ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

THE FORTRAN-PL/I INTERFACE PROBLEM

by

Michael T. Matthies

Applied Mathematics Division

November 1968

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission

A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

TABLE OF CONTENTS

	<u>Page</u>
ABSTRACT	5
I. INTRODUCTION.	5
II. ATTRIBUTES OF COMPATIBLE DATA	6
III. ADDRESSING SCHEMES	7
IV. EXPLICIT PASSAGE OF DATA THROUGH DOPE VECTORS .	9
V. IMPLICIT PASSAGE OF DATA	13
VI. COMMUNICATION OF DATA TO FORTRAN	17
VII. FORTRAN-TO-PL/I INTERFACE	18
VIII. PL/I-TO-FORTRAN INTERFACE	22
APPENDIXES	
A. FORTRAN-to-PL/I Programming Example	24
B. PL/I-to-FORTRAN Programming Example	28
BIBLIOGRAPHY.	30

THE FORTRAN-PL/I INTERFACE PROBLEM

by

Michael T. Matthies

ABSTRACT

This report describes the implementation of two techniques that provide for the establishment of a FORTRAN-PL/I interface. The report is not intended as a programmer's guide, but rather as a thorough discussion of the linkage and communication areas that allow PL/I and FORTRAN to be dynamically combined. The means for explicit and implicit passage of data types compatible to both languages is examined. This requires a study of two types of PL/I object time data-declaration blocks, namely, the string and array dope vectors. A brief discussion of the aspects of the internal control structures of PL/I and FORTRAN, related to linking the two languages, is also included.

I. INTRODUCTION

With the arrival of the PL/I compiler, many useful facilities, such as bit and character manipulation, have now been made available to the scientific programmer at a high programming level. To simulate these facilities before PL/I usually required sophisticated programming techniques in FORTRAN or assembly language.

Now, through PL/I, the programmer has available to him methods that accomplish simply what may have previously been rather complex. The hope of making these PL/I facilities available to FORTRAN programmers has stimulated the interest in dynamically combining PL/I procedures with FORTRAN routines.

On the other hand, there is a considerable advantage to the PL/I programmer in including existing FORTRAN routines, which have been developed over the years, in the PL/I library. The ability to use the efficiencies of FORTRAN to overcome the current deficiencies in PL/I adds another motive for combining FORTRAN routines with PL/I procedures.

To avoid extraneous programming effort and to use the efficiencies of both languages, the dynamic combination of PL/I and FORTRAN becomes most desirable. However, the incompatibility between object routines

currently produced by the compilers for IBM-360 FORTRAN and PL/I has drastically reduced the advantageous use of PL/I and FORTRAN routines in combination. This report is concerned with this problem of the FORTRAN-PL/I interface.

The problem is considered in two parts, namely, FORTRAN to PL/I and PL/I to FORTRAN. The specific areas of incompatibility in each are discussed, and reasonable approaches are presented for handling the difficulties arising from dynamically linking the routines and establishing communication between them.

II. ATTRIBUTES OF COMPATIBLE DATA

Before we discuss the interface problem, a presentation of the data-type facilities that serve as the objects of our communication may also serve as a basis for understanding the nature of the problem. It is important to know what data facilities are available as well as how to communicate data types.

The data types compatible between FORTRAN and PL/I can be grouped as arithmetic data and string data. The attributes of the arithmetic data-type items can be summarized as follows:

<u>FORTRAN</u>	<u>PL/I</u>
INTEGER *4	BINARY FIXED (p,0) with $0 < p \leq 31$
REAL *4	BINARY FLOAT (p) with $0 < p \leq 21$ or DECIMAL FLOAT (p) with $0 < p \leq 6$
REAL *8	BINARY FLOAT (p) with $22 < p \leq 53$ or DECIMAL FLOAT (p) with $7 \leq p \leq 16$
COMPLEX *8	COMPLEX DECIMAL FLOAT (p) $0 < p \leq 6$
COMPLEX *16	COMPLEX DECIMAL FLOAT (p) $7 \leq p \leq 16$

By declaring a PL/I data item with the attribute BINARY FIXED and scale factor zero, one has the FORTRAN four-byte integer. Half-word binary integers are not compatible with PL/I since all data items with BINARY FIXED require four bytes of storage. It is thus recommended that all

BINARY FIXED data items be given precision (31,0). The DECIMAL FIXED attribute is not compatible with any FORTRAN data type.

