TO: J. L. BASH
G. D. BENNETT
G. D. CHANG
C. T. CLINGEN
F. J. CORBATÓ
R. C. DALEY
J. W. GINTELL
J. D. MILLS
J. H. SALTZER
B. L. WOLMAN

FROM: R. A. FREIBURGHOUSE

SUBJECT: PL/I MANUAL

DATE: AUGUST 27, 1969

Some comments inside - mostly typos.
Very good guide for experienced programmer.
Can we use the data layouts section for the MPM?

Attached is a draft of the User's Guide to the Multics PL/I Implementation. Due to the lack of sufficient time to permit retyping, this copy is the original draft with hand-written editing notes. Your comments and suggestions are welcome.

In order to meet our publication deadline, all comments must be received by September 10, 1969.

R A Freiburghouse

R. A. FREIBURGHOUSE

/ll
(enclosure)

Bob-
It would be very useful to have
1. An option which prints the names of variables declared by implicit context on-line
2. An option to receive a warning diagnostic on every type conversion compiled for the program.
Jerry

Should delete references to MSPM - maybe change them to MPM

A USER'S GUIDE TO THE MULTICS PL/I IMPLEMENTATION

R. A. Freiburghouse
J. D. Mills
B. L. Wolman

INTRODUCTION

This document supplies the prospective Multics PL/I user with detailed information about the Multics PL/I implementation. It is a supplement to the Multics PL/I Language Specification and should be read by programmers who have a reasonable understanding of the PL/I Language. It was written primarily for use by Multics system programmers but selected portions may be used by any PL/I programmer. Sections (1,2,3,6,7,8) are of general interest while sections (4,5) may be of interest only to EPL programmers. Readers who do not ha e a copy of the Multics PL/I Language Specification may temporarily use the PL/I Language Specification published by the IBM Corporation, publication number X33-6003-0. The IBM document describes a language which is a superset of the Multics PL/I language. It may be used to initiate the reader to PL/I, but before writing programs he should obtain a copy of the Multics PL/I Language Specification.

CONTENTS

		Page
1.	The PL/I Command	1
	1.1 Purpose	1
	1.2 Usage	1
	1.3 Command Options	2
	1.4 Error Diagnostics	2
	1.5 Listing	4
2.	The Machine Representation of PL/I Data	4
	2.1 Scalar Data	4
	2.1.1 Real Fixed-Point Short	5
	2.1.2 Real Fixed-Point Long	5
	2.1.3 Real Floating-Point Short	5
	2.1.4 Real Floating-Point Long	5
	2.1.5 Aligned Non-Varying Bit String	6
	2.1.6 Unaligned Non-Varying Bit String	6
	2.1.7 Aligned Non-Varying Character String	6
	2.1.8 Unaligned Non-Varying Character String	6
	2.1.9 Varying Bit String	6
	2.1.10 Varying Character String	7
	2.1.11 Pointer	7
	2.1.12 Offset	7
	2.1.13 Label	7
	2.1.14 Entry	8
	2.2 Aggregate Data	8
	2.2.1 Arrays	8
	2.2.2 Structures	9
	2.3 Data Type Codes Used in PL/I Descriptors	9
3.	The PL/I Call and Argument Passing Conventions	9
	3.1 Arguments	9
	3.2 Argument Descriptors	10
	3.2.1 Descriptor Formats	10
	3.3 The Format of a PL/I Call	11
4.	Argument Compatability Between PL/I and EPL	12
	4.1 Restrictions on the Types of Arguments	12
	4.2 Notes on the Use of the Compatibility Feature	13
	4.3 A Detailed Description of the Modifications Made to EPL Object Code to Implement the Compatibility Feature	13

CONTENTS (Continued)

Page

5. The Conversion of EPL Programs to PL/I Programs 15
5.1 Fixed-Point Division 15
5.2 The Length Built-in Function 15
5.3 Bit String to Arithmetic Conversion 15
5.4 Use of the Returns Attribute 16
5.5 The Significance of the Entry Attribute 16
5.6 Label Arrays as Transfer Vectors 19
5.7 The Use of cv-string and char(*) Declarations 19
5.8 Simple rules for Converting EPL Array or Structure Declarations to Equivalent PL/I Declarations 20
5.9 The Effect of the Alignment Attributes on Storage Allocation and on the Packing of Arrays and Structures 21
5.9.1 Definitions 21
5.9.2 The Storage Allocation Rules of the PL/I Compiler 21
5.10 The Effect of the Alignment Attribute on the Accessing of Based Variables and Parameters 22

6. Tips on Writing Efficient PL/I Programs 23
6.1 General Comments 23
6.2 The Use of Entry Declarations 23
6.3 Parameters 23
6.4 Begin Blocks 24
6.5 Internal Procedures 24
6.6 Accessing Code 24
6.6.1 The Effect of Block Structure on Accessing 24
6.6.2 Static Storage References 24
6.6.3 References to Based Storage 24
6.6.4 The Design of Aggregates and their Effect on Accessing Code 25
6.7 The Efficient Use of String Data 26
6.8 Label Variables 27
6.9 Use of the Initial Attribute 27
6.10 Multiple Assignment Statements 27

7. The Implementation of PL/I Storage Classes 28
7.1 Automatic Storage 28
7.2 Internal Static 28
7.3 External Static 28
7.4 Based Storage 29

CONTENTS (Continued)

		Page
8.	The Fundamentals of Multics for PL/I Programmers	30
	8.1 Segments and Directories	30
	8.2 A Process	30
	8.3 Dynamic Linking	31
	8.4 The Search Mechanism	31
	8.5 The Hidden Dangers of Dynamic Linking	31
	8.6 A Process and the Execution of a PL/I Program	32

Appendices

A PL/I Compiler Generated Error Messages

B Error Messages Generated During the Execution of a PL/I Program

1. THE PL/I COMMAND

1.1. Purpose

The pl1 command invokes the PL/I compiler to translate a segment containing ~~PL/I source code~~ the text of a PL/I external procedure into a MULTICS object segment. A listing segment may also be produced. These results are placed in the user's current working directory.

1.2. Usage

The command

```
pl1 pathname -optl- . . . -optn-
```

invokes the PL/I compiler to translate a PL/I source segment identified by pathname. (The typed command consists of the letters "pl" and the numeral "1"). A directory path name and an entry name, segname, are derived from pathname by calling expand_path_ (Ref.BS.13.50) and the compiler takes its input from segname .pl1. optl, . . ., optn are optional arguments to the compiler whose interpretation is defined below under "Options".

A normal compilation will produce an object segment, <u>segname</u> and leave it in the user's working directory. If <u>segname</u> existed previously in the directory its ACL is saved and given to the new copy of <u>segname</u>. Otherwise the user is given RE access to the segment with ring brackets V 48 48 where V = validation level of the process active when the object segment is created.

The user's options will control the absence or presence of the listing segment for segname.pl1 and the contents of that listing. If created, the listing segment is named segname.list. The ACL is as described for the object segment except that it is given RWA access when newly created. Previous copies of segname and (if the list option is on) segname.list are replaced by the new segments created by the compilation.

Note that because of the MULTICS standard which restricts the length segment names, a PL/I source segment name may not be longer than 25 characters.

The pl1 command will look for the presence of "%"; as the first two characters in segname.pl1. The presence of such characters implies that segname.pl1 contains "% include" compile-time statements. pl1, therefore, creates a new source segment with all "% include" statements expanded. The compiler then takes its input from the expanded results, and the segment segname.ex.pl1 is left in the working directory.

