G0081 1/20/69
R. FREIBURGHOUSE

COMPATIBILITY CONSIDERATIONS OF THE PL/I IMPLEMENTATION

Considerable interest has been expressed concerning the extent to which the implementation of PL/I is compatible with the implementation of EPL and related Multics standards. This document is an attempt to satisfy this interest and at the same time to explain certain aspects of the PL/I implementation. The companion document titled "SIGNIFICANT FEATURES OF MULTICS PL/I" describes more completely the advantages of the PL/I implementation.

The issue of compatibility should be separated into two areas of discussion. Syntactic differences in the two languages represent one type of incompatibility. Implementation differences which affect the semantics of the language are a second type of incompatibility. Our discussion considers both of these areas of difference.

1. Syntactic Incompatibility

The evolution of the PL/I language has resulted in considerable improvement in the language and has also made previous definitions of the language obsolete in the sense that they are no longer subsets of the current language.

The language of the Multics PL/I implementation is defined by IBM publication Y33-6003-0. A number of features (listed in Appendix 1) are not implemented in the initial compiler. Certain more exotic features (tasking, sterling data, etc.) will never be implemented. A more precise description of the implemented language will be available at a later date.

Conversion of an EPL Program to a PL/I Program

A Multics command will be available which will process an EPL program issuring warnings of syntactic imcompatibilities. It is expected that the programmer will make the indicated modifications to his program. The number of syntactic imcompatibilities is reasonably small and is given in Appendix 2 of this document.

2. Implementation Strategies and the extent of Semantic Incompatibility

The PL/I implementation differs from the EPL implementation in several aspects. The use of new strategies in these areas was deemed necessary in order to achieve a higher degree of object program efficiency. The PL/I implementation of varying strings, on conditions, and argument descriptors differs from EPL's implementation of these features.

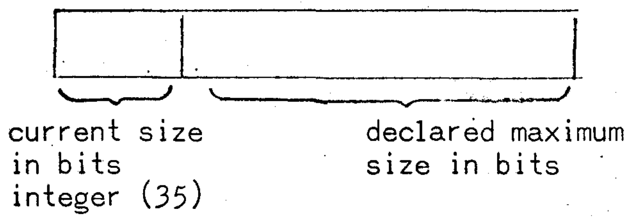
2.1 ON-Conditions

ON conditions are less important than the other two areas so we will dispense with them first. The PL/I implementation of the condition prefix and on-unit enabling uses the main stack. It is done in such a way as to make these features extremely efficient (there are no epilogues and no enabling calls in the prologue). However, the scope of an enabled on-unit is limited to the ring in which it was enabled. If a PL/I programmer wishes to invoke the Multics signalling machinery, he must use explicit calls to that machinery. Signalling across rings can be easily done by the appropriate use of both the system machinery and the PL/I implementation. A detailed explanation of this mechanism will be available at a later date.

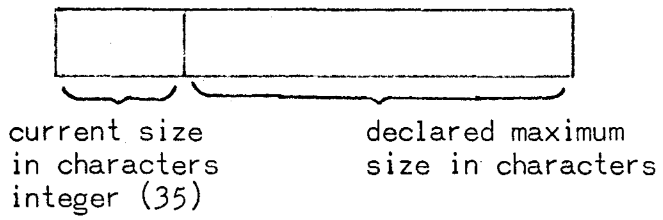
2.2 Varying Strings

The PL/I implementation of varying strings incorporates two new data types (521 and 522).

PL/I varying bit string (521)



PL/I varying character string (522)



Note: The amount of storage allocated to a varying string is always an integral number of words. The address of a varying string is the address of the data. The address of the current size is the address of the data minus one.

This implementation has the following advantages:

1. No epilogue is needed to de-allocate automatic varying strings or string temporaries.
2. This design allows string temporaries to be created in such a way as to appear to be either varying or non-varying strings.
3. Operations on varying strings are just as efficient as operations on non-varying strings.
4. No specifiers or dope are needed to operate on the data. The declared size is available to the program from the declaration. The current size and data are addressable with a single pointer.