Floating-point data are completely compatible between the two languages. PL/I, however, allows this type of data to have DECIMAL or BINARY base. The internal representations of both types are identical.

Complex data can be either FIXED or FLOAT in PL/I, but only floating in FORTRAN. The floating-point representations are identical in the two languages, the imaginary part following the real part.

There are two types of string data items--character and bit. Characters are stored one per byte in S/360. PL/I can access each character in the string individually; FORTRAN is only able to access each group of four. To maintain a physical equivalence, a PL/I character string of length n must be defined in FORTRAN as an array with dimension $[n/4]$, where the brackets represent the smallest integer greater than or equal to the quotient.

Example

PL/I: STRG CHARACTER (80)

FORTRAN: STRG (20)

The maximum length allowed for variables with the CHARACTER attribute is 32,767. Bit strings are stored eight bits per byte. The attributes of BIT(8) and BIT(32) in PL/I correspond, respectively, to LOGICAL*1 and LOGICAL*4 in FORTRAN. The maximum length allowed for a bit-string variable is 32,767.

The main compatibility problem in communicating data types between FORTRAN and PL/I arises with respect to multidimensional arrays. In PL/I, arrays are stored so that the rightmost subscript varies most rapidly; in FORTRAN, arrays are stored so that the leftmost subscript varies most rapidly. Clearly, one-dimensional arrays pose no problem, but for multidimensional arrays we must provide some technique to overcome this array-addressing problem that does not involve serious restrictions in its implementation.

III. ADDRESSING SCHEMES

To find the address of the array element $A(S_1, S_2, \dots, S_n)$, the following scheme is used:

$$\text{Addr}[A(S_1, S_2, \dots, S_n)] = \text{ADDR}[A(0, 0, \dots, 0)] + L_B \left(\sum_{i=1}^n S_i M_i \right),$$

where L_B is the length in bytes of each array element. $\text{Addr}[A(0,0,\dots,0)]$ is obtained by an extrapolation and does not correspond to an actual array element.

In PL/I, we have

$$M_n = 1 \text{ and } M_i = \prod_{j=i+1}^n D_j, \quad i = 1, 2, \dots, n-1,$$

where D_j is the value of the j th declared dimension of the array A .

In FORTRAN, the difference appears with

$$M_1 = 1 \text{ and } M_i = \prod_{j=i-1}^n D_j, \quad i = 2, 3, \dots, n.$$

One fact we note, however, is that the PL/I array is the transpose of the FORTRAN array; that is, $A(S_1, S_2, \dots, S_n)$ in PL/I corresponds in position to $A(S_n, S_{n-1}, \dots, S_1)$ in FORTRAN. This leads us to the following approach to resolve the addressing difference.

If we have a PL/I procedure calling a FORTRAN routine, we would declare the dimensions of the FORTRAN array in reverse order from those of the PL/I array and reverse all subscripts.

Example

```
PL/I:      DECLARE K(3,2) FIXED BINARY (31);
FORTRAN:   DIMENSION K(2,3)
```

If we wanted to place in $K(I,J)$ the factor $2*I+J$, where I and J are the PL/I row and column indices, respectively, we would write

```
PL/I:
      L: DO I = 1 TO 3;
          DO J = 1,2;
              K(I,J) = 2*I + J;
          END L;

FORTRAN:
      DO 1 I = 1,3
      DO 1 J = 1,2
1      K(J,I) = 2*I + J
```

If we have a FORTRAN routine calling a PL/I procedure, we could use the technique described here or use an additional device that involves the DEFINED attribute with iSUB variables. The DEFINED attribute specifies that the defined item is to occupy the same storage area as that assigned to other data, called the base item. It is thus similar to the EQUIVALENCE statement of FORTRAN. The iSUB variables are dummy variables representing the subscripts of the ith dimension of the defined array. The base element corresponding to a defined element is obtained by replacing each iSUB variable in the subscript list of the base variable by the integer value of the ith subscript of the defined element.