1.3. Command Options

In the absence of the full MULTICS option machinery, character string arguments to the command provide the user with a certain amount of control over the output from pl1. The options are summarized here. Further information is contained under "Error Diagnostics" and "Listing".

Option	Result
"source"	pl1 produces a line-numbered printable ascii listing of the source program. The default is no listing.
"symbols"	Listing all the variables declared in the program with their attributes. The default is no symbols.
"assembly_list"	Produce an assembly-like listing of the ~~test, link, and symbol segments that were compiled~~ compiled object segment. The default is no assembly listing.
"list"	Produce a listing of the source, symbols, and assembly listing. The default is no list. "list" is equivalent to "source", "symbols", and "assembly_list".
"brief"	Error messages written into the stream "user_output" will contain only an error number, statement identification, and when appropriate the identifier or constant in error. In the normal, non-brief mode an explanatory message of one or more sentences will also be written.
"severityi"	Error messages whose severity is less than i (where i is 1, 2, 3, or 4, e.g. severity 3) will not be written into "user_output" although all errors will be written into the listing. The default is 1.
"check"	Used for syntactic and semantic checking of a PL/1 (or epl) program. Only the first three phases of the compiler are executed. Code generation is skipped as is the manipulation of the working segments used by the code generator.

1.4. Error Diagnostics

The PL/I compiler can diagnose and issue messages for about 350 different errors. These messages are graded in severity as follows:

Severity Level	Meaning
1	Warning only - compilation continues without ill effect.

Severity Level	Meaning
2	Correctable error - the compiler remedies the situation and continues probably without ill effect. For example, too few end statements can be and is corrected by simulating the appending of a sufficient number of strings ";end;" to the source to complete the program. This does not guarantee the right results however.
3	An uncorrectable but recoverable error. That is, the program is definitely in error and cannot be corrected but the compiler can and does continue executing up to the point just before code is generated. Thus, any further errors will be diagnosed.
4	An unrecoverable error. The compiler cannot continue beyond this error. The message is printed and then control is returned to the pl1 command unwinding the compiler. The command writes an abort message into "user_output" and returns to its caller.

Error messages are written into the stream "user_output" as they occur. Thus, a user at his console can quit the compilation process immediately when he sees something is amiss. As indicated above, the user can set the severity level so that he is not bothered by minor error messages. He can also set the brief option so that the message is shorter. An example of an error message in its long form is:

```
ERROR 281.1 IN STATEMENT 1 ENDING ON LINE 17

   The entry name`zilch' has been declared internal but has not
   been defined within the block of declaration.
```

If the brief option had been set the user would see instead:

```
ERROR 281.1 IN STATEMENT 1 ENDING ON LINE 17
zilch
```

In the second case the user could look up error number 281 in appendix A of this manual and get the full message. The digit after the decimal point in the error number is the severity of the message. Thus, if the user had set his severity level to 2 he would have seen no message at all.

If the listing option is on, the error messages are also written into the listing segment. They appear, sorted by line number, after the listing of the source program. Because of an implementation restriction no more than 100 messages will be printed in the listing.

1.5. Listing

The listing created by PL/I is a line numbered image of the possibly expanded source segment. This is followed by a table of all of the variables declared within the program. The variables are categorized by declaration type which are:

1. Declare Statement

2. Explicit Context (labels, entries, and parameters)

3. Implicit and Context

Within these categories the symbols are sorted alphabetically and then listed with their location; storage class; data type; size, precision, or level; and attributes such as "initial", "array", "abnormal", "internal", "external", "aligned", "unaligned", and "irreducible". The symbol listing is followed by the error messages.

Finally, the listing contains the assembly-like listing of the object segment produced. The executable instructions are grouped under an identifying header indicating the source statement which produced the instructions. Opcode, base-register, and modifier mnemonics are printed along-side the octal instruction. The addresses are numerical but if an identifier or constant corresponds to the address it is printed in the "remarks" field of the line. Constants and links are printed with symbolic interpretation also.

2. THE MACHINE REPRESENTATION OF PL/I DATA

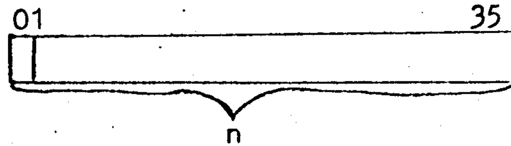
This section defines the representation of PL/I data in the GE 645. It includes a list of the data type codes used in PL/I argument descriptors but does not contain a description of the PL/I call or argument passing conventions. Refer to section 3.0 for a discussion of the PL/I call.

2.1. Scalar Data

The following is a description of the representation of Scalar Data. The term "double word" is defined to be two adjacent 36-bit words, the first of which is located at an even storage address. In the description of arithmetic data p is understood to be the binary precision of the data. Data declared with a decimal precision is represented as a binary value whose precision is determined from the declared decimal precision according to the rules of the language.

2.1.1. (type 1) Real fixed-point short

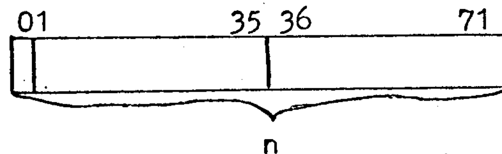
A real fixed-point datum of precision $0<p\leq35$ is stored as a 645 single word integer.



where n is a signed integer in 2's complement form.

2.1.2. (Type 2) Real fixed-point Long

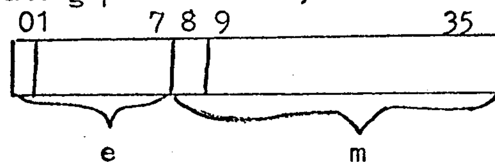
A real fixed-point datum of precision $35<p\leq71$ is stored as a 645 double word integer,



where n is a signed integer in 2's complement form.

2.1.3. (Type 3) Real floating-point short

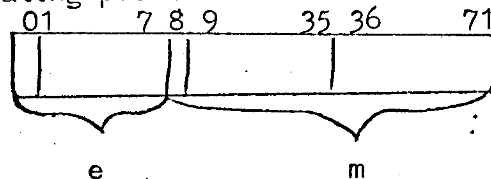
A real floating-point datum of precision $0<p\leq27$ is stored as a 645 single word floating-point number,



where e is the exponent and m is the mantissa. Both e and m are signed integers in 2's complement form.

2.1.4. (Type 4) Real floating-point Long

A real floating-point datum of precision $27<p\leq63$ is stored as a 645 double word floating-point number.



where e is the exponent and m is the mantissa. Both e and m are signed integers in 2's complement form.

2.1.5. (Type 519) Non-Varying Bit String

An aligned non-varying bit string is a set of contiguous bits which begins with the leftmost bit of a word and extends through as many words as are necessary to represent the string. The leftmost bit of the string is bit 1 while the rightmost is bit n.

2.1.6. (Type 519) Non-Varying Bit String

An unaligned non-varying bit string is a set of contiguous bits which may begin on any bit of a word and which extends through as many words as are necessary to represent the string. The leftmost bit of the string is bit 1 while the rightmost is bit n.

2.1.7. (Type 520) Non-Varying Character String

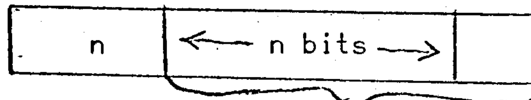
An aligned non-varying character string is a set of contiguous 9-bit bytes each of which contains a 7-bit ASC11 character right justified within the bytes. The string begins on the first bit of a word and extends through as many words as are necessary to contain the string. The leftmost character is character 1 while the rightmost is character n.

