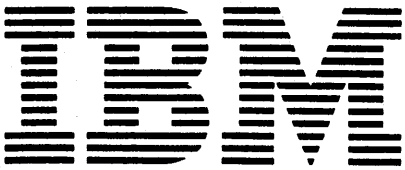
Interlanguage Communication

Facilities are available which allow procedures compiled by the PL/I Optimizing Compiler to communicate at execution time with programs compiled by any IBM COBOL compiler or by the IBM F-level FORTRAN IV compiler for the System/360 Disk Operating System.

Thus, it is possible for existing COBOL and FORTRAN users to write new applications in PL/I while still utilizing existing libraries of COBOL and FORTRAN programs; in addition, existing applications can be modified by the use of PL/I procedures.

Communication between programs written in different languages is specified in the usual way -- by a CALL statement, or alternatively, for FORTRAN and PL/I, a function reference.

The interlanguage communication facilities are requested in the PL/I procedure by the COBOL or FORTRAN option of the OPTIONS attribute or option. The remapping of COBOL structures and transposing of FORTRAN arrays, which would then normally take place, can be completely or partially suppressed by the NOMAP, NOMAPIN, and NOMAPOUT options. The INTER option can be used to specify that the COBOL and FORTRAN interrupts which would otherwise be handled by the system are to be handled by the PL/I interrupt handling facilities.



Program Product

File No. S360-29
Order No. GC33-0004-0

IBM System/360 Disk Operating System

PL/I Optimizing Compiler

General Information

Program Number 5764-PL1

This publication is a planning aid only. It is intended for use prior to the availability of the following IBM System/360 program products:

- DOS PL/I Optimizing Compiler, Program Product 5736-PL1
- DOS PL/I Resident Library, Program Product 5736-LM4
- DOS PL/I Transient Library, Program Product 5736-LM5

Used in conjunction with the program product publication IBM System/360 Disk Operating System PL/I Language Reference Manual, Order No. SC33-0005, this publication enables installation managers, systems analysts, and programmers to plan and write PL/I programs that are to be compiled and executed upon availability of these program products.

Preface

This publication contains a description of the PL/I Optimizing Compiler for the IBM System/360 Disk Operating System.

The subjects covered include the compiler facilities, the optimization features, the operating system environment, and a summary of the PL/I language implemented.

RECOMMENDED PUBLICATIONS

The PL/I language implemented by the DOS PL/I Optimizing Compiler is described in detail in the program product publication:

IBM System/360 Disk Operating System: PL/I Language Reference Manual, Order No. SC33-0005

The following publications contain other information that might be valuable to the PL/I programmer or to a programmer who is learning PL/I:

A PL/I Primer, Order No. SC28-6808

A Guide to PL/I for Commercial Programmers, Order No. SC20-1651

A Guide to PL/I for FORTRAN Users, Order No. SC20-1637

Introduction to the List Processing Facilities of PL/I, Order No. GF20-0015

Introduction to the Compile-Time Facilities of PL/I, Order No. SF20-1689

Preface to PL/I Programming in Scientific Computing, Order No. SE20-0312

First Edition (March,1970)

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

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

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England.

Contents

Tables

Introduction

The DOS PL/I Optimizing Compiler is designed for compilation of efficient object programs. It requires a minimum of 44K bytes on a System/360 with the universal instruction set and is intended to meet the requirements of the PL/I user of medium-sized or larger installations. Good all-round performance is achieved by a new compiler design which incorporates the best design features of a number of well-proven compilers, including the PL/I D, PL/I (F), FORTRAN IV (G), FORTRAN IV (H), and COBOL (F) compilers. Main features include:

- Extensive Optimization.
- Advanced level of PL/I - a comprehensive implementation of PL/I with language extensions beyond both the D-level subset under DOS and the F-level subset under OS. New features for DOS users include:

  Compile-time preprocessing
  Arrays of structures
  DEFAULT statement
  Entry variables
  File variables
  Data-directed transmission

- Compilation Speed - compilation speeds without optimization will be equal to or better than those of existing IBM PL/I, COBOL, and FORTRAN compilers in the same environment.
- Extensive Debugging Aids - time and effort required for program checkout are minimized by:

  Extensive implementation of on-units
  Support of the CHECK condition
  Data-directed output
  Comprehensive range of compiler options
  Clear and precise compile-time and execution-time diagnostic messages
  Optional statement number trace facilities

Optimization is optional; three levels of optimization are available to the programmer:

- Object code optimized to minimize the time required for execution of the object program. A secondary effect may be a reduction in object program size.
- Object code optimized to reduce the storage space required for the object program. A secondary effect may be a reduction in program execution time.
- No optimization, permitting fastest compilation. This is the standard default.

The information in this manual is intended for existing and prospective users of PL/I.

For detailed planning of PL/I programs for the DOS PL/I Optimizing Compiler, the program product publication IBM System/360 Disk Operating System PL/I Language Reference Manual, Order No. SC33-0005, should be used.

The DOS PL/I Optimizing Compiler translates PL/I source statements into machine instructions. However, the compiler does not generate all the machine instructions required to represent the source program; in some cases it inserts references to subroutines that are stored in a resident library or in a transient library.

Subroutines from the resident library are incorporated into the PL/I object program by the DOS linkage editor program.

Subroutines from the transient library are loaded, executed, and discarded during program execution.

The libraries are not an integral part of the DOS PL/I Optimizing Compiler. These libraries consist of the following separate IBM System/360 program products:

- DOS PL/I Resident Library, Program Product 5736-LM4
- DOS PL/I Transient Library, Program Product 5736-LM5

Throughout this publication the terms 'resident library' and 'transient library' refer to these IBM program products.

Chapter 1: The DOS PL/I Optimizing Compiler

The DOS PL/I Optimizing Compiler implements an advanced subset of PL/I which incorporates most of the features implemented by the IBM System/360 Operating System PL/I (F) Compiler (the principal exceptions are multitasking, teleprocessing, REGIONAL(2) data set organization, and sterling pictures). The Optimizing Compiler also incorporates recent extensions to the PL/I language, such as entry variables, file variables, and the DEFAULT statement. The compiler permits efficient use of the advanced language in respect of both compilation and execution-time performance. The optional optimization levels permit the choice of fast compilation, object-program space minimization, or fast execution. Object programs produced with use of the fast compilation option are nevertheless at least as efficient as they would be if compiled by the D Compiler.

Compilation Speed

Performance estimates indicate that compilation speeds without optimization will be equal to or better than those of the IBM FORTRAN E, COBOL D and E, and PL/I D (Version 4) compilers in a 44K partition. If optimization is specified, these compilation times will increase by up to 25% for typical PL/I programs and by up to 100% for some types of program, such as certain types of scientific programs, particularly those with deeply nested DO-loops or array operations. Multiple external procedures can be compiled in the same step to save time used for overheads.

Execution Speed

The Optimizing Compiler produces extremely efficient object code. Estimates indicate that the code produced when optimization for speed is requested will be equal to or better than that produced by the existing COBOL, FORTRAN, or PL/I compilers under DOS. The execution speed of a sample set of PL/I programs compiled by the Optimizing Compiler when optimization for speed is requested is estimated to be 3.3 times that of the same PL/I programs when compiled by the D Compiler. However, the amount of improvement for any particular program will depend on factors such as the overheads for input/output and the nature of the internal processing of the program. Whereas the execution speed of some programs could improve substantially others may show only slight improvements. Programs most likely to show appreciable improvements in execution speed are those that perform extensive loop and array operations and those that make extensive use of edit-directed input/output.

Object Program Space Requirements

Because of increased housekeeping requirements, small object programs compiled by the Optimizing Compiler may require up to 2K bytes more storage than would be required by the D Compiler. The reduced amount of object code generated by the Optimizing Compiler, and the increased modularity of the library subroutines, however, diminishes the overall storage requirement for larger programs; programs larger than approximately 400 statements will require less storage than they would if compiled by the D Compiler.

Debugging Aids

The PL/I Optimizing Compiler provides the following debugging aids to minimize the time and effort needed for program checkout.

Diagnostics: Comprehensive diagnostic messages are provided at both compile time and execution time. In processing an erroneous statement, the compiler may attempt to correct the error by making an assumption about the intention of the statement. Messages produced at compile-time will be for errors in the categories of unrecoverable, severe, error, warning, and informatory. The messages will be listed in category groups in statement number order. Each message will indicate the number of the erroneous statement and, when applicable, the part of the statement involved, the exact nature of the error, and any assumptions made and action taken by the compiler.

On-Units: Program condition-handling is facilitated by the provision of on-units for all PL/I conditions. On-units permit either programmer-defined or system-defined

action to take place when a particular interrupt occurs during execution of a PL/I program. The range of conditions covers all arithmetic interrupts, input/output errors, and special program-checkout conditions such as CHECK, SUBSCRIPTRANGE, STRINGRANGE, SIZE, and STRINGSIZE.