Example

```
FORTRAN:  DIMENSION K(2,3)

PL/I:     DECLARE KBASE (3,2) FIXED BINARY (31,0),
           K(2,3) FIXED BINARY (31,0) DEFINED KBASE
           (2SUB,1SUB);
```

We declare the dimensions of the PL/I array KBASE in reverse order from those of the FORTRAN array K. We define K onto KBASE using iSUB variables, where 1SUB (2SUB) represents the subscripts of the first (second) dimension of the defined array K. Consider now the element K(I,J) of the defined array. For this element, I is the value of 1SUB and J is the value of 2SUB. Replacing these iSUB values in the subscript list associated with the base array, we obtain KBASE(J,I) as the base element corresponding to K(I,J). But KBASE(J,I) corresponds to the FORTRAN array element K(I,J). Thus K(I,J) represents the same element of the array whether it appears in FORTRAN or PL/I.

It will now be advantageous to consider the interface problem in two parts. The establishing of an interface for linking PL/I procedures to FORTRAN routines will be considered first; later the reverse problem will be discussed.

IV. EXPLICIT PASSAGE OF DATA THROUGH DOPE VECTORS

In Section III, we discussed techniques for generating source code that will refer to the same area in memory and assignment statements that refer to the same location. We now want to consider how to establish a communication link that enables areas of storage to be addressed by invoked PL/I procedures and the invoking FORTRAN routine.

The first means of communication to consider is the explicit passage of data items as parameters in a FORTRAN CALL statement. No problem exists for compatible scalar data types[†] since both languages handle this

[†]See Section II.

type of data in an identical manner. The use of arrays and strings as formal parameters produces the main interface problem in explicit data communication.

PL/I requires the creation of an object time data-declaration block, called a dope vector, when explicitly passing strings and arrays. The address of this dope vector, rather than the array or string address, is passed to the PL/I procedure. To establish a better understanding of the problem and its solution, we present a brief discussion of the array and string dope vectors.

An array dope vector contains the dimension information for the array that enables the PL/I-generated object program to address each element in the array. For arithmetic data, the array dope vector has the following form:

ORIGIN ADDRESS	
MULTIPLIER ₁	
⋮	
MULTIPLIER _n	
UPPER BOUND ₁	LOWER BOUND ₁
⋮	
UPPER BOUND _n	LOWER BOUND _n

The array dope vector contains $(2n+1)$ 32-bit words, where n is the number of dimensions in the array. ORIGIN ADDRESS is the address of an element whose subscript values are all zero, i.e., $A(0,0,\dots,0)$. MULTIPLIER_{*i*} is a full-word binary integer, which has the value $L_B \cdot M_i$ in the PL/I addressing scheme.[†] UPPER (LOWER) BOUND_{*i*} is the maximum (minimum) value permitted for the *i*th dimension expressed as a half-word integer. (In the present context of FORTRAN-defined arrays, we consider LOWER BOUND_{*i*} = 1 for all *i*.)

Example

For the array $A(2,3,4)$ declared as FIXED BINARY (31,0), the array dope vector, assuming $\text{Addr}[A(1,1,1)] = 18078_{16}$, is

[†]See Section III.

0	0	0	1	8	0	3	4
0	0	0	0	0	0	3	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	4
0	0	0	2	0	0	0	1
0	0	0	3	0	0	0	1
0	0	0	4	0	0	0	1

All numbers are given in hexadecimal.

Recall how the addressing schemes for PL/I and FORTRAN differ and that PL/I uses the multipliers directly to determine array-element addresses. Therefore, by considering the PL/I array as the transpose of the FORTRAN array and reversing the order of the multipliers and array bounds, we, in fact, overlay the FORTRAN array addressing scheme onto PL/I via the array dope vector and are thus enabled to access FORTRAN arrays directly in the PL/I environment. The new values of the M_i 's are the same as those for FORTRAN, namely, $M_1 = 1$ and

$$M_i = \prod_{j=1}^{i-1} D_j, \quad i = 2, \dots, n.$$

Example

If we consider the array of the previous example, the new array dope vector becomes

0	0	0	1	8	0	5	4
0	0	0	0	0	0	0	4
0	0	0	0	0	0	0	8
0	0	0	0	0	0	1	8
0	0	0	2	0	0	0	1
0	0	0	3	0	0	0	1
0	0	0	4	0	0	0	1

All numbers are given in hexadecimal.