2.1.8. (Type 520) Non-Varying Character String

An unaligned non-varying character string is a set of contiguous 9-bit bytes each of which contains a 7-bit ASC11 character right justified within the byte. The string may begin on bit 0, 9, 18, or 27 within a word and extends through as many words as are necessary to contain the string.

2.1.9. (Type521) Varying Bit String

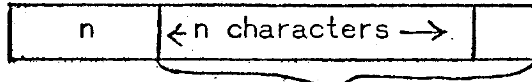
A varying bit string is a compound datum consisting of a real fixed point integer followed by an aligned non-varying bit string whose length is the declared maximum length of the string. The fixed-point value indicates the current length of the string.



where n is the current size of the string and l is the maximum size of the string. Both n and l are measured in bits. The address of a varying bit string is the address of the current length + 1.

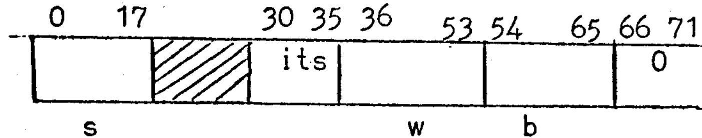
2.1.10. (Type 522) Varying Character String

A varying character string is a compound datum consisting of a real fixed-point integer followed by an aligned non-varying character string whose length is the declared maximum length of the string. The fixed-point value indicates the current length of the string. ← Include comment "the address of a varying character string" use K!



2.1.11 (Type 13) Pointer

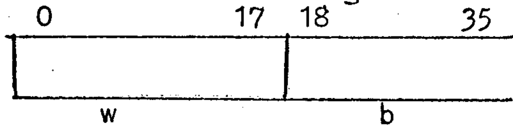
A pointer datum is stored as a 645 double word ITS pair.



where s is the segment number, w is the word offset from the beginning of the segment, and b is the bit offset from the beginning of the word addressed by the segment number and word offset. Pointers containing the address of unaligned data have a bit offset whose value is $0 \leq b < 36$.

2.1.12. (Type 14) Offset

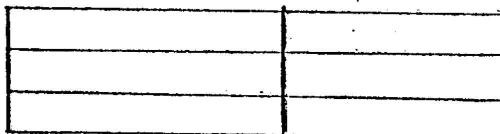
An offset datum is stored in a single word.



when w is a word offset from the beginning of a PL/I area datum. b is the bit offset within the word addressed by the word offset. Offsets containing the address of aligned data always have zero bit offsets. Offsets containing the address of unaligned data have a bit offset whose value is $0 \leq b < 36$.

2.1.13 (Type 15) Label

A Label datum is stored in three Consecutive double words.

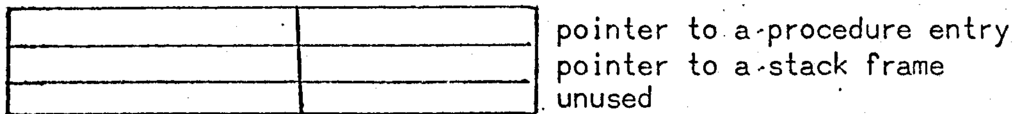


pointer to a location in text segment
pointer to a stack frame
unused

The first double word is a pointer to a location in the text segment of some procedure. The second double word is a pointer to the stack frame of the invocation of the procedure active at the time the label value was created. The last double word is unused and reserved for error check data.

2.1.14. (Type 16) Entry

An entry datum is stored in three consecutive double words.



The first double word is a pointer to an entry point in the text segment of some procedure. If the entry point is an entry to an internal procedure, the second double word is a pointer to the stack frame of the invocation of the procedure which immediately contains the internal procedure. If the entry point is an entry to an external procedure the second double word contains a null pointer. The last double word is unused and reserved for error check data.

2.2. Aggregate Data

The term "Aggregate" refers to PL/I arrays and structures. The following statements apply to PL/I Aggregates:

The amount of storage occupied by an aggregate is the amount of contiguous storage necessary to contain all of its elements.

The storage boundary on which an aggregate is allocated is the maximum boundary required by any of its elements.

An Aggregate is said to be "packed" if it contains only unaligned non-varying bit strings (or packed aggregates of such strings) or if it contains only unaligned non-varying character strings (or packed aggregates of such strings).

Level one packed aggregates are allocated on a word boundary.

The elements of an aggregate are allocated on the boundary that is natural to that element. The natural boundary for most data is a word or double word. Packed aggregates and unaligned non-varying string scalars are allocated on bit or character boundarys. ies.

2.2.1. Arrays

An Array is an n-dimensional, ordered collection of scalars or structures, all of which have identical attributes. The elements of an array are stored in row major order. (When accessed sequentially the rightmost subscript varies most rapidly).

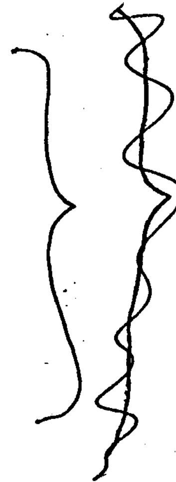
2.2.2. Structures

A structure is a hierarchical collection of scalars, arrays, and structures, all of which need not have the same attributes. The elements of a structure are stored contiguously in the order of their declaration. If an element of a structure is an aggregate, all the members of that aggregate taken together constitute the storage for that element.

2.3. ~~DATA TYPE CODES USED IN PL/I DESCRIPTORS~~ Data Type Codes Used in PL/I Descriptors

1 single precision real fixed-point
2 double precision real fixed-point
3 single precision real floating-point
4 double precision real floating-point
13 pointer data
14 offset data
15 label data
16 entry data
17-20 arrays of types 1-4
29-31 arrays of types 13-15

514 structure
518 area
519 bit string
520 character string
521 varying bit string
522 varying character string
523 array of structures
524 array of areas
525 array of bit strings
526 array of character strings
527 array of varying bit strings
528 array of varying character strings



data types which are not Multics standard

3. THE PL/I CALL AND ARGUMENT PASSING CONVENTIONS

The calling sequence produced by th PL/I compiler is the MULTICS standard call as described in BD.7.02 with one minor modification. BD.7.02 states that the right half of the first word of the argument list is a 0 for calls to external procedures and 2 for calls to internal procedures. Pl/I uses the codes 4 and 8 to indicate these same conditions. For the purpose of argument compatibility, it is essential for Pl/I [PL/I] and EPL object programs to know whether or not they are being called from a PL/I procedure or from an EPL (or EPL-like) procedure. The use of the new codes (4 and 8) serve this purpose.

3.1. Arguments

The argument pointers of the PL/I call point directly to the value of the argument. All arguments are directly addressed including those which do not begin on a word boundary. If the data does not begin on a word boundary the pointer refers to the first word which contains the data. The pointer contains the bit offset necessary to address the data. The format of a PL/I pointer is discussed in Section 2.2.11.

3.2. Argument Descriptors

The PL/I implementation of strings and the design of the compiler has eliminated the need for argument descriptors except when a parameter is declared with an * extent or when descriptors are used for some purpose outside the PL/I language, such as callback or PL/I to EPL compatibility. The only conditions which will cause the PL/I compiler to produce descriptors are:

1. The parameters of the entry are not described through the use of an entry(<parameter descriptions>) attribute in the calling program.

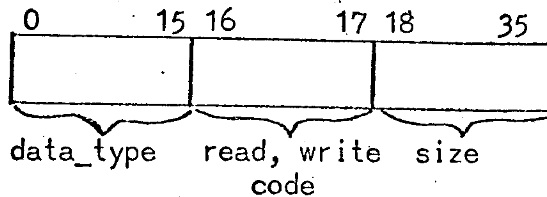
2. An entry(<parameter descriptions>) attribute in the calling program has one or more parameter descriptions containing an * extent.