Statement Number Trace: The statement number trace facility is used optionally to produce a record of the PL/I source statements executed prior to the occurrence of an interrupt. The trace is produced as part of the standard system action for any condition, and as part of the diagnostic output produced by use of the SNAP option. It contains a list of the statement numbers of PL/I source statements executed up to the point of interrupt. The number of statements included in the trace is given by the user in requesting the option. This feature is in addition to the optional inclusion of the statement number in diagnostic messages produced at execution time.

Compiler Options

A number of compiler options are available for use by the programmer to specify information or to request optional compiler facilities.

The options perform the following functions:

- Specify the amount of main storage for use by the compiler during compilation.
- Specify whether the source program is coded in the PL/I 48-character set or 60-character set.
- Specify whether the source program is presented to the compiler in BCD or EBCDIC format.
- Request the compile-time preprocessor facilities of the compiler.
- Define the portion of a source program input record that contains PL/I statements.
- Request conditional compilation to follow the use of the preprocessor depending on the severity of errors found during preprocessing.
- Specify the type of optimization to be performed by the compiler. Optimization can be requested either to obtain fast execution of object programs, or to minimize the size of object programs. The time taken for compilation is increased by the use of either type of optimization.
- Request the compiler to produce additional code to insert the relevant source statement number in each execution-time message.
- Request printed listings of the following:

  Attribute Table
  Cross-Reference Table
  External Symbol Dictionary
  Object Module
  Source Program
  Block and DO-group nesting levels
  Preprocessor Input
  Warning and Error Messages
  Compiler options used in the compilation

- Specify the number of lines on each page of listing.
- Request the compiler to place the object module on the symbolic device SYSPCH or SYSLNK.
- Request the preprocessor to place its output on the symbolic device SYSPCH.
- Specify the phase-name of the object module.

Compatibility with the DOS PL/I D Compiler

Source programs written for the PL/I D Compiler can be compiled by the DOS PL/I Optimizing Compiler and will execute correctly without modification, except for a few minor differences which exist between these implementations of the language. These incompatibilities are described in Appendix B.

Object modules that have been compiled by the DOS PL/I D Compiler, and library subroutine modules from the DOS PL/I D Compiler library, cannot be incorporated into programs compiled by the PL/I Optimizing Compiler. This problem can be overcome by recompilation of all PL/I source modules with the Optimizing compiler (and by ensuring that all library subroutines used are called from the resident or transient libraries as appropriate).

Chapter 2: Optimization

The main aim of the Optimizing Compiler is to generate object programs which execute as fast as possible and which occupy as little space as possible during execution. In many cases this will involve generating efficient code for statements in the sequence written by the programmer; in other cases, however, the compiler may alter the sequence of statements or operations to improve the performance while producing the same result.

The following types of optimization are carried out by the compiler:

- Common expression elimination.
- Transfer of invariant statements and expressions out of loops.
- Redundant expression elimination, including minimizing the number of load and store operations, and non-execution of redundant logical statements.
- Simplification of expressions into forms that can be more readily optimized.
- Initialization of arrays and structures.
- In-line code for most conversions.
- Reduction of key conversions for REGIONAL data sets.
- Matching format lists with data lists at compile-time.
- In-line code for string manipulation.
- In-line code for many built-in functions.
- Special-case code for DO statements.
- Structure assignments.
- Register and address optimization, including maintenance of values in registers for as long as possible, and producing efficient address arithmetic based on optimal flow-paths.
- Keeping program branches as much as possible within the scope of the same base address.
- Packaging library routines into logical units to minimize space requirements.
- Elimination of common constants and program control data to minimize space usage.

Common Expression Elimination

An expression can occur twice or more in such a way that the flow of control always passes through the first occurrence of the expression to reach a subsequent occurrence. If the second and any subsequent evaluation produces a result identical to the result produced by the first evaluation, the subsequent expressions are termed common expressions. The compiler eliminates common expressions by saving the value of the first occurrence of the expression either in a temporary (compiler-generated) variable, or in the program variable to which the result of the expression is assigned. For example:

```
X1 = A1 * B1;
.
.
.
Y1 = A1 * B1;
```

Provided that the values of A1 and B1 do not change between the execution of these statements, the statements can be optimized to the equivalent of the following PL/I statements:

```
X1 = A1 * B1;
.
.
.
Y1 = X1;
```

If the first occurrence of the common expression involves the assignment of the value to a variable that is modified prior to the occurrence of the later expression, the value is assigned to a temporary variable. The example given above would become:

```
TEMP = A1 * B1;
X1 = TEMP;
.
.
.
Y1 = TEMP;
```

Also, if the common expression occurs as a subexpression within a larger expression, a temporary variable is created to hold the value of the common subexpression. For

example, in the expression C1 + A1 * B1 a temporary variable would be created to hold the value of A1 * B1 if this were a common subexpression.

An important application of this technique occurs in statements containing subscripted variables where the same subscript value is used for each variable. For example:

```
PAYROLL_TAX(MANNO) = PAY_CODE(MANNO) *
   WEEKPMNT(MANNO);
```

The value of the subscript expression MANNO is computed only once when the statement is executed (the computation would involve the conversion of a value from decimal to binary if MANNO were declared a decimal variable).

Transfer of Expressions from Loops

Where it is possible to produce error-free execution without affecting the results of a program, the optimization process moves expressions from inside a loop to a point outside which immediately precedes it. A loop can be either a DO loop or a loop in a program which can be detected by the analysis of the flow of control within the program. Expressions that can be transferred are normally expressions which return the same value for every iteration of a particular loop. For example:

```
DO I =  1 TO N;
B(I) = C(I) * SQRT(N);
P = N * J;
END;
```

This loop can be optimized to produce object code corresponding to the following statements:

```
TEMP = SQRT(N);
P = N * J;
DO I = 1 TO N;
B(I) = C(I) * TEMP;
END;
```

Note that the assignment statement P = N * J; can also be moved out of the loop to a point preceding it.

If the programmer wishes this type of optimization to be carried out, he must specify the optimization option REORDER on a BEGIN or PROCEDURE block which contains the loop. If the option is not specified, the default option, ORDER, is assumed and the optimization is inhibited.

Programming Considerations:

1. The transfer of expressions from inside a loop is performed on the assumption that every expression in the loop is executed more frequently than expressions immediately outside the loop. Occasionally this assumption fails, and expressions can be moved out of loops to positions where they are executed more frequently than they would have been if they had remained inside the loop. For example:

   ```
   DO I = J TO K WHILE (X(I) = 0);
   X(I) = Y(I) * SQRT(N);
   END;
   ```

   The expression SQRT(N) can be moved out of the loop to a position where it is possible for it to be executed more frequently than it would be in its original position inside the loop. This undesired effect of optimization can be prevented by the use of the ORDER option for the block in which the loop occurs.

2. Loops are detected by a flow-analysis process. This process can fail to recognize a loop, owing to the existence of flowpaths which the programmer knows will never be used. For example, the use of label variables can inadvertently cause optimization to be inhibited by making the recognition of a desired loop impossible.

Redundant Expression Elimination

A redundant expression is an expression that need not be evaluated in order that a program be executed correctly. For example, the logical expression (A=D)|(C=D) contains the subexpressions (A=D) and (C=D), the second of which need not be evaluated if the first is true. The effect of this optimization is to make the use of logical expressions in IF statements more efficient than a series of nested IF statements.

Simplification of Expressions

Simplification of an expression involves its conversion into a form which can be translated into more efficient object code. Where possible, multiplication and division operations are converted into addition and subtraction operations that can be performed by faster machine instructions.

For example:

```
DO I = 1 TO N BY 2;
   .
   .
   .
X = I * 4;
   .
   .
   .
END;
```

When optimized, the statements in this loop are converted into object code which is equivalent to the following PL/I statements:

```
I = 1;          /* LOOP INITIALIZATION */
IF I > N THEN GO TO F;
TEMP = 4 * I;
G:              /* LOOP ENTRY POINT */
   .
   .
   .
X = TEMP;       /* EXPRESSION AFTER */
                  /* SIMPLIFICATION */
   .
   .
   .
I = I + 2;
TEMP=TEMP + 8;
IF I < N THEN GO TO G;
                /* END-OF-LOOP TEST */
F:               /* LOOP EXIT POINT */
   .
   .
   .
```

Modification of Loop Control Variables

Where possible, the expression-simplification process will modify both the control variable and the iteration specification of a DO-loop to achieve more efficient processing when the control variable is used as a subscript. The calculation of addresses of array elements can be made faster by replacing multiplication operations by addition operations. For example, the loop:

```
DO I = 1 TO N BY 1;
A(I) = B(I);
END;
```

causes N element values from array B to be assigned to corresponding elements in array A. On the assumption that each element is 4 bytes in length, the address calculations which are used for each iteration of the loop are:

```
Base(A) + (4*I)    for array A, and
Base(B) + (4*I)    for array B,
```

where 'Base' represents the base address of the array in storage. The repeated multiplication of the control variable by a constant representing the length of an element can be converted to faster addition operations. The optimized DO statement above is converted into object code equivalent to this PL/I statement:

```
DO TEMP = 4 BY 4 TO 4*N;
```

The element address calculations are converted to the equivalent of:

```
Base(A) + TEMP    for array A, and
Base(B) + TEMP    for array B.
```

Note that a loop control variable and its iteration specification can be optimized only when the control variable used as a subscript is incremented by a constant, or by a variable the value of which is not reset during the execution of the loop, and if the value of the control variable is not required outside the loop in which it is specified.