As for arrays, a string dope vector must be created to pass string-data items to PL/I. The address of the string dope vector, rather than the

address of the string, is passed to the PL/I procedure. The string dope vector has the following form:

ADDRESS OF STRING	
MAXLN	CURRLN

The string dope vector contains two 32-bit words. ADDRESS OF STRING is the address of the first byte of the string. MAXLN (CURRLN) is a half-word binary integer whose value is the maximum (current) length of the string. For character strings, these lengths refer to the number of bytes the string occupies (one character per byte); for bit strings, the number of bits is specified.

Examples

Assume ISTRG(10) represents a 40-character string with $\text{Addr}(\text{ISTRG}) = 18054_{16}$. The dope vector has the form

0	0	0	1	8	0	5	4	
<hr/>				<hr/>				
0	0	2	8		0	0	2	8

All numbers are given in hexadecimal.

Suppose we want to define the FORTRAN real variable TEST as a bit string for purposes of manipulation in PL/I. We would generate a string dope vector of the following form, assuming $\text{Addr}(\text{TEST}) = 18040_{16}$:

0	0	0	1	8	0	4	0	
-----				-----				
0	0	2	0		0	0	2	0

All numbers are given in hexadecimal.

If we instead defined TEST in FORTRAN as LOGICAL*4, the same dope vector would be generated.

If we choose to define a string as an array, then a string-array dope vector must be created. The form of the string-array dope vector for fixed-length strings is:

ARRAY DOPE VECTOR	
MAXLN	CURRLN

The string-array dope vector consists of $(2n + 2)$ 32-bit words, where n is the number of dimensions in the array. The lengths implied in the string-array dope vector refer to the length of each element in units of bytes for character strings, but in bits for bit strings.

In the parameter list of a call to PL/I, we have a sequence of addresses. However, this list consists not only of actual addresses, as for scalar data types, but also addresses of dope vectors for string and array data. Proposed routines to create these dope vectors and assist in the explicit communication of data will be described in Section VII, together with a description of their use.

V. IMPLICIT PASSAGE OF DATA

Consider now the problem of providing an implicit means of communication from FORTRAN to PL/I. The implicit passage of data items is handled via the COMMON statement of FORTRAN and the STATIC EXTERNAL attribute of PL/I. The actual correspondence is established by declaring the label of the Common block in FORTRAN to have the attribute of STATIC EXTERNAL in PL/I.

When a variable is declared with the STATIC EXTERNAL attribute, a separate storage area or control section is generated. Each area consists of the data preceded by a dope vector describing the data. The COMMON statement must then have the form

COMMON/LABEL/DPVECT,DATA

where DPVECT is the appropriate vector describing the data type DATA. First we consider arithmetic data.

Example

```
FORTRAN  COMMON/A/DPVECT(3),DATA(50)
PL/I     DECLARE A(50) STATIC EXTERNAL FLOAT
          DECIMAL (6);
```

The associated area of storage can be outlined as

The total length of the storage area associated with the name A is 53 words.

Since PL/I requires an array dope vector to physically precede non-scalar data that have been declared EXTERNAL, it would seem reasonable that PL/I should also create this dope vector before its use. This is in fact the case. However, this dope vector is consistent with the PL/I addressing scheme. In the previous example, we incur no difficulty since we are dealing with a one-dimensional array. For multidimensional arrays, we are faced with our old problem. The array dope vectors discussed in Section IV can be used for data passed as parameters only, a fact we shall discuss in Section VII. Thus we must return instead to the techniques discussed in Section III.

We must also note that this one dope vector describes all the data in the entire external control section. This implies that this section must consist of one type of data and have a unique name. This leads to the obvious solution requiring each Common block to consist of only one data name.

Example

Using the transposed array technique,

```

FORTRAN  COMMON/DATA/ADV(5),A(5,10)
PL/I     DECLARE DATA(10,5) STATIC EXTERNAL FLOAT
          DECIMAL (6);

```

Using iSUB variables,

```

FORTRAN  COMMON/DATA/ADV(5),A(5,10)
PL/I     DECLARE DATA(10,5) STATIC EXTERNAL FLOAT
          DECIMAL (6),A(5,10) FLOAT DECIMAL (6) DEFINED
          DATA(2SUB,1SUB);

```

When using iSUB variables and the DEFINED attribute, we are defining one set of data onto another set of data. In other words, there is a sharing of storage by different sets of data. There must be only one base item, but we are not as limited in sharing the storage allocated to this base item. We can, in fact, define several different arrays of the same type onto unique sections of the base item as long as the defined arrays do not exceed the storage allocated to the base item. This correspondence defining leads to a more versatile approach to the problem of implicit communication of arithmetic data.