3.2.1. Descriptor Formats

The design of descriptors is an extension of the design given in BS.7.02 and is compatible with it.

BD.7.02

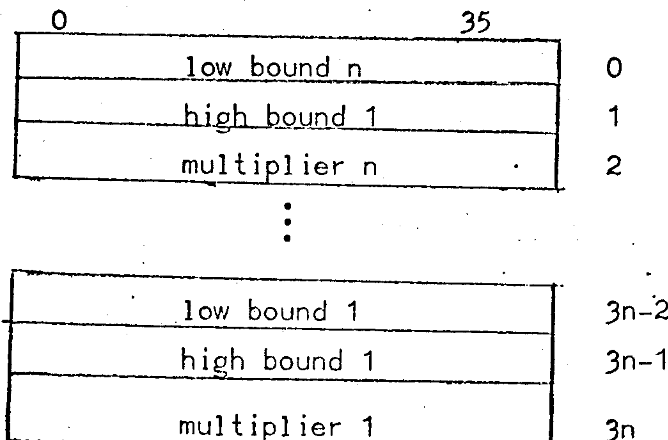
Basic Descriptor



All arguments (scalars or aggregates) have a basic descriptor of this form. The size field is defined only for strings and areas. It represents the declared size in bits, characters, or words. The read/write code is used to indicate whether the argument is a read/only or a return argument. If the argument is a temporary (PL/I dummy) the code will be 1 otherwise it will be 2. The data type codes of PL/I are listed in Section 2.3.

Array Descriptor

If the data type of the basic descriptor is that of an array the basic descriptor will be followed by an array descriptor of the following form:

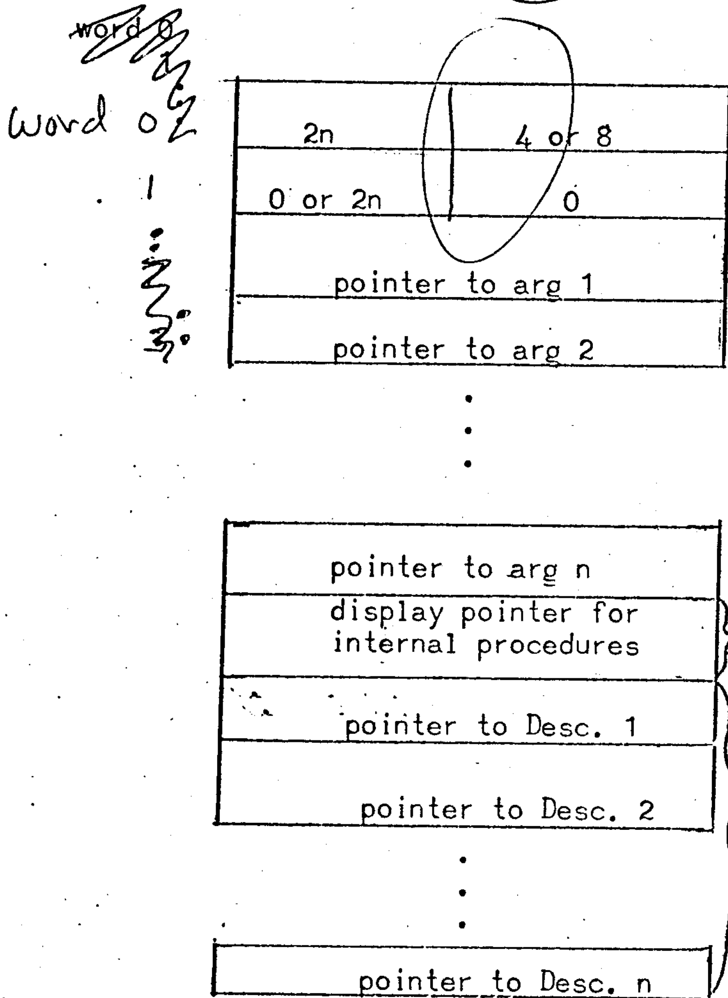


The bounds are derived directly from the declared bounds of the argument. The multipliers are computed from the bounds and the element size. Multipliers of packed arrays are expressed in bits, multipliers of non-packed arrays are expressed in words.

Structure Descriptor

If the data type of the basic descriptor is that of a structure or array of structures, the basic descriptor or array descriptor is followed by descriptors of each element of the structure. The relative position of an element descriptor within the structure descriptor is the same as the relative position of the data it describes within the data structure.

3.3. The Format of a PL/I Call



The display pointer is optional and only present if the right half of word 0 is 8.

Pointers to argument descriptors. These pointers are optional and only present when the left half of word 1 is 2n.

All argument pointers are PL/I data pointers as described in section 2.1.11. n is the total number of arguments.

4. ARGUMENT COMPATABILITY BETWEEN PL/I AND EPL

Because of differences in the way EPL and PL/I object programs pass string and aggregate arguments, procedures written in the two languages cannot communicate as freely with each other as they can with procedures written in the same language. This chapter discusses the restrictions which apply to calls between procedures written in different languages and describes the compatibility feature which is designed to minimize these restrictions.

An entry to a PL/I procedure is sensitive to the kind of call it receives. When called by an EPL procedure (or any other non-PL/I procedure) it maps EPL specifiers and dope into PL/I argument descriptors. The EPL compiler was recently modified (July 1969) so that EPL object programs perform a similar function. When called by a PL/I procedure containing argument descriptors they map the argument descriptors into appropriate specifiers and dope vectors.

4.1. Restrictions on the Types of Arguments

PL/I - EPL data can be classified into three categories: category I is data which can be passed between procedures written in different languages without descriptor/dope vector mapping. Category II data is data which can be passed between procedures written in different languages only with the aid of descriptor/dope vector mapping. Category III data is data which cannot be passed between procedures written in different languages.

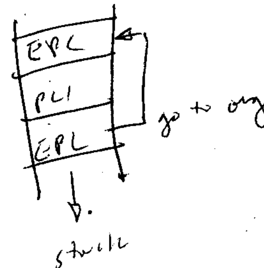
Category I

1. Arithmetic scalars
2. Scalar pointers
3. Scalar label variables *
4. Label constants *
5. External entry names

Category II

1. Non-varying scalar strings
2. One dimensional arrays of any of the previously listed data types.
3. Scalar varying strings. These strings always appear to the called procedure as non-varying strings whose length is the current length at the time the call occurred. The must be declared as non-varying in the called procedure-if declared the calling program they must be varying.

*A procedure written in either language can not be entered via a non-local go to orginating from a procedure written in the other language.



Category III

1. Structures
2. Multi-dimensional arrays
3. Areas
4. Offset variables
5. Internal entry names

4.2. Notes on The Use of The Compability Feature

EPL calls to PL/I procedures automatically invoke the compatability feature of PL/I. If any EPL argument contains a specifier that specifier is mapped into a correct PL/I argument pointer. If the PL/I procedure has any parameters whose extents were declared by asterisks, the EPL dope vector is mapped into a PL/I argument descriptor. Space for these descriptors is created in the PL/I program's stack.

PL/I calls to EPL procedures, compiled after July 1969, automatically invoke the compatability feature of EPL. If the PL/I call contains no argument descriptors the EPL program is executed just as if it had been called by an EPL program. If the PL/I call does contain argument descriptors, those descriptors are examined and mapped into appropriate EPL specifiers and dope vectors. Space for these dope vectors and specifiers is created in the EPL program's stack.

Note that it is extremely important that PL/I calls to EPL procedures contain argument descriptors if any of the arguments are category II data types. If all of the arguments are of Category I then the argument descriptors are not necessary. In the latter case, the call will execute more efficiently if argument descriptors are not present.