Defactorization

Whenever possible, a constant in an array subscript expression is used in an offset in the address calculation. For example, the address of a four-byte element A(I+10) would be calculated as (BASE(A)+4*10)+I*4.

Replacement of Constant Expressions

The expression-simplification process replaces constant expressions with the equivalent constant. For example the expression 2+5 is replaced by 7.

Replacement of Constant Multipliers and Exponents

The expression-simplification process replaces certain constant multipliers and exponents. For example, A*2 becomes A+A, and A**2 becomes A*A.

Initialization of Arrays

When arrays which have the BASED, AUTOMATIC, or CONTROLLED storage class are to be initialized by a constant specified

in the INITIAL attribute, the first element of the variable is initialized by the constant, and the remainder of the initialization consists of a single move which propagates the value through all the elements of the variable. For example an array declared as:

```
DECLARE A(20,20) FIXED BINARY
   INITIAL((400)0);
```

would be initialized in this way.

In-Line Code for Conversions

Most conversions are performed by in-line code, rather than by calls to the PL/I resident library. The exceptions are:

conversions between character and arithmetic data

conversions from numeric character (PICTURE) data where the picture includes characters other than 9, V, or a single sign or currency character.

conversions to numeric character (PICTURE) data where the picture includes scale factors or floating point picture characters.

Note that conversions to 'ZZ9V99' will be done in-line.

Key Handling for REGIONAL Data Sets

In certain circumstances, key handling for REGIONAL data sets is simplified by avoiding unnecessary conversions between fixed binary and character-string data types, as follows:

REGIONAL(1): If the key is supplied as a fixed binary integer, there is no conversion from fixed binary to character-string and back again.

REGIONAL(3): If the key is supplied in the form K||I, where K is a character string and I is fixed binary with precision (13,0), the rightmost eight characters of the resultant string are not reconverted to fixed binary. (This conversion would otherwise be necessary in order to obtain the region number.)

Matching Format Lists with Data Lists

Whenever possible, i.e. where neither the format list nor the data list contains expressions the values of which are unknown at compile time, items specified in format lists and data lists in edit-directed input/output statements are matched at compile time. This permits conversion to or from the data list item to be performed by in-line code where possible. Also, on input the item can be taken directly from the buffer or on output placed directly in the buffer, thus eliminating library calls except when it is necessary to transmit a block of data between the input/output device and the buffer.

```
DCL (A,B,X,Y,Z) CHAR(25);
GET FILE(SYSIN) EDIT (X,Y,Z) (A(25));
PUT FILE(SYSPRINT) EDIT (A,B) (A);
```

In the above example, format list matching is performed at compile time; hence at execution time library calls will be required only when the buffer contents are to be transmitted to or from the input/output device.

In-Line Code for String Manipulation

Operations on character strings such as concatenation and assignment of adjustable, varying-length, and fixed-length strings are performed in-line. In-line code is also generated for many cases of aligned bit strings.

In-Line Code for Built-in Functions

Many built-in functions are executed by in-line code. INDEX and SUBSTR are examples of functions for which in-line code is usually generated. TRANSLATE, VERIFY, and REPEAT are examples where in-line code is generated for simple cases.

Special Case Code for DO Statements

Wherever possible, the Optimizing Compiler will generate code for DO-loops in which the value of the control variable and the values used in the iteration specification are held in registers throughout execution

of the loop. For example, the compiler will attempt to maintain in registers the values of the variables I, K, and L in the following statement:

```
DO I = A TO K BY L;
```

This form of optimization permits the most efficient loop control instructions to be used.

Structure and Array Assignments

Structure and array assignment statements are implemented by single move instructions whenever possible. Otherwise the assignment is performed by the simplest loop possible for the operands specified in the assignment. For example:

```
DCL A(10), B(10), 1 S(10), 2 T, 2 U;

1.  A=B;

2.  A=T;
```

The first assignment will be implemented by a single move instruction, while the second will be implemented by a loop since array T is interleaved with array U, thereby making a single move impossible.

Library Routines

The PL/I resident and transient library routines used by the Optimizing Compiler have been designed as a set of modules containing logically-related functions such that each function in a particular module is likely to be required in the same object program. Thus the link-edited object program will contain only code necessary for the functions used in that program.

The groups of functions particularly concerned with this efficient structuring include record-oriented input/output, stream-oriented input/output, conversions, and error handling.

Elimination of Common Constants and Common Program Control Information

If a constant is used more than once in a program, a single copy of that constant is kept. For example:

```
WEEK_NO = WEEK_NO + 1;
RECORD_COUNT = RECORD_COUNT + 1;
```

Then the 1 is stored only once.

The compiler generates control information to describe certain program elements such as arrays. If there are two or more similar arrays, then this descriptive information is generated once only.

Use of Registers

More efficient execution of loops can be achieved by maintaining in registers the values of variables which are subject to frequent modification during the execution of the loops. When error-free execution permits, values can be kept in registers, and considerable efficiency can be achieved by dispensing with time-consuming load-and-store operations to reset the values of variables in their storage locations. If the latest value of a variable is required after a loop has been executed, the value is assigned to the storage location of the variable when control passes out of the loop.

Register allocation can be more significantly optimized if REORDER is specified for the block. However, the values of variables that are reset in the block are not guaranteed to be the latest assigned values when a computational interrupt occurs, since the latest value of a variable may be present in a register but not in the storage location of the variable. If ORDER is specified, optimization of register allocation is impeded by the requirement that all values of variables reset in the block are guaranteed, and must therefore be assigned immediately to the storage locations of their respective variables.

Minimization of Code for Program Branches

The base registers for branch instructions in the object program are allocated in accordance with the logical structure of the program. This ensures that the load instructions required for program addressing do not occur in the middle of deeply nested loops.

Also, the branch instructions generated for IF statements are arranged by the compiler to be as efficient as possible. For example, a statement such as:

```
IF condition THEN GOTO label;
```

is defined by the PL/I language as being a test of the condition followed by a branch on *false* to the statement following the THEN clause. However, when the THEN clause consists only of a GOTO statement, the statement is compiled as a branch on *true* to the label specified in the THEN clause.

Chapter 3: The PL/I Language Implemented

Introduction

The language implemented by the DOS PL/I Optimizing Compiler is an advanced subset of PL/I.

With one minor exception (sterling pictures), this subset includes all the language implemented by the DOS PL/I D Compiler. Appendix A is a complete list of the keywords implemented.

In general, source programs written for the D Compiler can be compiled and executed succesfully by the Optimizing Compiler without amendment. However, there are some minor incompatibilities; these are listed in Appendix B.

Language Features

The language implemented by the DOS PL/I Optimizing Compiler contains the following features:

1. Data Types:
   - Character and bit string.
   - Fixed-point binary and decimal.
   - Floating-point binary and decimal.
   - Character and numeric picture.
   - Real and complex arithmetic.
   - Label and entry.
   - File.
   - Event.
   - Pointer and offset.
   - Automatic, static, controlled, and based storage classes for data.
   - External and internal scope for data.
2. Data Manipulation:
   - Assignment, with automatic conversion if necessary, between data variables.
   - Element, array, and structure expressions.
   - Comparison, logical (boolean), and string-manipulation operators.
   - Built-in functions for mathematical and arithmetic computation, string manipulation, based and controlled storage manipulation, and error handling.
   - Pseudo-variables for computation and error handling.
   - Programmer-defined functions.
3. Data Aggregates:
   - Arrays of data elements.
   - Structures of data elements.
   - Arrays of structures.
   - Areas.
4. Program Block Structure:
   - Separate compilation of external procedures of the same program.
   - Block structuring to establish scope of identifiers and permit dynamic storage allocation of automatic data.
   - Recursive invocations of a procedure with stacking and unstacking of generations of automatic data to preserve and reestablish the environment of each invocation.
5. Input and Output:
   - Stream-oriented input/output with automatic conversion to and from internal and external forms of data representation.
   - Record-oriented input/output with both move and locate modes of operation.
   - Sequential and direct-access processing modes.
   - Data transmission overlap with internal processing.