The primary disadvantage of this implementation is that the new data types are not acceptable to EPL nor are EPL style varying strings acceptable to PL/I. Users are urged to switch to PL/I and use varying strings rather than adopt some other strategy. A second and intrinsic disadvantage of this implementation stems from the fact that it allocates the declared maximum amount of storage.

2.3 The PL/I Call Interface

The calling sequence produced by the PL/I compiler will be 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 will be 0 for calls between external procedures and 2 for calls to internal procedures. PL/I will use the codes 4 and 8 to indicate these same conditions.

For the purpose of argument compatibility, it is essential for PL/I and EPL 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.

2.3.1 Arguments

The argument pointers of the PL/I call will 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 will refer to the first word which contains the data. The pointer will contain the bit offset necessary to address the data. The data types of PL/I are listed in Appendix 3 of this document.

The format of a pointer is that of a 645 ITS pair with a possibly empty bit field located in the second word. No incompatibility is introduced because of the presence of this bit field.

2.3.2 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 was declared with an * extent. The only conditions which will cause the PL/I compiler to produce descriptors are:

1. The parameters of the entry have not been described through the use of an entry (<description>) attribute in the calling program.
2. An entry (<description>) attribute has been specified in the calling program but one or more of the parameter descriptions contain an * extent.

The design of the compiler, the definition of new string types, and the use of argument descriptors has enabled the compiler to eliminate all dope and specifiers and achieve the following design objectives:

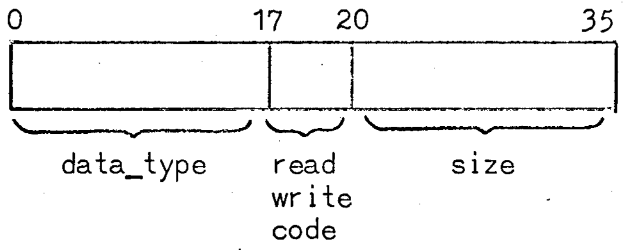
1. All data, including members of adjustable aggregates, is addressed directly by efficient in-line code. If the address is constant it is computed at compile time. If it is a mixture of constant and variable terms the constant terms are combined at compile time.
2. Substr is always done in-line as a part of the normal addressing function.

3. All string operations are done by in-line code or by "tsb" type subroutinized code. No descriptors or calls are produced for any string operation.
4. Epilogues are never created.

2.3.4 Descriptor Formats

Descriptors are used to implement the * extent feature of the PL/I language. They will also serve to allow argument compatibility between PL/I and EPL and will implement the call back feature. The design of the descriptors is an extension of the design given in BD.7.02 and is compatible with it. The descriptor pointers will refer to argument descriptors of the following forms:

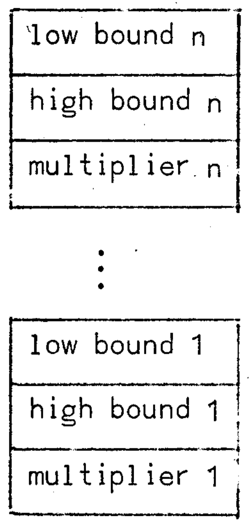
Basic Descriptor



All arguments (scalars or aggregates) will 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.

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:



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 is the same as the relative position of the data it describes. Appendix 3 gives the complete list of PL/I data types.

2.4 Argument Compatibility between PL/I and EPL

The previous discussion may have created the impression that EPL and PL/I are quite incompatible because of differences in the use of dope specifiers and descriptors. This is not the case since all Multics standard data types (except varying strings) can be passed between PL/I and EPL or between EPL and PL/I.

The PL/I entry is sensitive to the kind of call it is receiving. When called by EPL (or an EPL like call) it will map EPL specifiers and dope into its own desired form. In this way all the Multics standard data types, except long and short varying strings, can be passed to PL/I programs.