A PL/I call will contain argument descriptors if and only if:

> The entry name was declared by default or by the short form of the entry attribute, i.e., entry
>
> or
>
> The entry name was declared by the long form of the entry attribute and one or more parameter descriptions contained an asterisk extent, i.e., entry(bit(*))

4.3. A Detailed Description of the Modifications made to EPL Object Code to Implement the Compatability Feature

The EPL compiler has been modified to allow EPL compiled programs to be called from programs compiled by PL/I.

The save sequence generated by EPL now consists of the instructions:

```
.svext:     ldaq        op|0
            cana        8+4, dl
            tze         .sv
            tsbbp       <pl1_to_epl>|[pl1_to_epl]
.sv:        . . .
            old save sequence
            . . .
```

The sequence starting at .svext is used whenever the EPL entry might possibly be called from a PL/I program (all external and internal procedure entries). The old sequence starting at .sv is used whenever the block can never be invoked by a PL/I program (begin blocks and bounds procedures).

The new sequence checks for the numbers 4 or 8 in the right half of the first word pointed at by the ap register; these denote, respectively, a PL/I program without and with the extra stack frame pointer required by internal procedures. If the special codes are not present, the call is assumed to be coming from an EPL procedure and the old save sequence is used. If either code is present, the call is assumed to come from a PL/I program and a jump is made to the special compatibility program pl1_to _epl.

pl1_to_epl performs the following functions:

a) If no descriptors are present, a return is made to bp|0 causing the standard sequence at .sv to be executed. Otherwise the following steps are performed:

b) pl1_to_epl performs the EPL standard save sequence (the stack size is available in register X7).

c) The new stack frame is extended by N words, where N is the size of the original argument list rounded up to the next higher multiple of 8. This space is used to create a new EPL style argument list.

d) Each argument descriptor on the PL/I call is examined. If the corresponding EPL data type requires a specifier and dope, they are created in space obtained by extending the stack as needed. If no specifier is required, the data pointer is copied into the new argument list.

e) The ap register and stack location 26 are set to point at the new argument list at the end of the EPL stack frame. pl1_to_epl then returns to the EPL program at location bb|0,0 thus by-passing the instructions at .sv.

Who set XR0 ?

5. THE CONVERSION OF EPL PROGRAMS TO PL/I PROGRAMS

This section is designed to assist Multics system programmers to convert existing EPL programs into PL/I programs. It may also be useful to the PL/I programmer who wishes to interface PL/I programs with EPL programs. The section is organized around a set of issues each of which is a source of possible incompatability between EPL and PL/I. ~~At the end of this section is a quick checklist which may be useful to programmers who are converting EPL programs to PL/I.~~ The reader is also urged to study section 3 which describes the PL/I call.

5.1. Fixed-point Division

In EPL object programs the division of fixed-point values yielded a floating-point result. The current version of the PL/I compiler does not impelment the divide operator with fixed-point operands. The compiler issues a diagnostic which describes how to accomplish fixed-point division using the divide built-in function.

implement

```
given:
    k=i/j
Rewrite as:
    k=divide(i,j,17,0);
```

where: 17 is the desired precision of the quotient and may be any integer constant between 2 and 35. The fourth argument must always be a zero.

5.2. The length built-in function

EPL programs which contain calls to the library routines lg$cs or lg$bs should be rewritten to use the length built-in function. The length built-in function always returns the current size of varying strings and the declared size of non-varying strings. The function is extremely efficient since it is usually compiled as a single lda instruction. PL/I does not implement any function which returns the maximum length of a varying string.

5.3. Bit string to arithmetic conversion

The PL/I precision rule governing the conversion of bit strings to fixed-point values is different from the EPL rule. This difference between the two languages does not effect most programs because the usual practice has been to perform the conversion through the use of a fixed built-in function. EPL programs which depend on the default bit string to arithmetic conversion rule should be modified.

PL/I rule: bit_strings are converted to a fixed-point binary value of precision 71.

EPL rule: bit_strings are converted to a fixed-point binary value of precision 63, except when the length of the string is a constant whose value is less than 36, in that case the resultant precision is the length of the string.

```
given:
    dcl b bit(10), a fixed bin (17);
       .
       .
       .
    a = a+b;
Rewrite as:
    a = a+fixed(b,10);
```

5.4. Use of the Returns Attribute

The EPL compiler allowed the attributes of the return value of a function to be written anywhere within the declaration of the function. The PL/I compiler requires that these attributes be written in the returns attribute. The EPL compiler will accept the returns attribute as well as the more permissive form.

```
given:
    dcl e entry bit(1);

Rewrite as:
    dcl e entry returns (bit(1));
```

5.5. The Significance of the Entry Attribute

The entry attribute may be written in either a short or long form, i.e., entry or entry(<Parameter descriptions>). The EPL compiler accepted both forms but processed the long form exactly like the short form. The PL/I compiler makes extensive use of the long form and it is essential that the PL/I programmer understand its meaning. The entry attribute has the following two uses in Multics PL/I:

If the attributes of the actual argument used in a call to this entry differ from the declared attributes of the corresponding parameter, a conversion will be done in order to force the argument to conform to the attributes specified in the parameter description.

The careless use of this feature will result in unexpected conversions and a call by value instead of a call by reference. In other words the argument will be converted to a temporary whose attributes agree with the parameter description and the temporary will be passed as the argument.

If the short form of the entry attribute is used, or if any extents of any parameter descriptions are specified by an * then all calls to the entry will contain argument descriptors. The presence or absence of argument descriptors does not need to concern the casual PL/I programmer, but it is of great importance for programmers who must interface with EPL programs or who are concerned about the efficiency of calls. The discussion of the PL/I call in section 3 provides a more detailed discussion of the use of argument descriptors.

Rules governing the conversion of arguments to user declared entry names:

a) Arithmetic scalars arguments are considered to match a parameter description only if all of the following attributes agree:

fixed
float
binary
@decimal
precision
scale

b) String scalar arguments are considered to match a parameter description only if all of the following attributes agree:

bit
char
varying
aligned
unaligned
declared length.

The declared lengths are considered equal only if:

the length declared in the parameter description was an asterisk.

@ A bug in the current compiler causes binary OR and decimal to be considered equivalent. This will be corrected in a later version of the compiler.

(or) to previous page

both the parameter description length and the argument length are declared as decimal integer constants of equal value.

c) Scalar locator arguments are considered to match a parameter description only if the following attributes agree:

pointer
offset

The area reference of the offset attribute is ignored for purposes of attribute matching. No conversions are performed on locator data. If the attributes do not match, the compiler will issue a diagnostic.

d) If the parameter description has been declared with the label attribute, the corresponding argument must be a label constant or scalar label variable. A mismatch will result in a compiler diagnostic.

e) If the parameter description has been declared with the area attribute, the corresponding argument must be a scalar area variable whose size matches the size of the parameter description. The rules for previously given string length matching apply to area sizes. The compiler issues a diagnostic if the attribute or size does not match.

f) If the parameter description has been declared with an entry attribute, the argument must be an entry name without arguments or enclosing parentheses.

g) If the parameter description describes a structure, the argument must be a structure whose component elements all match their corresponding elements of the parameter description. The compiler issues a diagnostic if any items do not match.

h) If the parameter description describes an array, the argument must be an array of identical bounds and dimensionality and whose elements match the elements of the parameter array according to the rules given for scalars and structures. The rules previously given for string length matching also apply to array bounds. A compiler issues a diagnostic if any items do not match.