The following technique can be used in the PL/I procedure: The label of the Common block in the FORTRAN routine is declared as a

one-dimensional STATIC EXTERNAL array whose dimension is the total length (number of words) of the Common block, excluding the first three words* reserved for the one-dimensional array dope vector PL/I will create. Its other attributes, except dimensionality, are those of the data type in the Common block. The label is thus the name or identifier of the base item, and the variables of the Common block become the defined items.

The variables of the Common block are defined onto the base identifier through a subscript expression in the base subscript. The expression takes the form

$$a \pm a_1 * 1SUB \pm \dots \pm a_n * nSUB$$

where

$$a_1 = 1,$$

$$a_i = \prod_{j=1}^{i-1} D_j, \quad i = 2, \dots, n,$$

$$a = P - \sum_{i=1}^n a_i,$$

D_j is the value of the j th dimension of the n -dimensional array,

and

P is the position of the first element of the array in the Common block.

Example

```

FORTRAN  COMMON/LABEL/DV(3),IA(10),IB(2,5),IC(2,3,5)
PL/I     DECLARE LABEL (50) STATIC EXTERNAL FIXED
          BINARY (31), IA(10) FIXED BINARY (31) DEFINED
          LABEL, IB(2,5) FIXED BINARY (31) DEFINED
          LABEL (8+1SUB+2*2SUB),IC(2,3,5) FIXED BINARY (31)
          DEFINED LABEL (12+1SUB+2*2SUB+6*3SUB);

```

The Common block, LABEL, is 50 words long, excluding the dope vector. The elements of the block are four-byte integers and hence require the attribute of FIXED BINARY (31) in the PL/I procedure. IA occupies the first 10 positions of the Common block, IB occupies positions 11 through 20, and IC positions 21 through 50. Note that in the case of IA, the expression involving iSUB variables reduces to 1SUB. Hence IA is defined directly onto LABEL.

*Four words must be reserved if an array element occupies a double word. The fourth word ensures proper boundary alignment.

We now want to consider the problem of implicit communication of fixed-length string data. This communication can be achieved through a technique similar to the correspondence defining used for arithmetic array data. The DEFINED attribute is again used in conjunction with a base item declared as STATIC EXTERNAL. The base identifier or name is the same as the label of the associated Common block. In this case, however, a string dope vector is implied and only two words precede the string data in the Common block. This dope vector is again created by PL/I before its use. The base identifier becomes a string declared as CHARACTER or BIT depending on the type of strings in the Common block. The length of this base string is equal to the total number of characters or bits in the block, excluding the string dope vector.

Each defined string is overlayed or positioned onto the base string by specifying the position of the initial character or bit of the string relative to the beginning of the strings in the Common block. In correspondence defining, we had to specify the position of each defined array element through the iSUB variables. Now, in overlay defining, we supply the position of the initial character or bit of each defined string through the POSITION attribute. The decimal-integer-constant of the POSITION attribute is the position of the initial character or bit of the string in the Common block.

The first string of the Common block is defined directly onto the base identifier. The remaining strings are uniquely defined onto the base string through the POSITION attribute. The maximum length is 32,767 characters or bits for any one Common block. The strings must be fixed length, and all arrays must have the PACKED attribute.

Example

```

FORTRAN  COMMON/LSTRG/SDV(2),ISTRG(10),JSTRG(20)
PL/I      DECLARE LSTRG CHARACTER(120) STATIC
           EXTERNAL, ISTRG CHARACTER(40) DEFINED
           LSTRG, JSTRG(40) CHARACTER(2)
           PACKED DEFINED LSTRG POSITION(41);

```

The Common block LSTRG containing 30 words (neglecting the dope vector) becomes a PL/I character string of length 120 (four characters per word). ISTRG occupies the first 40 characters (10 words) of the string, and JSTRG, as an array of dimension 40 of two-character elements, occupies positions 41 through 120.