Comparison with the PL/I D-Level Subset Language

In general, a PL/I program written for the DOS PL/I D Compiler will produce identical results when compiled by the DOS PL/I Optimizing Compiler. However, the DOS PL/I Optimizing Compiler contains many additional language facilities as well as some differences in the way common language is implemented by the DOS PL/I D Compiler; these are described below.

IMPLEMENTATION DIFFERENCES

The DOS PL/I Optimizing Compiler imposes fewer constraints on the programmer than the D Compiler. The restrictions that do apply to the Optimizing Compiler are listed in Table 1, together with the corresponding limits for the D Compiler.

LANGUAGE EXTENSIONS

Language features implemented by the DOS PL/I Optimizing Compiler which are only partially implemented or not implemented at all by the DOS PL/I D Compiler are described in outline below.

Optimization

The DOS PL/I Optimizing Compiler carries out extensive optimization of PL/I programs. The degree of optimization is controlled by selection of the appropriate compiler option and the use within the PL/I program of the ORDER and REORDER options for program blocks.

Optimization is discussed in Chapter 2.

DEFAULT Statement

The DEFAULT statement enables the programmer to define default attributes for identifiers. The DEFAULT statement can override PL/I language or implementation default attributes and precisions.

Compile-Time Processing

The compile-time preprocessor has two functions:

1. Modification of source programs prior to compilation (e.g., translation of non-English or other local keywords into the equivalent PL/I keywords).
2. Inclusion of source statements previously stored in the source statement library or a private library (%INCLUDE statement).

Storage Control

Based Storage: Based variables are used for locate-mode input/output and for list-processing applications. They can be allocated separately or within a predefined area of storage identified as an area variable. An area variable, declared with the AREA attribute, can itself be a based variable, and can be assigned to another area variable or transmitted as a record in record-oriented input/output statements. Several allocations of the same based variable can exist at the same time, each allocation being identified by a locator variable. There are two types of locator variables: pointer variables (declared with the POINTER attribute) and offset variables (declared with the OFFSET attribute). The value of a pointer variable identifies a main storage location, and the value of an offset variable identifies a storage location within an area variable relative to the start of the area. The values of offset variables remain valid when transmitted with the associated area to or from external storage, but the values of pointer variables do not, since they refer to absolute storage locations.

A particular allocation of a based variable is identified by the use of the locator qualifier (->) to associate the relevant pointer or offset with the name of the based variable. For example, PTR1->XRAY refers to that allocation of XRAY identified by the value of the pointer variable PTR1.

Self-Defining BASED Structures: Based structures with adjustable string lengths, area sizes, or array dimensions can be declared as self-defining structures by means of the REFER option. Self-defining structures transmitted as variable-length records can be read into storage from an external storage device by execution of a READ statement with the SET option. The

Table 1. Differences in Implementation Restrictions (Part 1 of 2)

Feature	PL/I D Compiler	PL/I Optimizing Compiler
Arrays, max.number per compilation	32	no limit
Arrays, max.number of dimensions	3	15
Arrays, lower bound	always 1	range: -32768 to 32767
Structures	8 levels	15 levels
Aggregates (arrays and structures)	max.size:32767 bytes	no limit
Floating-point data	short or long precision decimal and binary	short or long precision decimal, binary, and numeric character
Character strings	max. length: 255 bytes	max. length: 32,767 bytes.
Character-string constants	max.length: 255 bytes	max.length: 1000 bytes before application of repetition factors
Bit strings	max.length: 64 bits	max.length: 32,767 bits
Bit-string constants	max.length: 64 bits	max.length: 8000 bits (1000 bytes) before application of repetition factors
Arguments and parameters	max.number: 12	max.number: 64
PAGESIZE	maximum: 255 lines	maximum: 32767 lines
INITIAL attribute: nested iterations	max.number: 8	max.number: 50 (including factored attributes)
Attribute factorization: nesting levels	max.number: 8	max.number: 16
Block nesting	max.depth: 3	max.depth: 50[1]
Blocks: number in compilation	max.number: 63	max.number: 255
Source statement margins	positions 2 to 72	positions 1 to 100
Storage limitations: STATIC INTERNAL and AUTOMATIC	maximum: 64K bytes	no fixed limit
DISPLAY string length	maximum: 80 bytes	72 bytes
REPLY string length	maximum: 256 bytes	72 bytes

[1] There is a limit on the depth of block nesting imposed by the length of the block labels given by the user. If the average length of the labels exceeds eight characters, the maximum number of available nesting levels is reduced.

Table 1. Differences in Implementation Restrictions (Part 2 of 2)

Feature	PL/I D Compiler	PL/I Optimizing Compiler
String repetition factor maximum number of characters:	3	15
maximum value of factor:	255	32,767
Maximum statement length (not including DECLARE statement) (maximum statement length is function of internal storage available for a compilation)	at least: 230 identifiers, constants and delimiters	at least: 1012 significant characters
Iteration factors: maximum level of nesting	in a format list: 2 in INITIAL attribute: 8	20

use of the REFER option enables the compiler to map the structures in the buffer storage identified by the pointer variable named in the SET option.

Processing Based and Locator Variables: Pointer and offset variables can be set to null values by the NULL built-in function. Allocations of based variables can be freed by the FREE statement; all allocations of based variables within a single area can be freed by the EMPTY built-in function. List processing is the technique by which a number of allocations of a based variable can be manipulated in internal storage by means of pointer and offset qualification and assignment, the NULL and EMPTY built-in functions, and the FREE statement.

Controlled Storage: Data variables can be declared with the CONTROLLED attribute. Such declarations do not cause storage to be reserved for the variable when the block is entered, but are used to map the elements of the variable when storage is obtained for it by execution of an ALLOCATE statement. Attributes given in an ALLOCATE statement override any conflicting attributes given in the declaration of the variable to be allocated. Storage allocated for a controlled variable is released either by execution of a FREE statement, or by termination of the program. Successive allocations of storage are permitted for the same controlled variable. Each current allocation is stacked when a new allocation is created, so that a reference to a controlled variable is a reference to its value in the latest allocation. When the current allocation is freed, the previously stacked allocation, if any, becomes the current allocation. Thus, unlike based storage, only one generation of controlled storage is available at any one time. The ALLOCATION built-in function is used to determine whether any allocations exist for a controlled variable, and if so, how many there are.

Stream Input/Output

Data-Directed Input/Output: An additional mode of stream-oriented input/output is available. This is the data-directed mode. The option DATA is used in a GET or PUT statement when this mode is used. Data-directed input/output causes values to be transmitted in a form similar to PL/I assignment statements. The name of the variable, the assignment symbol (=), and the value are transmitted. On input the value is assigned to the storage location of the given target identifier; on output the identifier and its current value are transmitted.

Edit-Directed Input/Output: The full edit-directed input/output facilities of the PL/I language are implemented.

COPY Option: Stream-oriented input statements can include the COPY option in order to transmit data from the input file automatically to SYSPRINT.

LINESIZE Option: Stream-oriented output files can be opened with a linesize which, in general, represents the size of a logical record on the stream data set.

Expressions in Format Lists: Field widths and repetition factors in repetitive specifications in format lists for edit-directed input/output can be specified as expressions to be evaluated when the input/output statement is executed.

Record Input/Output

IGNORE Option: The IGNORE option can be specified in an input statement when one or more records in the data set are to be skipped.

EVENT Option: The transmission of a record to or from an unbuffered file can be overlapped with the execution of following statements. Such transmission statements must specify the EVENT option to designate a transmission of a record as an event to be associated with the event variable named in the option. The event variable is set active when the transmission operation starts, and inactive after a WAIT statement (and any on-units) have been executed. The WAIT statement synchronizes execution of logically-following statements with the completion of the event, by delaying any further execution until the event associated with the event variable named in the WAIT statement is completed. The COMPLETION built-in function can be used to test the completion value of an event variable. The STATUS built-in function can be used to test the status value of an event variable. Both COMPLETION and STATUS can be used as pseudo-variables.

REGIONAL Data Sets: REGIONAL data sets can be processed sequentially.

Expressions in Options of the ENVIRONMENT Attribute: Any of the environment options which require a numeric value to complete the specification, such as the maximum-record-length (but not the number of buffers or EXTENTNUMBER) can have this value specified as a decimal integer constant or a STATIC FIXED BINARY variable of precision (31,0). The variable, if used, must be assigned a value before the file is opened.

File Variables

A file variable is an identifier that either is declared FILE in an array or structure or is declared FILE VARIABLE. A file variable can represent file name values (file constants). Only a file constant can, however, be declared with any of the additive or alternative file attributes.

Data Aggregates

LIKE Attribute: The LIKE attribute is used to simplify the declaration of similar structures. The declaration of an identifier with the LIKE attribute is copied from that of the identifier named in the attribute; the declarations are thus identical in all respects apart from dimensions and storage class attributes, and the structure names. Note that the use of a DEFAULT statement might cause different default attributes to be applied to the elements of the two structures.