5.6. Label Arrays as Transfer Vectors

The EPL compiler compiled declarations of the form

```
dcl s(5) label initial(a,b,c,d,e);
```

into a transfer vector. While this provided an efficient method of implementing a switch it was not legal PL/I. The PL/I compiler has no construct equivalent to the EPL transfer vector. It will compile the de-declaration of s into an automatic label array and will initialize it during the prologue. Transfers to elements of s will invoke the PL/I unwinder. This situation can be improved two ways:

1. All label variables which only contain local label values should be declared label(x) where x is any label constant in the block of declaration. Transfers to such labels will result in a direct transfer instead of an unwinder call.

2. To avoid initialization of a label array during the execution of the prologue, remove the initial attribute and declare the label array to be static. The array must be initialized by explicit code, but that code can easily be written so that it is done once per process.

   ```
   given:  dcl l(3) label initial(a,b,c);
   rewrite as:
           dcl l(3) label(a) static,
                    l_swt fixed static initial(0);
                    If l_swt = 0 then
                          do;
                          l_swt = 1;
                          l(1) = a;
                          l(2) = b;
                          l(3) = c;
                          end;
   ```

   If written as:

   ```
           dcl l(3) label(a) initial(a,b,c);
   ```

   The array will be initialized at each invocation of the procedure, but transfers of the form go to l(i) will by-pass the unwinder.

5.7. The Use of cv-string and char (*) Declarations

The lack of proper diagnostics in the EPL compiler resulted in the comilation of several illegal language constructs. Among these were declarations of non-parameter bit or character string variables whose length was declared as an asterisk.

[Handwritten annotations: "This must be a one. 1" — "don't use "l" as a sample" — "no space" — "Compilation"]

The procedure cv-string was written to use the results of the compilation of such strings. The procedure has a number of uses most of which can be replaced by an appropriate use of the substr built-in function. The substr built-in function is not only correct PL/I it is also generally more efficient than cv-string.

The char (*) feature was also used by the program command_agr. It is suggested that fetch_arg_ replace all uses of command_arg.

5.8. Simple Rules for Converting EPL Array or Structure Declarations to Equivalent PL/I Declarations

If a PL/I data declaration is expected to describe exactly the same arrangement of values in storage as an identical EPL data declaration, then the following three rules can be used to insure the desired compatability:

1. Level one arrays of non-varying strings must be declared with the aligned attribute.

2. A structure containing elements of dissimilar data types of which one or more are non-varying strings must be declared with the aligned attribute. (This describes any EPL structure which is not packed and which contains one or more non-varying strings.)

3. Because PL/I varying strings do not have the same storage representation as EPL varying strings, remove all varying string declarations.

It is necessary to follow these rules when ever the same data is accessed by both an EPL and a PL/I procedure. This circumstance occurs when:

1. The declaration is a include file incorporated into both an EPL and a PL/I program.

2. The declaration describes external static data accessed by both EPL and PL/I procedures.

3. The declaration describes an argument/parameter passed between EPL and PL/I.

4. The declaration describes a portion of based storage whose address is accessable to both an EPL and a PL/I procedure.

Note that the EPL compiler will accept the aligned and unaligned attributes but will ignore them.

5.9. The Effect of the Alignment Attributes on Storage Allocation and on the Packing of Arrays and Structures.

Readers who want a simple set of rules which insure compatability between EPL and PL/I structure or Array declarations should read Section 5.8.

5.9.1. Definitions:

The attribute "aligned" means that the variable to which it applies is allocated on a word boundary.

The Attribute "unaligned" means that the variable to which it applies is not necessarily allocated on a word boundary.

The term "packed" is not a source language attribute in PL/I but is used to describe a property of EPL or PL/I arrays and structures. A PL/I or EPL structure is packed if it contains only unaligned non-varying bit strings (or packed sub-structures of such strings), or if it contains only unaligned non-varying character strings (or packed sub-structures of such strings).

A PL/I Array is packed if its elements consist of packed structures or unaligned non-varying strings.

An EPL Array is packed only if it is a member of a packed structure.

To review briefly - aligned and unaligned describe the storage boundary on which a variable is allocated, and packed describes a particular type of array or structure. It is thus possible for a structure or array to be both packed and aligned-meaning that it is allocated on a word boundary but is packed according to the previous definition.

5.9.2. The Storage Allocation Rules of the PL/I Compiler:

All level one variables are allocated storage beginning on a word boundary or a multiple of a word boundary.

Aligned variables and all non-string or non-structure variables are allocated storage beginning on a word boundary or a multiple of a word boundary.

Unaligned strings or unaligned packed aggregates are allocated storage on the nearest bit or character boundary.

The Effect of the Alignment Attributes on the Accessing of Based Variables and Parameters

There are two situations in the PL/I language which permit data allocated on arbitrary bit boundaries to be accessed through the use of another declaration: A reference to a parameter, and a reference to a based variable whose address was derived by the addr function. In both instances a level one variable can be used to describe data residing on an arbitrary bit address.

The PL/I language requires that the alignment attributes of arguments and parameters agree. It imposes the same requirement on the based declaration and the item it represents (the item whose address was derived via the addr function).

The requirement that the alignment attributes agree in these cases permits the PL/I compiler to generate more efficient accessing code for referenced to based variables or parameters.

<u>It is absolutely necessary that the Multics PL/I Programmers insure that the Alignment Attributes of Arguments/Parameters and based Variables/Arguments of addr agree</u>. The following discussion of the Multics PL/I pointer should make the reason for the requirement clear.

The Multics PL/I pointer consists of a machine address (segment number and word offset) and a bit offset within the word selected by the machine address. Accessing code used to reference aligned parameters or based variables ignores the bit offset since the data is known to be allocated on a word boundary. Accessing code used to reference unaligned parameters or based variables uses the full pointer including the bit offse. offset.
If an attempt is made to reference an unaligned variable through the use of an aligned declaration the bit offset will be ignored and the access will be incorrect.

6. Tips on Writing Efficient PL/I Programs

This section is designed to help Multics PL/I programmers write efficient programs. The discussion is organized around a set of specific issues each of which is a potential cause of poor object code. All statements about the nature of the object code refer to the object code produced by the Multics PL/I compiler and may not be valid for some other implementation of the PL/I language.

6.1. General Comments

Usually the most natural way of writing a PL/I Program is the most efficient. For example, if a valve is in fact a bit string, it should by declared as a bit string. Of a varying character string is the most convenient representation for a valve then it should be declared as a varying character string. Similarly the choice between using a structure or an array should be make on the basis of which construct seems most "natural" to the problem. Do statements should be employed wherever they are convenient; the "do;" form is particularly nice since it generates no code.

6.2. The Use of Entry Declarations

All entries called by a PL/I procedure should be explicitly declared by declare statements in the calling procedure. If the called procedure is a PL/I procedure or Fortran subprogram the entry declaration should completely describe all parameters of the entry. The attributes used in the entry declaration must agree exactly with the attributes declared for the actual parameter in the called procedure. The use of complete entry declarations results in more efficient calls to the entry.

If the called procedure is an EPL procedure and one or more of its parameters is a string or an aggregate then the entry declaration in the calling procedure must not contain a description of the parameters.

Refer to section 3 for a discussion of the PL/I call and argument passing conventions.

6.3. Parameters

Asterisk extents should not be used unless they are really necessary because they force all calls to the procedure to contain agrument descriptors, and calls with argument descriptors are twice as long as calls without argument descriptors.