We defer to Section VII the description of establishing and using a FORTRAN-to-PL/I interface until after a discussion of the PL/I-to-FORTRAN communication has been presented.

VI. COMMUNICATION OF DATA TO FORTRAN

In this section, we turn our attention to the explicit and implicit communication of compatible data types from PL/I procedures to FORTRAN routines. In previous sections, we discussed various ways of generating source code to represent the same area of storage and assignment statements that refer to the same location. Most of these techniques are also valid in this interface system. The differences we encounter are related to the creation and use of string and array dope vectors.

For implicit communication, storage must still be reserved in an external control section for dope vectors as well as data; i.e., Common blocks must reserve storage for the dope vectors of the STATIC EXTERNAL data. Now, as before, PL/I will create these dope vectors before their use, and therefore they present no problem to us.

In explicit passage of data via the parameter list in a PL/I CALL statement, a point of conflict arises. Recall that an argument list is represented by a sequence of addresses. For PL/I, this list can include not only actual addresses of data, but also the addresses of the dope vectors. The former situation refers to scalar data, and the latter to string and array data types. FORTRAN, on the other hand, assumes this address list to be composed of the addresses of actual data only, not of data description blocks. Since both languages generate the actual addresses of scalar data, no problem exists for this data type.

For nonscalar data, we must somehow replace the address of dope vectors with the actual data address. First, consider the case of arithmetic array data. To pass an array in a PL/I environment that is under control of the PL/I system, the name of the array must appear in the argument list. The address of the associated array dope vector is then appropriately inserted in the list of addresses and properly interpreted for the called PL/I procedure. If the first element of the array is written in the argument list as a subscripted variable, the array element is considered as a scalar data item, and its address is inserted in the address list. But this latter address is what FORTRAN expects for arrays. Therefore, to pass an array of arithmetic data from PL/I to FORTRAN, we merely write the first element of the array and not the name of the array.

Example

```
DECLARE (A(10,5), B(50)) FLOAT DECIMAL (6);
```

Instead of

```
CALL FORTRAN(A,B);
```

use

```
CALL FORTRAN(A(1,1), B(1));
```

where we have

```
SUBROUTINE FORTRAN(A,B)
```

The former generates array dope vector addresses; the latter generates array addresses.

Since we are not communicating array data to FORTRAN through the dope vector, we cannot generate dope vectors consistent with FORTRAN addressing. For multidimensional arrays, this implies using the transposed array technique in the FORTRAN routine or the DEFINED attribute with iSUB variables in PL/I. For the latter, the base array should appear in the argument list.

For array or nonarray string data, a dope vector is always created. Thus, to explicitly pass string data to a FORTRAN routine, the address of the first element of the string must be obtained from the dope vector. This is most efficiently achieved at assembly language level. The address so obtained replaces the address of the string dope vector before transfer of control to the FORTRAN routine. A more complete discussion accompanies the description of the PL/I-to-FORTRAN interface routine in Section VIII.

VII. FORTRAN-TO-PL/I INTERFACE

This section describes the use of the assembly-language routine that creates dope vectors and an environment that allows transfer of control to a PL/I procedure. This assembly-language routine consists of three separately callable sections:

1. The linkage section, PL1CAL, which creates the PL/I environment and calls the PL/I procedure.
2. The string-dope-vector section, STRGDV, which creates string dope vectors.
3. The array-dope-vector section, ARAYDV, which creates array dope vectors.

The PL/I environment is created through a call to the program initialization routine, IHESAP. This Library routine provides the necessary initialization functions which provide for later dynamic management of the PL/I object program. The initialization functions are:

1. Allocation of storage for the task-oriented communications area called the pseudoregister vector, PRV. The PRV contains a number of pseudoregisters, which effectively operate as argument lists to various object time Library routines. For example, they give information about current program status.

2. Initial allocation of Library Workspace, LWS. LWS is dynamically allocated storage. This primary storage allocation is used by various Library routines for internal functions and linkage to other routines.

3. Specification to the supervisor of the address of the PL/I Library error handling subroutine, which is given control when a PL/I program interruption occurs.

Upon completion of these initialization functions, control is returned to the interface routine, which then establishes the linkage facilities to call the PL/I procedure.