Arrays of Structures: The PL/I D Compiler does not permit the use of arrays of structures, although they can be simulated by the use of based variables. The PL/I Optimizing Compiler has no such restriction. For example:

```
DECLARE 1 INREC,
          2 CUSTOMER CHAR(20),
          2 ADDRESS CHAR(50),
          2 ITEM (20),
            3 STOCK_# CHAR(5),
            3 PRICE FIXED DECIMAL (6,2),
            3 QUANTITY FIXED DECIMAL
              (3);
```

ITEM is an array of structures contained in the structure INREC.

Adjustable Extents for Arrays: Arrays, and arrays of structures, can have their extents specified as expressions to be evaluated when storage is allocated for them. If the adjustable array or array of structures is a parameter, the extents should be specified as asterisks if they are to match the extents of any corresponding argument.

Array Built-In Functions: Additional built-in functions for array manipulation are provided. The POLY built-in function forms a polynomial from two one-dimensional arrays. The DIM built-in function finds the current extent for a specified dimension of an array. The HBOUND and LBOUND built-in functions find the current upper and lower bounds of a specified dimension of an array.

Array Subscript Checking

One condition is provided to detect an array-handling error. SUBSCRIPTRANGE is raised, if enabled, whenever an array reference uses a subscript that is outside the current range of subscript values for the array. This condition checks an attempt to assign a value to storage not allocated for the array.

Conversions

Character-to-arithmetic (when valid) and arithmetic-to-character conversions are permitted. Most conversions are performed in-line.

String Handling

Varying-Length Strings: Character and bit strings can be declared with the VARYING attribute as varying-length strings. A varying-length string has two length values: the declared maximum length and the current length. The current length can vary from zero to the maximum length during execution of the program. The LENGTH built-in function can be used to obtain the current length of a varying-length string. The data which forms the current length of a varying-length string can be transmitted as a V-format record by a record-oriented input/output statement.

Adjustable Strings: Automatic, based, and controlled strings can be declared with adjustable lengths. The string length is declared as an expression that is evaluated when storage is allocated at execution time. The length of a string parameter should be declared as an asterisk if the length is to match that of any argument to be passed to it.

String-Handling Built-In Functions: In addition to the LENGTH built-in function, two other string-handling functions are available: TRANSLATE and VERIFY. TRANSLATE permits translation of a string according to a translation table; VERIFY compares two strings to verify that each character or bit in the first is represented in the second string. The SUBSTR built-in function and pseudo-variable are implemented in full. All three arguments of SUBSTR can be expressions.

STRING Pseudo-Variable: The STRING pseudo-variable permits assignment of a string, portion by portion, into successive elements of an aggregate until either the entire string has been assigned, or until a value has been assigned to all the elements of the aggregate.

String-Handling Conditions: Two conditions are provided to detect string-handling errors: STRINGSIZE and STRINGRANGE. STRINGSIZE is raised when a string is assigned to a string with a shorter maximum length thereby causing truncation of the bits or characters which cannot be accommodated; STRINGRANGE is raised when the string built-in function SUBSTR is used with arguments which do not comply with the rules applying to range for this built-in function.

Program Checkout and Error Control

On-Units: An on-unit may be either a single statement or a begin block. Any statement can be used in a single statement on-unit. After execution of an on-unit, control is returned to a point defined for each condition unless transferred elsewhere by a GO TO statement.

FINISH Condition: The FINISH condition is always raised during the execution of a STOP statement or an END or RETURN statement that causes termination of the program. An abnormal return from a FINISH condition will permit the program to continue.

Condition Built-In Functions and Pseudo-Variables: Certain built-in functions and pseudo-variables are available only for use within an on-unit, for error detection and correction. These are: ONCHAR and ONSOURCE for CONVERSION conditions; DATAFIELD, used to examine data involved with the NAME condition; ONCODE, for use in any on-unit to determine the actual cause of the interrupt; ONCOUNT, used in any on-unit associated with abnormal completion of an input/output operation to determine the number of interrupts that remain to be handled; ONKEY, used to extract the key of a keyed record which caused a condition to be raised; ONLOC, used to give the entry point of the procedure in which the condition was raised. The CONDITION condition can be used to establish programmer-defined condition names; such conditions are raised only on execution of a SIGNAL statement.

Program Checkout: The CHECK condition is used to cause an interrupt either when control passes through a statement label, or for each assignment to a variable, where the labels and variables are identified in the CHECK name list. The standard system action for such an interrupt is to transmit to SYSPRINT both the name of the identifier which caused the CHECK condition to be raised, and, if it is not a program control or locator variable, its new value in data-directed format, and then to continue execution. Programmer-defined action may be specified as an alternative in an on-unit.

The PUT DATA statement (without a data list) causes the names and current values of all variables known in the block to be

transmitted in data-directed format on the specified file.

The SNAP option of the ON statement is used to list a trace of all procedures that are active when the interrupt occurs. It will also provide a list of the numbers of the statements that have been executed if the appropriate compiler option was specified.

Data Attributes

Defined Storage: A variable can be defined on the storage occupied by another variable by use of the DEFINED attribute in the declaration of the defined variable. Simple defining is in effect when the base variable has attributes that match those of the defined variable. String overlay defining is in effect when a defined character-string or bit-string variable is declared with the POSITION attribute. (The defined item and the base must be of the same class, either character or bit. The POSITION attribute specifies the character or bit of the base variable that corresponds to the first character or bit of the defined variable.) iSUB defining is in effect when the bounds of a defined array are specified by expressions that include iSUB variables; the iSUB expressions establish a relationship between the elements of the defined array and those of the base array, such that a reference to an element of the defined array is, in effect, a reference to the corresponding element of the base array.

CONNECTED Attribute: Parameters which represent data held in a contiguous area of storage can have the CONNECTED attribute. Such parameters can be transmitted by record-oriented transmission statements, or be used as the base in string overlay defining.

INITIAL Attribute: The initial value specified with an INITIAL attribute can be represented by an expression to be evaluated when storage is allocated for the variable. The INITIAL attribute cannot be used with an expression for STATIC variables.

Arithmetic Data Attributes

Complex and Real Arithmetic Data: Two modes of arithmetic data are implemented: REAL and COMPLEX. The REAL attribute specifies that the arithmetic variable is to represent real numbers. The COMPLEX attribute specifies that the arithmetic variable consists of two parts, one representing a real number and the other an imaginary number. Associated with the use of complex arithmetic data are the C format item for edit-directed input/output, and the COMPLEX, REAL, IMAG, and CONJ built-in functions and pseudo-variables.

Scale Factor for Fixed-Point Binary Numbers: Arithmetic variables declared with the attributes FIXED BINARY can also be declared with scale factors that permit binary fractions and an implied binary point.

Subroutines and Functions

ENTRY Attribute: Calls to internal procedures have been simplified by automatic conversion of arguments to the parameter type if necessary. Calls to external procedures have been simplified by full implementation of the explicit ENTRY declaration, which permits the same parameter-argument matching process to be carried out as for internal procedures.

Entry Variables: This compiler supports the use of entry variables. An entry variable represents entry-name values. It is declared with the ENTRY and VARIABLE attributes. By assigning different entry-name values to an entry variable, the programmer can use the same procedure reference to invoke different entry points.

GENERIC Attribute: The GENERIC attribute defines an identifier as representing a family of entry points to one or more procedures. Each entry point in a family is given in the declaration of the generic name, and is qualified by a generic descriptor in a WHEN clause. The choice of a particular entry point is made according to the number of arguments, and their attributes, that appear in a reference to the generic name. The argument list is compared with each generic descriptor in turn until a matching generic descriptor is found; the associated entry point is then invoked.

Recursive Invocation of PL/I Procedures: A PL/I procedure with the RECURSIVE option can be reinvoked while it is still active. Reinvocation can be from within the procedure or from an external procedure

that was invoked by the recursive procedure. When a procedure is invoked recursively, the environment of the invoking procedure, including the values of automatic variables, is preserved. The preserved environment of a particular activation of a recursive procedure is restored when control is returned to that activation of the procedure.

Mathematical Built-In Functions

Two additional mathematical built-in functions are implemented. The ASIN built-in function returns the arc sine as a value expressed in radians. The ACOS built-in function returns the arc cosine as a value expressed in radians.

Chapter 4: System Requirements

Machine Requirements

Compilation

The minimum machine requirement for the PL/I Optimizing Compiler under the control of the DOS Supervisor program is a 64K byte IBM System/360 of which at least 44K bytes of main storage must be available to the compiler. If more than 44K is available, the compiler will make use of the additional space to improve compilation speed. The central processing unit of the machine must have the decimal and floating-point instruction sets. If the time taken for compilations is to be printed out, the central processing unit must have the timer feature, and use of this feature must be specified at system generation.