Accessing a parameter is equivalent to a simple pointer qualified reference of the form p->X. So generally nothing is gained by copying a parameter into a working variable. However, if the parameter is an unaligned string it is definitely more efficient to copy the string into an aligned working string variable. <u>String parameters should be declared "aligned" if and only if they never receive an "unaligned" argument.</u>

6.4. Begin Blocks

Begin blocks are implemented as internal procedures and carry the full cost of the recursive call mechanism. Unless the users wishes to define a new set of internal variables a do; . . . end; construct can be used in place of a begin block. The call created to invoke a begin block is <u>slightly</u> more efficient than the call to an internal procedure, so begin blocks may be used instead of internal procedures where possible.

6.5. Internal Procedures

The call used to invoke an internal procedure is nearly equivalent to the call used to invoke external procedures. Because the internal procedure is able to access variables belonging to its containing procedures the internal procedure offen requires fewer arguments than an equivalent external procedure. It also shares its constants with the containing procedures. For these reasons internal procedures are usually more efficient than an equivalent external procedure.

6.6. Accessing Code

6.6.1. The effect of block structure on accessing

A reference to an automatic or parameter variable declared in a block other than the block making the reference is called a non-local reference. These references are equivalent to a reference of the form:

```
p1->p2->p3 . . . pn->x
```

where n is the number of blocks between the reference and the declaration. References to static or based variables declared in an outer block are just as efficient as references occuring within the block of declaration.

6.6.2. Static Storage References

A reference to an internal static variable is equivalent to p->x. References to external static variables are equivalent to p->x on all references except the first. First references to external static variables cause a linkage fault.

6.6.3. References to based Storage

Simple based references of the form p->x usually result in an indirect address type of reference. References which use multiple pointer operators.

(i.e.p->q->r->x) result in the use of as many base register load instructions are there are pointers in the reference. References to based aligned string variables are much more efficient than references to based unaligned string variables. A based string variable should be declared "aligned" if and only if it is never used to describe an "unaligned" string.

6.6.4. The Design of Arggregates and Their Effect on Accessing Code

Elements of arrays or structures have what is known as an offset. The offset of an element is the distance of the element from the beginning of its level one containing aggregate. An items offset is either expressed in words or in bits depending on the nature of the elements which preceed it. The offset of an "aligned" item is always expressed in words. The offset of an "unaligned" item is expressed in bits and words if it is immediately preceeded by a string variable or packed aggregate. The design of structures and arrays should be influenced by the following:

1. Constant word offsets result in the best accessing code.

2. Constant offsets are more efficient than variable offsets.

3. Word offsets are better than bit offsets.

4. A variable offset developed from items whose sizes are expressed in the same terms is more efficient than a variable offset developed from items whose sizes are expressed in different terms.

An example of the fourth point is given below.

Example:

```
dcl 1 s,
      2 a(n) float,
      2 b(m) char(k) unaligned,
      2 c area(J),
      2 d bit(kk),
      2 x ptr;

dcl 1 ss,
      2 a(n) float,
      2 p(m) ptr,
      2 c area(J),
      2 x ptr;
```

References to ss.x are more efficient than references to s.x because all the items which preceed ss.x are items whose size is expressed in words or multiples of words.

The offset of a string variable is the distance between the string and its level one containing aggregate, or it may be the distance between a sub-string described by a "substr" built-in function and the beginning of the string, or it may be the distance between the beginning of a string array and some element of that array. A string's offset may also be a combination of any of these three offsets.

6.7. The Efficient Use of String Data

The efficiency of PL/I String Operations depends on the offset and length of the strings used in the operation. Offsets are discussed in section 6.6.4.

The length of a string is declared by the programmer. Varying strings have two lengths: the declared length determines the amount of storage allocated for the string and represents the maximum size of the string. The current length is not declared by the programmer but is initially set to zero by the compiler or by compiled code. Each assignment to a varying string updates the value of the current length. In this discussion a varying string should be considered to be just as efficient as a variable length non-varying string. The following general guidelines should be considered when using string variables:

1. Operations on constant length non-varying strings whose length is less than or equal to 72 bits or 8 characters are performed by in-line code and are more efficient than other string operations.

2. Operations on variable length non-varying strings, varying strings and constant length non-varying strings are all equally efficient.

3. The nature of the string offset is generally more significant than the nature of the length or the difference between varying and non-varying.

4. Operations on unaligned non-varying parameters or based strings produce the worst accessing code.

The accessing of string variables is improved greatly if the variables are declared "aligned" this is particularly true of parameter and based string variables. However, if a parameter or based string variable serves as the image of an "unaligned" string the parameter or based string must be "unaligned".

6.8. Label Variables

Multics PL/I implements Label Variables whose value is known to be restricted to a set of label constants all of which are labels of statements within the same block much more efficiently than it does label variables whose valves are unrestricted.

Example:

```
    dcl lab label(a,b,c);

    dcl lb label;
```

Where: a,b and e are Labels of statements within the block containing this declaration.

Transfers to label lab which occur in the block of declaration do not invoice the PL/I unwinder and are nearly as efficient as transfers to a label constant. Transfers to lb invoke the unwinder and are costly.

6.9. Use of the Initial Attribute

The initial attribute is implemented for all storage classes. Internal static variables are initialized by the compiler and generate no code in the object program. External static variables whose names do not contain a dollar sign are initialized on the occurance of the first reference within the process. The initialization is done by copying a pre-initialized image from internal static storage into the external static variable. This implementation is reasonably quick but the space used to contain the image may be quite large in the case of large initialized arrays or structures. Note that static label variables cannot be initialized.

Based and automatic variables are initialized when they are allocated. The initialization is done by means of assignment statements similar to those written by the programmer. ~~Array~~ Initialization of based or Automatic Arrays can often be more efficiently done by the programmer using multiple assignment statements.

6.10. Multiple Assignment Statements

Multiple assignment statements whose left side elements have identical attributes are very efficient and should be used where possible. Multiple assignment statements whose left side elements have different attributes are no worse than separate assignment and in many cases are slightly better than separate assignments.

Example:

```
i,j,k,l = 0;
```

Generates:

```
lda  zero
sta  i
sta  j
sta  k
sta  l
```

7. The Implementation of PL/I Storage Classes

The storage mechanism used to contain the values of PL/I variables depends on the declared storage class of the variable.

7.1. Automatic Storage

Variables whose storage class is automatic are allocated upon entry to the block in which they are declared. The space used by automatic variables is provided by the stack associated with the current process. Upon block entry this stack is extended by an amount sufficient to contain all automatic variables declared in the block; upon return from the block the stack is reverted or popped releasing the storage occupied by the automatic variables. All compiler produced temporaries are also allocated in the stack. String temporaries whose size exceeds two words are allocated just before the execution of the statement in which they are used, and are released upon completion of that statement.

7.2. Internal Static

Internal static variables are assigned locations in the procedure's linkage section by the compiler. Any initial values are established by the compiler at the time the space is assigned. When the procedure is first invoked the linkage section is copied into the combined linkage segment associated with the current process. This action has the effect of allocating and initializing all internal static variables declared in the procedure. Subsequent invocations of the procedure in the same process do not re-initialize internal static storage.

7.3. External Static

External static variables whose names do not contain a dollar sign are allocated and initialized by the procedure which first references them. Space for these variables is created at the time of their allocation in a segment named stat_$stat_. This segment is created in the users process directory when it is first needed and remains there until the process dies. The "rename" option may be specified on a

procedure statement and may be used to change the name of the segment used for the allocation of external static variables. Refer to the Multics PL/I Language Specification. External static variables whose names contain a dollar sign are assumed to be previously allocated in a segment found in the user's current working directory. A variable named

a$b

is assumed to be located in segment a at an offset named b. A variable named:

a$

is assumed to be located at the beginning of a segment named a. Initial attributes specified for these type of external variables are ignored.

