOTHOURSOFNOCREDIT,     ' GRADE PTS.', TOTGRADEPOINTS,
                 ' HRS. TO GRAD.', TOTHOURSTOWARDGRADUATION)
                 (A,F(6),A,A(3),A,F(5,1),A,F(5,1),A,F(4),A,F(4));
           GO TO NEWCARD;
    IDERROR:  PUT EDIT ('  ID ERROR, UPDATE ID',ID,'  ALPHA',CNAME,'  FILE I
26  D',STUDENTIDNUMBER,'  FILE ALPHA',FNAME)
            (SKIP,A,F(6),A,A(3),A,F(6),A,A(3));
           GO TO NEWCARD;
    DATAERROR:  PUT EDIT ('  DATA ERROR, UPDATE ID',ID,'  ALPHA',CNAME)
27           (SKIP,A,F(6),A,A(3));
           GO TO NEWCARD;
    CHECKNEXTCARD:
           OD = ID;
           ONAME = CNAME;
             GET EDIT (ID,CNAME,DUMMY)  (COLUMN(1),F(5),A(2),A(3));
           IF STUDENTIDNUMBER = ID THEN GO TO UPDATE;
                                       ELSE GO TO CORRECTFILE;
    STOP: END;
```

Figure 76. Updating Sequential Disk File (3 of 3)

```
// JOB CUMPCH
// ASSGN SYS010,X'291'
// DLBL IJSYS10
// EXTENT SYS010,PLID03,1,0,1500,250
// OPTION LINK,SYM,LIST,ERRS,60C,NODECK,LISTX
// EXEC PL/I

    DISK3: PROCEDURE OPTIONS (MAIN);
       DECLARE IJSYS10 FILE INPUT  RECORD
1             BUFFERED
           ENVIRONMENT(MEDIUM(SYS010,2311) F(80) BUFFERS(2));
       DECLARE CUMCARD FILE OUTPUT RECORD
4          BUFFERED
           ENVIRONMENT(MEDIUM(SYSPCH,2540) F(80) BUFFERS(2));
       DECLARE SEMGRADECARDS CHAR (80);
       ON ENDFILE(IJSYS10) GO TO STOP;
       OPEN FILE(IJSYS10),FILE(CUMCARD);
2      READ: READ FILE(IJSYS10) INTO (SEMGRADECARDS);
3      WRITE FILE(CUMCARD) FROM (SEMGRADECARDS);
       GO TO READ;
       STOP: CLOSE FILE(IJSYS10) ,FILE(CUMCARD);
       END;
```

Figure 77. Punching Disk Data Into Cards

The DO WHILE Statement

The program shown in Figure 78 shows how the sum of the series 1 + 1/2 + 1/3 + 1/4 + 1/5 + ... may be computed using the DO WHILE statement.

```
WHILE:  PROCEDURE OPTIONS (MAIN);
SUM=1;
N=1;
DO WHILE (1/N > 1.E-3*SUM);
N=N+1;
SUM=SUM+1/N;
END;
PUT LIST(SUM,N);
END;
```

Figure 78. Example Illustrating The Use of The DO WHILE Statement

Using The Console For Communications

The example in Figure 79 (encircled numbers are used for reference purposes only) shows how the console may be used for communication with a program.

Four files are used: a tape input file (1), a tape output file (2), a card output file (3), and a print file (no declaration, default). The input from the tape input file may be written on tape, punched into cards, or printed. The answer as to what is to be done, must be given by the operator using the console (4). The operator has to type any combination of "PRINT", "COPY", or "PUNCH".

The answer is then scanned by the program to determine first whether COPY 5 , then whether PRINT (6), and finally whether PUNCH (7) has been returned. If the reply that is searched for is not found in ANSWER, 0 is returned by the INDEX built-in function.

Depending on the answer of the operator, the input is written onto tape (8), punched into cards (9), or printed (10).

```
// JOB MON_43
*    PAUSE READY TAPES MASTER ON 180
// ASSGN SYS010,X'182'
// ASSGN SYS011,X'181'
// OPTION LINK, LIST,SYM,ERRS,60C,NODECK,LISTX,DUMP
 ACTION NOMAP
// EXEC PL/I

     UTIL:  PROCEDURE OPTIONS (MAIN);
           DECLARE TAPEIN FILE RECORD SEQUENTIAL INPUT ENVIRONMENT
 (1)       (MEDIUM (SYS010,2400) CONSECUTIVE F(800,80) BUFFERS (2)
           NOLABEL);
           DECLARE TAPEOU  FILE RECORD SEQUENTIAL OUTPUT ENVIRONMENT
 (2)       (MEDIUM (SYS011,2400) CONSECUTIVE F(800,80) BUFFERS (2)
           NOLABEL);
 (3)       DECLARE CARDOU  FILE RECORD SEQUENTIAL OUTPUT ENVIRONMENT
           (MEDIUM (SYSPCH,2540) CONSECUTIVE F(80) BUFFERS (2));
           DECLARE CARD CHAR(80);
           DECLARE ANSWER CHAR(20);
           DECLARE (COPYBIT,PRINTBIT,PUNCHBIT) CHAR(1)   ;
           DECLARE LOGIC CHAR(3);
           ON ENDFILE (TAPEIN) GO TO DONE;
     START:
           ANSWER = ' ';  COPYBIT = ' ';  PRINTBIT=' ';  PUNCHBIT = ' ';
 (4)       DISPLAY ('TYPE CONTROL WORDS - PRINT, PUNCH, COPY- ANY COMBINATION
                   ' ) REPLY (ANSWER);
 (5)       M=INDEX (ANSWER,'COPY');
           IF M ¬= 0 THEN COPYBIT= '1';
 (6)       M=INDEX (ANSWER,'PRINT');
           IF M ¬= 0 THEN PRINTBIT = '1';
 (7)       M=INDEX (ANSWER,'PUNCH');
           IF M ¬= 0 THEN PUNCHBIT = '1';
           ELSE DO;
           LOGIC = COPYBIT || PRINTBIT || PUNCHBIT;
               IF LOGIC ¬= '   ' THEN GO TO OKAY;
               DISPLAY('INVALID RESPONSE, TRY AGAIN');
               GO TO START;
               END;

     OKAY:
           OPEN FILE (TAPEIN)

      IF COPYBIT = '1' THEN
           OPEN FILE (TAPEOU);
      IF PUNCHBIT = '1' THEN
           OPEN FILE (CARDOU);
      LOOP:  READ FILE (TAPEIN) INTO (CARD);
 (8)  IF COPYBIT = '1' THEN
           WRITE FILE (TAPEOU)  FROM (CARD);
 (9)  IF PUNCHBIT = '1' THEN
           WRITE FILE (CARDOU)  FROM (CARD);
 (10) IF PRINTBIT = '1' THEN
           PUT EDIT (CARD) (SKIP,A(80));
           GO TO LOOP;
     DONE:  CLOSE FILE (TAPEIN);
      IF COPYBIT = '1' THEN
           CLOSE FILE (TAPEOU);
      IF PUNCHBIT = '1' THEN
           CLOSE FILE (CARDOU);
           END;
```

Figure 79. Using the Console for Communications

Index

(Where more than one page reference is given, major reference appears first.)

GC24-9005-6

IBM System/360 DOS/TOS PL/I Programmer's Guide GC24-9005-6

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]