Execution

The machine requirement for the execution of a PL/I object program compiled by the DOS PL/I Optimizing Compiler depends on the size of the object program, although the minimum machine size for the Disk Operating System is 16K. The Disk Operating System has overlay facilities for handling segmented object programs, enabling them to be executed on relatively small machines or in relatively small partitions. Each DOS installation will give guidance for the optimum phase size according to local considerations for machine or partition size and operational efficiency.

The machine must have the decimal and floating-point instruction sets. If timing information is required, the machine must have the timer feature, and use of this feature must be specified at system generation. (If the DELAY statement is used, the ability to handle the interval timer from the application program must also be specified at system generation.) If the DATE built-in function is used, the DOS Supervisor should have date facilities incorporated at system generation.

Compiler Residence

The PL/I Optimizing Compiler will occupy approximately 900,000 bytes of direct-access storage space in the core image library. The PL/I resident library subroutines which support this compiler will occupy approximately 100,000 bytes of direct-access storage space in the relocatable library. The PL/I transient library subroutines will occupy approximately 40,000 bytes of direct-access storage space in the core image library.

Working Storage

The PL/I Optimizing Compiler always requires direct-access storage space for working storage areas. The amount of space required depends upon the size of the source program and the amount of main storage available to the compiler. The system symbolic device SYS001 is used for this auxiliary storage. This device is also used when the PL/I 48-character set is used in the source program, and when the PL/I compile-time preprocessor is used in a job step in which its output is immediately compiled.

Input/Output Devices

At compile time and during subsequent link-editing, devices are required for the following types of input/output:

Source program input
Printed listings
Object module in relocatable format
Object module in relocatable card-image format

The symbolic name of the device associated with a particular class of compiler input/output, and the permitted device types for each, are shown in Table 2.

Table 2. Compiler Input/Output Devices

Function	Symbolic Name	Device Type	When Required
Input	SYSIPT	DASD Magnetic tape Card reader	Always
Print	SYSLST	DASD Magnetic tape Printer	Always
Output to Linkage Editor	SYSLNK	DASD Magnetic tape	When linkage editing follows compilation in the same job
Output to Linkage Editor (card deck)	SYSPCH	DASD Magnetic tape Card Punch	When linkage editing takes place in a subsequent job
Compiler Spill File	SYS001	DASD	Always
Source statement library	SYSSLB (if the source statement module is held in a private source statement library)	DASD	When preprocessor %INCLUDE is used
Relocatable library	SYSRLB (if the object module is held in a private relocatable library)	DASD	When the Linkage Editor is used to incorporate an object module from the relocatable library

Operating System Requirements and Facilities

The PL/I Optimizing Compiler can only be used in the batched-job processing mode, and cannot be used in a foreground partition of a multiprogramming version of the Disk Operating System.

When a PL/I program is compiled and executed, the operating system supervisor program initiates such operations as compilation, link-editing, and object program execution as individual job steps according to instructions received in job control language statements.

Before a compiled object program can be executed, it must be link-edited to form an executable program phase. The operating system linkage editor program must be employed to process the object module into an executable program phase to be stored in a program library from which it can subsequently be invoked.

The job control and linkage editor programs are described in the publication IBM System/360 Disk Operating System: System Control and System Service Programs, Order No. GC24-5036.

Object programs compiled by the DOS PL/I Optimizing Compiler make use of the Disk Operating System data management facilities. These facilities include:

Sequential Access Method (SAM)

Indexed Sequential Access Method (ISAM)

Direct Access Method (DAM)

The PL/I Optimizing Compiler has interface facilities with several components of the operating system. These can be used directly from within a PL/I program. They include:

1. SORT

   The PL/I program may contain statements to invoke the operating

system SORT program, and pass records to be sorted, or receive sorted records, or both pass and receive records.

2. Checkpoint/Restart

   The PL/I program may contain a statement to invoke the checkpoint facility to record the current status of a program and its data on an external storage medium. The checkpoint data can be used by the restart facility to restart the program at the point in execution reached when the checkpoint was taken.

Appendix A: Summary of Keywords

The following is a complete list of the PL/I and implementation-defined keywords implemented by the DOS PL/I Optimizing Compiler. Each of these keywords is described in depth in the publication IBM System/360 Disk Operating System: PL/I Language Reference Manual, Order No. SC33-0005.

Keyword	Abbreviation	Use of Keyword
ABS(x)		built-in function
ACOS(x)		built-in function
%ACTIVATE	%ACT	preprocessor statement
ADD(x,y,p[,q])		built-in function
ADDBUFF(n)		option of ENVIRONMENT attribute
ADDR(x)		built-in function
ALIGNED		attribute
ALL(x)		built-in function
ALLOCATE		statement
ALLOCATION(x)		built-in function
ANY(x)		built-in function
AREA		condition
AREA[(size)]		attribute
ARGn		option of the NOMAP, NOMAPIN, and NOMAPOUT options
ASIN(x)		built-in function
ATAN(x[,y])		built-in function
ATAND(x[,y])		built-in function
ATANH(x)		built-in function
AUTOMATIC	AUTO	attribute
BACKWARDS		file description attribute
BASED[(locator-expression)]		attribute
BEGIN		statement
BINARY	BIN	attribute
BINARY(x[,p[,q]])	BIN(x[,p[,q]])	built-in function
BIT(length)		attribute
BIT(expression[,size])		built-in function
BLKSIZE(max-blocksize)		option of ENVIRONMENT attribute
BOOL(x,y,w)		built-in function
BUFFERED	BUF	file description attribute
BUFFERS({1|2})		option of ENVIRONMENT attribute
BUILTIN		attribute
BY		clause of DO statement
BY NAME		option of the assignment statement
CALL		statement or option of INITIAL attribute
CEIL(x)		built-in function
CHAR(expression[,size])		built-in function
CHARACTER(length)	CHAR(length)	attribute
CHECK [(name-list)]		condition
CLOSE		statement
COBOL		option of ENVIRONMENT attribute or of the OPTIONS attribute/option
COLUMN(w)	COL(w)	format item
COMPLETION(event-name)		built-in function, pseudo-variable
COMPLEX	CPLX	data attribute
COMPLEX(a,b)	CPLX(a,b)	built-in function, pseudo-variable
CONDITION(name)		condition
CONJG(x)		built-in function
CONNECTED	CONN	attribute
CONSECUTIVE		option of ENVIRONMENT attribute
CONTROLLED	CTL	attribute
CONVERSION	CONV	condition
COPY		option of GET statement
COS(x)		built-in function
COSD(x)		built-in function
COSH(x)		built-in function

Keyword	Abbreviation	Use of Keyword
COUNT(file-expr)		built-in function
CTLASA		option of ENVIRONMENT attribute
CTL360		option of ENVIRONMENT attribute
DATA		STREAM I/O transmission mode
DATAFIELD		built-in function
DATE		built-in function
%DEACTIVATE	%DEACT	preprocessor statement
DECIMAL	DEC	attribute
DECIMAL(x[,p[,q]])	DEC(x[,p[,q]])	built-in function
DECLARE	DCL	statement
%DECLARE	%DCL	preprocessor statement
DEFAULT		statement
DEFINED	DEF	attribute
DELAY(n)		statement
DESCRIPTORS		option of the DEFAULT statement
DIM(x,n)		built-in function
DIRECT		file description attribute
DISPLAY		statement
DIVIDE(x,y,p[,q])		built-in function
DO		statement
%DO		preprocessor statement
EDIT		STREAM I/O transmission mode
ELSE		clause of IF statement
%ELSE		clause of %IF statement
EMPTY		built-in function
END		statement
%END		preprocessor statement
ENDFILE(file-expr)		condition
ENDPAGE(file-expr)		condition
ENTRY		attribute or statement
ENVIRONMENT	ENV	file description attribute
ERF(x)		built-in function
ERFC(x)		built-in function
ERROR		condition
EVENT		option of READ, WRITE, REWRITE, DISPLAY, and DELETE statements, attribute
EXP(x)		built-in function
EXTENTNUMBER(n)		option of ENVIRONMENT attribute
EXTERNAL	EXT	attribute
F		option of ENVIRONMENT attribute
FB		option of ENVIRONMENT attribute
FILE		attribute
FILE(file-expr)		option of input/output statements
FINISH		condition
FIXED		attribute
FIXED(x[,p[,q]])		built-in function
FIXEDOVERFLOW	FOFL	condition
FLOAT		attribute
FLOAT(x[,p])		built-in function
FLOOR(x)		built-in function
FORMAT(format-list)		statement
FORTRAN		option of the OPTIONS attribute/option
FREE		statement
FROM(variable)		option of WRITE or REWRITE statement
GENERIC		attribute
GET		statement
GO TO	GOTO	statement
%GO TO	%GOTO	preprocessor statement
HBOUND(x,h)		built-in function
HIGH(i)		built-in function
HIGHINDEX		option of ENVIRONMENT attribute
IF		statement
%IF		preprocessor statement
IGNORE(n)		option of READ statement
IMAG(x)		built-in function, pseudo-variable