The second section of the routine is used to create a string dope vector. It is invoked as follows:

```
CALL STRGDV (STRING,NCHAR,SDV)
```

where

STRING: address of (the first character (bit) in) the string,

NCHAR: number of characters (bits) in the string,

and

SDV: string-dope-vector address; consists of two 32-bit words.

The third section is used to create an array dope vector. It is invoked as follows:

```
CALL ARAYDV (ARRAY,ADV,N,LENGTH,D1,...,DN)
```

where

ARRAY: address of (first element in) the array,

ADV: address of the array dope vector; consists of $(2N+1)$ 32-bit words with N the number of array dimensions,

LENGTH: length, in bytes of each element in the array,

D1: maximum value of first dimension,

and

DN: maximum value of N th dimension.

To create a string-array dope vector, consider the string first as an array using ARAYDV with ADV of type INTEGER*4 to avoid an erroneous conversion problem. Then for the $(2n+2)$ nd 32-bit word of the dope vector, use the value

$$\text{NCHAR} * 2^{**16} + \text{NCHAR}$$

where NCHAR is as defined under STRGDV.

For arrays of bit strings, the multipliers must be multiplied by 8 to change units from bytes to bits as PL/I requires. In this case, ADV must again be INTEGER*4 to avoid erroneous conversion problems. Recall that the multipliers are the second through $(n+1)$ st words of the dope vector.

Example

```

FORTRAN
  DIMENSION TEST(10),ITDV(4),CHR(10),ICHRDV(4)
  CALL ARAYDV(TEST,ITDV,1,4,10)
  ITDV(4) = 32*2**16 + 32
  ITDV(2) = ITDV(2)*8
  CALL ARAYDV(CHR,ICHRDV,1,4,10)
  ICHRDV(4) = 4*2**16 + 4

```

coupled with

```

PL/I
  DECLARE TEST(10) BIT(32),
           CHR(10) CHARACTER(4);

```

Scalar parameters pass directly without a dope vector. The name of the PL/I procedure must be defined in an EXTERNAL statement. The call to the PL/I procedure is accomplished as follows:

```

FORTRAN
  EXTERNAL PLPROC
  :
  :
  CALL PLICAL (PLPROC,ARG1,...,ARGN)
  :
  :
  END

```

coupled with

```

PL/I
  PLPROC: PROCEDURE (ARG1,...,ARGN);
  :
  :
  END PLPROC;

```

Appendix A gives a programming example. Table I (in Appendix A) lists the assembly language routine for the FORTRAN-to-PL/I interface.

If character or bit string arrays are declared with the DEFINED attribute, they must also have the PACKED attribute. This will prevent an incorrect alignment when the DEFINED arrays are overlayed onto the base array. Arrays that are passed explicitly must be declared with asterisk notation for array bounds in the PL/I procedure. This ensures that PL/I-generated object code will use all the multipliers in the dope vector for computing the address of an array element.

If, instead, the array bounds are explicitly declared, the entire set of multipliers may not be used. If the "expected" value of a multiplier is 4 or 8, a sequence of consecutive additions of the appropriate subscript is used. If the *i*th multiplier is expected to be 4 (8), then the *i*th subscript is added to itself two (three) times rather than multiplying the *i*th subscript by the *i*th multiplier.

Example

```
DECLARE TWOD(2,2) FIXED BINARY (31);
```

PL/I DOPE VECTOR

ORIGIN ADDRESS							
0	0	0	0	0	0	0	8
0	0	0	0	0	0	0	4
0	0	0	2	0	0	0	1
0	0	0	2	0	0	0	1

For this case, the address of TWOD(I,J) is determined from code that adds I to itself three times. That is,

```
L    7,I
AR   7,7
AR   7,7
AR   7,7 → 8*I in register 7.
```

A similar sequence of addition is performed for J.

This "shortcut" of additions is in keeping with the PL/I addressing scheme, but not with our attempt to incorporate FORTRAN addressing into

PL/I, because the "expected" value of a multiplier is not the value we would supply through out (FORTRAN environment) created dope vectors. In the previous example, we would desire 4 for the first multiplier and 8 for the second.

During the linkedit step of the job, the system library, SYSLIB, must contain both the FORTRAN and PL/I libraries to properly resolve all external references.