7.4. Based Storage

Based variables are allocated storage and initialized by the execution of an allocate statement. The storage used to contain the value of a based variable is released by the execution of a free statement. If an area variable is specified in an allocate statement, storage for the based variable is allocated in the area variable. If no area variable is given in the allocate statement, a system supplied area named free_$free_ is used. free_$free_ ia an area variable 64K words in size which occupies an entire segment. The first use of the area causes the segment to be created in the user's process directory. It remains there throughout the life of the process. Unless variables are explicitly freed by the execution of a free statement the free_$free_ segment will continue to fill up with based variables throughout the life of the process.

Three run time support routines are used by the PL/I object code to manage space within area variables. All areas except free_$free_ are automatically initialized to the empty state. The initialization is done at compile time if the area is located in internal static storage; otherwise it is done by the "empty" built-in function. The empty built-in function invokes the "area_" run time routine. The execution of an allocate statement invokes the "alloc_" run time routine, and the execution of a free statement invokes the "freen_" run time routine. Reference BP.4.02.

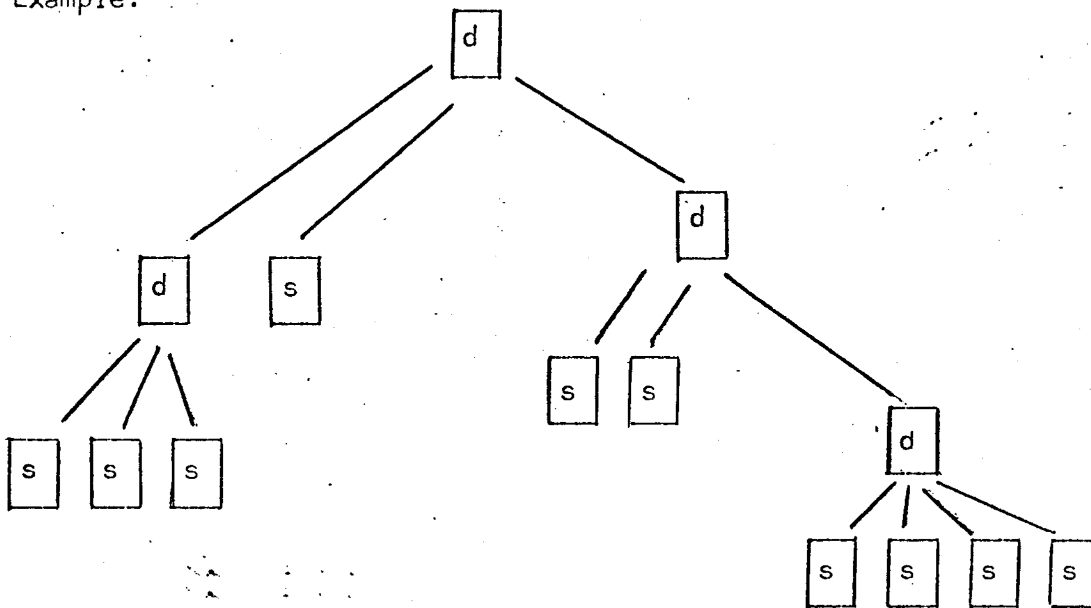
8. The Fundamentals of Multics for PL/I Programmers

This section provides the user with a brief and simplified description of several fundamental characteristics of Multics. Terms such as process and working directory which are used elsewhere in this document are defined here. Experienced Multics users may skip this section.

8.1. Segments and Directories

The address space in which an object program executes is organized as a set of segments each of which is a linear address space 64K words long. Each segment is identified by a name and belongs to a directory. A directory is a list of segment names and other segment attributes. It is itself a segment and belongs to another directory. A segment or directory belongs to only one directory - resulting in a well defined tree structure of segments and directories as shown below:

Example:



Each segment labeled s is a 64K address space identified by a unique name within its directory. Each segment labeled d is a directory containing the names and attributes of its member segments. Directory segments are named members of a directory must like any other segment.

The name of a segment need only be unique to its directory - two segments may have the same name if and only if they belong to different directories.

8.2. A Process

For our purposes a process can be considered to be an execution activity which begins when a user logs in and continues until he logs out, or

until he explicitly terminates the process by starting another process through the use of the "new-proc" command. A Multics process corresponds to a "task" in certain other operating systems.

8.3. Dynamic Linking

References to any portion of the address space consist of a segment name and a location within the segment. In order to increase the efficiency of a storage reference a segment name becomes associated with a segment number when the segment name is first referenced within a process. A segment number is merely an alias for the segment name, but it is more easily translated into a storage address by the system. The association between a segment, its name and its number is retained throughout the life of the precess.

If program *a* references program *b* by means of a call or function reference a link is established between *a* and *b* such that all subsequent references to *b* by *a* will be accomplished by using the segment number of *b* instead of the name *b*. A similar link is established if program *a* references data contained in segment *c*. The establishment of these links during the execution of the program is called dynamic linking.

8.4. The Search Mechanism

At all times ~~the system~~ considers one directory to be the current working directory. The user designates the current working directory through the use of the "change_wdir" command. ~~At the initiation of each process the user must establish a current working directory.~~

When a segment name is referenced during the execution of a program the operating system performs a search algorithm which identifies the segment and associates a segment number with the segment name. The search mechanism first determines if the name has been referenced during this process; if it has, the segment which was orginally referenced is used, otherwise, the working directory is searched for a segment whose name is that used in the reference. If no such segment is found the search continues in one or more system directories which contain library procedures and portions of the operating system. When a segment is found it is given a uniqque segment number and a link is established between the segment containing the reference and the referenced segment.

8.5. The Hidden Dangers of Dynamic Linking

The user who is unaware of the fundamental workings of the dynamic linking mechanism may experience unexpected difficulties when he attempts to execute programs in Multics. Most problems are caused by the fact that the system maintains the association between a segment name and its segment number throughout the life of the process. For example if a user executes program *a* which calls a library procedure *X*, and the user then changes to a new working directory, executes program *b* which calls a procedure *X* located in his new working directory, the system will establish a link

to the original library segment because the name X is still associated with the orginal segment and segment number. A more frequently encountered problem occurs during program de-bugging: Suppose that the user compiles a program Z and executes it by calling it from the console or from another program. He discovers an error and re-compiles the program. If he then calls the program as he did the first time he will not get the new version but will instead get the old version because the name Z is still associated with the orginal segment and segment number.

Since the operating system itself consists of ~~separate~~ named segments it is also possible for the user to inadvertently link to an operating system segment rather than to his own segment. This problem can be avoided by using segment names which do not end with an underscore character "_". All system segment names should end with underscore.

The first problem we discussed could have been avoided by not switching working directories in the middle of a process. The second problem can best be avoided by using distinct names for each version of a program.

The "new_proc" command can be used to erase all previous associations between segment names and segment numbers. It is somewhat time consuming because an entirely new set of associations must be built up during the execution of the new process, but it often is the only way out of a complicated situation.

8.6. A Process and the Execution of a PL/I Program

The Multics Process has a number of interesting side effects which are of concern to the PL/I programmer.

1. Static variables declared with initial attributes are initialized once per process. Subsequent executions of a procedure within the same process do not cause initialization to occur.

2. External variables are allocated and initialized once per process. Similarly external file names and condition names are established for the life of the process. (The external name X cannot be used for more than one purpose within a process).

3. Based variables allocated in the default area free_$free_ remain allocated in that area throughout the life of the process, unless explicitly freed by the execution of a free statement.

Appendix A is not Available

Appendix B is not Available