Keyword	Abbreviation	Use of Keyword
IN (area-variable)		option of ALLOCATE and FREE statements
%INCLUDE		preprocessor statement
INDEX(string,config)		built-in function
INDEXAREA [(index-area-size)]		option of ENVIRONMENT attribute
INDEXED		option of ENVIRONMENT attribute
INDEXMULTIPLE		option of ENVIRONMENT attribute
INITIAL (expression)	INIT(expression)	attribute
INPUT		file description attribute, option of the OPEN statement
INTER		option of the OPTIONS attribute/option
INTERNAL	INT	attribute
INTO(variable)		option of READ statement
IRREDUCIBLE	IRRED	attribute
KEY(file-expr)		condition
KEY(x)		option of READ, DELETE, and REWRITE statements
KEYED		file description attribute
KEYFROM(x)		option of WRITE statement
KEYLENGTH		option of ENVIRONMENT attribute
KEYLOC		option of ENVIRONMENT attribute
KEYTO(variable)		option of READ statement
LABEL		attribute
LENGTH(string)		built-in function
LBOUND(x,n)		built-in function
LEAVE		option of ENVIRONMENT attribute
LIKE		attribute
LINE(w)		format item, option of PUT statement
LINENO(file-expr)		built-in function
LINESIZE		option of OPEN statement
LIST		STREAM I/O transmission mode
LOCATE		statement
LOG(x)		built-in function
LOG2(x)		built-in function
LOG10(x)		built-in function
LOW(i)		built-in function
MAIN		option of PROCEDURE statement
MAX(x_1,x_2...x_n)		built-in function
MIN(x_1,x_2...x_n)		built-in function
MOD(x_1,x_2)		built-in function
MULTIPLY(x_1,x_2,p[,q])		built-in function
NAME(file-expr)		condition
NOCHECK[(name-list)]		condition prefix (disables CHECK)
NOCONVERSION	NOCONV	condition prefix (disables CONVERSION)
NOFIXEDOVERFLOW	NOFOFL	condition prefix (disables FIXEDOVERFLOW)
NOLABEL		option of ENVIRONMENT attribute
NOMAP[(arg-list)]		option of the OPTIONS attribute/option
NOMAPIN[(arg-list)]		option of the OPTIONS attribute/option
NOMAPOUT[(arg-list)]		option of the OPTIONS attribute/option
NOOVERFLOW	NOOFL	condition prefix (disables OVERFLOW)
NORESCAN		option of %ACTIVATE statement
NOSIZE		condition prefix (disables SIZE)
NOSTRINGRANGE	NOSTRG	condition prefix (disables STRINGRANGE)
NOSUBSCRIPTRANGE	NOSUBRG	condition prefix (disables SUBSCRIPTRANGE)
NOTAPEMK		option of ENVIRONMENT attribute
NOUNDERFLOW	NOUFL	condition prefix (disables UNDERFLOW)
NOWRITE		option of ENVIRONMENT attribute
NOZERODIVIDE	NOZDIV	condition prefix (disables ZERODIVIDE)
NULL		built-in function

Keyword	Abbreviation	Use of Keyword
OFFSET[(area-name)]		attribute
OFLTRACKS(n)		option of ENVIRONMENT attribute
ON		statement
ONCHAR		built-in function, pseudo-variable
ONCODE		built-in function
ONCOUNT		built-in function
ONFILE		built-in function
ONKEY		built-in function
ONLOC		built-in function
ONSOURCE		built-in function, pseudo-variable
OPEN		statement
OPTIONS(list)		option of PROCEDURE statement
ORDER		option of PROCEDURE and BEGIN statements
OUTPUT		file description attribute, option of the OPEN statement
OVERFLOW	OFL	condition
PAGE		format item, option of PUT statement
PAGESIZE(w)		option of the OPEN statement
PICTURE	PIC	attribute
POINTER	PTR	attribute
POLY(a,x)		built-in function
POSITION (expression)	POS(expression)	attribute
PRECISION(x,p[,q])	PREC(x,p[,q])	built-in function
PRINT		file description attribute
PROCEDURE	PROC	statement
%PROCEDURE	%PROC	preprocessor statement
PROD(x)		built-in function
PUT		statement
RANGE		option of the DEFAULT statement
READ		statement
REAL		attribute
REAL(x)		built-in function, pseudo-variable
RECORD		file description attribute
RECORD (file-expression)		condition
RECSIZE (max-record-size)		option of ENVIRONMENT attribute
RECURSIVE		option of PROCEDURE statement
REDUCIBLE	RED	attribute
REFER		option of BASED attribute
REGIONAL(1|3)		option of ENVIRONMENT attribute
REORDER		option of PROCEDURE and BEGIN statements
REPEAT(string,i)		built-in function
REPLY(c)		option of DISPLAY statement
REREAD		option of ENVIRONMENT attribute
RESCAN		option of %ACTIVATE statement
RETURN		statement
RETURNS (attribute-list)		attribute, option of PROCEDURE statement
REVERT		statement
REWRITE		statement
ROUND(x,n)		built-in function
SCALARVARYING		option of ENVIRONMENT attribute
SEQUENTIAL	SEQL	file description attribute
SET(locator-variable)		option of ALLOCATE, LOCATE, and READ statements
SIGN(x)		built-in function
SIGNAL		statement
SIN(x)		built-in function
SIND(x)		built-in function
SINH(x)		built-in function
SIZE		condition
SKIP[(x)]		format item, option of GET and PUT statements
SNAP		option of ON statement
SQRT(x)		built-in function
STATIC		attribute
STATUS[(event-name)]		built-in function, pseudo-variable
STOP		statement
STREAM		file description attribute

Keyword	Abbreviation	Use of Keyword
STRING(x)		built-in function, pseudo-variable
STRINGRANGE	STRG	condition
STRINGSIZE	STRZ	condition
STRING(string-name)		option of GET and PUT statements
iSUB		dummy variable of DEFINED attribute
SUBSCRIPTRANGE	SUBRG	condition
SUBSTR(string,i[,j])		built-in function, pseudo-variable
SUM(x)		built-in function
SYSIN		name of standard system input file
SYSPRINT		name of standard system output file
SYSTEM		option of the ON statement
TAN(x)		built-in function
TAND(x)		built-in function
TANH(x)		built-in function
THEN		clause of IF statement
%THEN		clause of %IF statement
TIME		built-in function
TO		clause of DO statement
TRANSLATE(string, replacement, position)		built-in function
TRANSMIT		condition
TRUNC(x)		built-in function
U		option of ENVIRONMENT attribute
UNALIGNED	UNAL	attribute
UNBUFFERED	UNBUF	file description attribute
UNDEFINEDFILE(file-expr)	UNDF(file-expr)	condition
UNDERFLOW	UFL	condition
UNSPEC(x)		built-in function, pseudo-variable
UPDATE		file description attribute
V		option of ENVIRONMENT attribute
VALUE		option of the DEFAULT statement
VARIABLE		attribute of file or entry variable identifier
VARYING	VAR	string attribute
VB		option of ENVIRONMENT attribute
VERIFY		option of ENVIRONMENT attribute
VERIFY(string1, string2)		built-in function
WAIT		statement
WHEN (generic-descriptor-list)		GENERIC declaration
WHILE		clause of DO statement
WRITE		statement
ZERODIVIDE	ZDIV	condition

Appendix B: Compatibility with the DOS PL/I D Compiler

The following notes describe those differences between the language implemented by the PL/I D compiler (version 4) and the DOS PL/I Optimizing Compiler that may cause minor incompatibilities.

Programs which use any of these features should be reviewed before recompilation to ensure that they will return the same results.

Expressions in DO Statements

Expressions in DO statements are evaluated by the D Compiler in the order "expression2" followed by "expression3", irrespective of the order of appearance in the DO statement. For example:

```
DO I = J TO K BY L;

DO I = J BY L TO K;
```

In both statements "expression2" is represented by K and "expression3" by L. The D Compiler always evaluates expression2 first, whereas the order in which the DOS PL/I Optimizing Compiler evaluates each expression is undefined.

SYSIN and SYSPRINT

Although the names SYSIN and SYSPRINT have no special meaning for the D Compiler, they do for the DOS Optimizing Compiler. PL/I programs can contain stream-oriented (GET or PUT) data transmission statements which do not specify a file name. The D Compiler treats such statements as referring to the symbolic devices SYSIPT and SYSLST; the DOS Optimizing Compiler makes the assumption that such input statements refer to SYSIN, and output statements to SYSPRINT.

E and F Format Items

1. SIZE error: For the D Compiler, if a SIZE error occurs during output of data under control of an E or F format item, the value in error is transmitted as a field of asterisks. For the DOS Optimizing Compiler, if SIZE is enabled, the value is transmitted as a field of asterisks; if SIZE is not enabled, the value is truncated to the size of the field.