In the near future the EPL compiler will be modified so that its object programs will recognize a PL/I call and will map the PL/I descriptors into specifiers and dope. Varying strings will not be mapped. Note that the PL/I call must include descriptors for this mapping to occur.

Arguments passed between PL/I and EPL or between EPL and PL/I through the use of external static or the use of based declarations in the called program may contain any PL/I data types except varying strings. A more complete discussion of the use of PL/I and a description of the process of converting an EPL program into a PL/I program will be available at a later date.

Summary of Legal Argument Types

Calls between PL/I and EPL or between EPL and PL/I may include arguments which are:

1. arithmetic scalars
2. pointers
3. labels
4. non-varying strings
5. entry variables
6. 1 dimensional arrays of items 1 to 4

these calls may not include:

1. structures
2. multi-dimensional arrays
3. areas
4. varying strings

<u>The Use of the Validate Option</u>

The validate option of PL/I and EPL can be used to perform argument mapping between PL/I and EPL procedures. This device provides for nearly complete argument compatability. A detailed discussion of this feature will be available in future documentation.

APPENDIX 1

LANGUAGE FEATURES NOT IMPLEMENTED IN THE FIRST VERSION OF THE PL/I COMPILER

1. All input/output features including all related statements, declarations, on conditions and builtin functions.
2. Sterling data and pictured data.
3. Tasking - all related options, declarations, on conditions and builtin functions.
4. Scaled fixed point arithmetic.
5. Complex arithmetic.
6. Precision controlled arithmetic.
7. Decimal arithmetic is implemented as binary with the appropriate conversion of the declared precision.
8. Controlled storage class.
9. The attributes: defined, position, like, cell, generic.
10. Conversion between character string and arithmetic and between arithmetic and character string.
11. Aggregate expressions and array cross sections.
12. Check and size condition prefix.
13. Some builtin functions are omitted but the EPL subset is available.
14. Division of fixed point values must be done using the divide function.
15. Prologue dependencies are not resolved. Values available upon entry to a block do not include values declared as automatic in that block.

APPENDIX 2

SYNTACTIC DIFFERENCES BETWEEN PL/I AND EPL

1. EPL allows expressions to specify the extents of a parameter. PL/I allows only constants or asterisk. Replace all expressions with *.

2. If the return value of a function is to be specified in the declaration of that function, those attributes must be enclosed in a returns ( ) attribute. EPL allows this form but also allows these attributes to be written anywhere.

3. The default packing rules for the two languages are different. An EPL structure which would not be packed under EPL rules must be declared aligned to achieve the same result.

4. The precision of a default bit to fixed point conversion in PL/I is always 71. In EPL the precision is 63 when the length of the string is not a constant, and is the length of the string when the string size is a constant.

5. The extents specified for the return value on an entry or procedure statement must be constant. An asterisk will work but will cause a diagnostic since it is not legal PL/I. EPL allows anything to be given for an extent.

7. EPL does not diagnose the use of an * extent specified for a non parameter string. This unfortunate circumstance has led to some gross language violations on the part of many EPL users. This device has mainly been used in conjunction with a routine cv-string to enable a varying string to be referenced as if it were a non-varying string. The use of <u>substr</u> is the proper way to achieve the same effect. The PL/I implementation of substr is extremely efficient and thus should be used for this case.

8. PL/I will not implement the divide operator for fixed point data. The divide function must be used to divide fixed-point values. EPL produces a floating-point result for fixed-point division.

APPENDIX 3

DATA TYPE CODES USED IN PL/I DESCRIPTORS

1	single precision real integer	
2	double precision real integer	
3	single precision real floating-point	
4	double precision real floating-point	
5	single precision complex integer (2 words)	
6	double precision complex integer (4 words)	
7	single precision complex floating-point (2 words)	
8	double precision complex floating-point (4 words)	
13	pointer data	
14	offset data	
15	label data	
16	entry data	
17-24	arrays of types 1-8	
29-31	arrays of types 13-15	
514	structure	data types which are not Multics standard
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	