2. Zero before decimal point: When F format fractional values or E format zero mantissa values are transmitted, the DOS PL/I Optimizing Compiler inserts a leading zero before the decimal point. The D Compiler does not put the zero before the point. For example:

```
D Compiler   -.500
DOS PL/I Optimizing Compiler
        -0.500
```

REGIONAL Data Sets

REGIONAL data sets for programs written for the PL/I D Compiler, are, when created, preformatted by a utility program. This program is executed as a separate job step prior to execution of the PL/I program in which the output file is opened to create the data set. Subsequent use can be made of this data set through an OUTPUT file without formatting it again.

REGIONAL data sets created for programs compiled by the DOS PL/I Optimizing Compiler are preformatted by a PL/I library routine when the output file is opened. Thus an output file cannot be opened to process a REGIONAL data set without destroying all the records contained in it. If records are to be added to the REGIONAL data set, an UPDATE file must be used.

Preformatting, including the preformatting of secondary extents, is performed as follows:

1. A REGIONAL(1) data set is preformatted with dummy records in which the first byte is set to X'FF' and the remaining bytes are left undefined. For SEQUENTIAL OUTPUT files, the dummy records are written into those regions which do not receive a data record during processing. For DIRECT OUTPUT files, the entire data set is formatted before any records are written onto it.

2. A REGIONAL(3) data set opened for SEQUENTIAL OUTPUT will have each track

formatted before new records are written onto it during processing; for a data set opened for DIRECT OUTPUT, the entire data set will be formatted when the file is opened.

Note that since dummy records can be retrieved by a READ statement, the programmer must ensure that such records are recognized by the program.

Halfword Binary Numbers

Fixed-point binary numbers with a precision of (15) or less are held in main storage as halfword binary numbers by programs compiled by the DOS PL/I Optimizing Compiler. Fixed binary numbers with a precision greater than (15) are held as fullword binary numbers. All fixed-point binary numbers in programs compiled by the DOS PL/I D Compiler are held as fullword binary numbers.

D Compiler programs to be recompiled should be checked for occurrences of FIXED BINARY variables which have precisions of (15) or less (thus including those with default precision), since they might occur in record-oriented transmission and cause differences in record lengths and in the alignment of records in locate-mode buffers. A similar problem could occur for programs that process data sets created by D Compiler object programs. Bit-string values returned by the UNSPEC built-in function when used with halfword binary numbers as arguments are 16 bits in length. The DEFAULT statement may be used to ensure that all undeclared fixed binary variables have the maximum precision (31,0).

Labels on DECLARE Statements

The PL/I D Compiler ignores any labels prefixed to DECLARE statements. The PL/I Optimizing Compiler recognizes such labels and treats branches to such labels as branches to null statements. An incompatibility can occur if in a recompiled D Compiler program such a label has the same identifier as a variable or is used as a label prefix to another statement.

Unaligned Bit Strings

The DOS PL/I Optimizing Compiler implements the UNALIGNED attribute for bit strings. UNALIGNED is the standard language default for bit strings. The DOS PL/I D Compiler allows only character class data to be unaligned. Programs for the Optimizing Compiler that are to process records containing bit strings from a data set created by a PL/I D Compiler object program should specify ALIGNED bit strings in the appropriate record variables. The DEFAULT statement may be used to achieve this effect where bit strings are not explicitly declared with the ALIGNED attribute.

ONSYSLOG Option

The DOS PL/I Optimizing Compiler does not support the use of the ONSYSLOG option, whereby all output resulting from actions derived from on-conditions is printed on the system log.

DYNDUMP

The DOS PL/I Optimizing Compiler does not permit use of the DYNDUMP, IJKTRON, IJKTROF, and IJKEXHC routines.

DISPLAY Statement and REPLY Option

The PL/I Optimizing Compiler permits strings up to 72 bytes in length for both the DISPLAY statement and the REPLY option. The PL/I D Compiler permits strings up to 80 bytes for the DISPLAY statement, and up to 256 bytes for the REPLY option.

INDEX Built-In Function

The INDEX built-in function can be used with a binary arithmetic argument that requires conversion to character string form before the function can be executed. This occurs wherever the other argument of this function is either a decimal arithmetic value or a character string. For example:

1. `INDEX(A,I)`
2. `INDEX(I,'B')`

In both cases I is a binary arithmetic variable. A is a decimal arithmetic variable. In the first case, both A and I are converted to character form, in the second case only I is converted.

An incompatibility exists between the methods and the results of conversion from binary arithmetic to character form for this function. The D Compiler converts a binary arithmetic argument to an intermediate bit string form which is then converted to character form. The Optimizing Compiler converts the argument to an intermediate decimal arithmetic form which is then converted to character form.

PRECISION Built-In Function

The PRECISION built-in function is implemented differently by the PL/I D and Optimizing Compilers. For the D Compiler, if the first argument is FIXED, and the third argument is omitted, the third argument is assumed to be zero. For the Optimizing Compiler, if the first argument is FIXED, and the third argument is omitted, the third argument is assumed to be zero, and an informatory diagnostic message is given.

SUM and PROD Built-In Functions

For the DOS PL/I Optimizing Compiler, the SUM and PROD built-in functions accept arguments that can be arrays of either fixed-point or floating-point elements. The value returned is in the same scale as the argument given, except for the PROD built-in function used with fractional fixed-point arguments, where the value returned is in floating-point scale. Note that string arguments are converted to fixed-point arithmetic form, and that the result is returned in this form.

For the DOS PL/I D Compiler, the arguments of these functions are, if necessary, converted to floating-point scale. The returned value always has floating-point scale.

Attributes of File Parameters

For the D Compiler, a file parameter can be declared with other attributes in addition to the FILE attribute. For the Optimizing Compiler, a file parameter can only be declared with the FILE attribute; all other attributes are inherited from the argument. If additional attributes are given, the compiler will issue an informatory message, and ignore them.

Defining of Arrays of Pictures

Simple defining of arrays of pictures will be diagnosed as an error if the defined elements do not exactly match the base elements. The PL/I D Compiler requires only that the base elements should be pictures or character strings.

Sterling Pictures

Sterling data is not supported by the DOS PL/I Optimizing compiler. A picture including any of the following characters is invalid:

G, M, H, P, 6, 7, 8

Source Program Errors

The D Compiler does not detect all the errors in a source program that can be detected by the Optimizing Compiler. Errors that are not detected include the transfer of control into an iterative DO group, comparison of structures, and incorrect overlay defining.

Programs which contain these errors and compile successfully with the D Compiler will not compile successfully with the Optimizing Compiler.

RETURNS Keyword in PROCEDURE and ENTRY Statements

PROCEDURE and ENTRY statements for function procedures that specify the attributes of the value returned by the procedure must, for the Optimizing Compiler, have such attributes contained in a parenthesized list preceded by the keyword RETURNS. For example, the following statement is valid for the D Compiler, but not for the Optimizing Compiler:

```
X:  PROCEDURE (Y,Z) FLOAT BINARY;
```

For the Optimizing Compiler, this statement should be written as follows:

```
X:  PROCEDURE (Y,Z) RETURNS (FLOAT
    BINARY);
```

Entry Names as Arguments

The D Compiler assumes an entry name argument in parentheses and without arguments of its own to be a function reference. For example, in the expression X((Y)) the function Y is invoked and the value it returns is used as the argument to procedure X. The Optimizing Compiler assumes that the entry name itself is to be passed as an argument. It creates an entry variable with the value of the entry constant argument, and passes this as a dummy argument to the invoked procedure. Function references such as this in programs written for the D Compiler should be modified to contain a null argument list in order to invoke the function. For example, the expression given above should be written as X(Y()).

Index

Where more than one page reference is given, the major reference is first.

Main page references are those listed first.

READER'S COMMENT FORM

IBM System/360 Disk Operating System
PL/1 Optimizing Compiler
General Information Manual

Order No. GC33-0004-0

- How did you use this publication?

  As a reference source ..... ☐
  As a classroom text ..... ☐
  As a self-study text ..... ☐

- Based on your own experience, rate this publication . . .

  As a reference source:

Very Good	Good	Fair	Poor	Very Poor

  As a text:

Very Good	Good	Fair	Poor	Very Poor

- What is your occupation? .....

- We would appreciate your other comments; please give specific page and line references where appropriate. If you wish a reply, be sure to include your name and address.

- Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A.

YOUR COMMENTS PLEASE . . . .

This SRL manual is part of a library that serves as a reference source for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note : Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

Cut Along Line

fold fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N.Y.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY...

IBM Corporation
112 East Post Road
White Plains, N.Y. 10601

Attention: Department 813 (HP)

fold fold

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

GC33-0004-0

IBM System/360 Printed in U.S.A. GC33-0004-0

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]