To avoid improper sequencing of printed output and to ensure that no output is lost if the PL/I procedure terminates abnormally, it is necessary to close the FORTRAN output buffer before calling the PL/I procedure. This can be accomplished by the REWIND n statement, where n refers to the FORTRAN output buffer. The PL/I Library routines will close the PL/I output buffer when the procedure terminates normally. It is also recommended that only one object routine handle standard input, i.e., card reader. The problem of external files opened in both systems, simultaneously or one at a time, is deferred to a later report.

VIII. PL/I-TO-FORTRAN INTERFACE

In this section, we discuss the use of the assembly-language routine that creates the environment that allows transfer of control to a FORTRAN subroutine and obtains string addresses from the PL/I dope vectors.

The FORTRAN environment is created through an entry point in the Library routine IHCF COMH. This call will specify an error exit for arithmetic-type program interruptions which may occur in the FORTRAN subroutine. Upon return from the Library routine, the interface routine will generate addresses for any string data types and then pass control to the FORTRAN subroutine.

The interface routine is invoked as follows:

```
CALL FORTRAN (FTNRTN,FLAG,ARG1,...,ARGN);
```

where

FTNRTN: address of the FORTRAN routine; FTNRTN must be declared with attributes EXTERNAL ENTRY prior to the CALL statement,

FLAG: FIXED BINARY word that describes the structure of the remaining argument list;

if

FLAG < 0	no further arguments,
FLAG = 0	no string data in the argument list,
FLAG = M > 0	the first M parameters are string data and thus are represented by string dope vectors.

Recall that, for arrays, use the first element of the array, written in subscript notation, as the argument. If the array is a string, again use the first element of the array. PL/I will generate a temporary string dope vector, which then allows us to obtain the address of the string from this temporary dope vector rather than from a string-array dope vector. Also, use the string name as an argument for nonarray strings.* The call to the FORTRAN subroutine is accomplished as follows:

```

PL/I
    DECLARE FTNRTN EXTERNAL ENTRY;
    FLAG FIXED BINARY (31);
    CALL FORTRAN (FTNRTN,FLAG,ARG1,...,ARGN);
    :
    :
FORTRAN
    SUBROUTINE FTNRTN (ARG1,...,ARGN)
    :
    :
    END

```

Appendix B presents a programming example. Table II (in Appendix B) lists the assembly language routine for the PL/I-to-FORTRAN interface. As in Section VII above, during linkedit, the PL/I and FORTRAN libraries must be combined to form SYSLIB to ensure proper interpretation of external references.

For synchronized printed output, PL/I output buffers must be closed before calling FORTRAN, and FORTRAN output buffers must be closed before returning to PL/I. The former can be accomplished by

```
CLOSE FILE (SYSPRINT);
```

and the latter by

```
REWIND n
```

where n refers to the FORTRAN output unit.

Further details are presented in the latter part of Section VII.

*It may be necessary to declare string data, appearing as an argument, with the EXTERNAL attribute to ensure proper boundary alignment.

APPENDIX A

FORTRAN-to-PL/I Programming Example

The following FORTRAN program creates string and array dope vectors and calls the PL/I procedure SEARCH through PLICAL. The PL/I procedure will search the array IPL and convert the value of IPL(I,J) to a bit string, if $IPL(I,J) \geq LIMIT$, and store it at the corresponding position in DTEST; or convert IPL(I,J) to a character string, if $IPL(I,J) < LIMIT$, and store it at the corresponding position in JCHR. A value of 0 or 1 is then placed into IPL(I,J) as a flag indicating the type of conversion performed (0 for bit, 1 for character).

Note that in the PL/I procedure, character and bit string arrays with the DEFINED attribute have the PACKED attribute also. In addition, arrays passed explicitly are declared with asterisk notation for array bounds. The REWIND 6 statement in the FORTRAN program ensures that the printed output will be synchronized.

FORTRAN

```

COMMON/CHR/CHRDV(2),ICHR(5),JCHR(10)
COMMON/INTGR/INTDV(3),IA(2),IB(2,5)
DIMENSION IPL(2,5),IPLDV(5),TEST(10),TESTDV(2)
EXTERNAL SEARCH
CALL ARAYDV(IPL,IPLDV,2,4,2,5)
CALL STRGDV(TEST,320,TESTDV)
READ 10,NLIM,((IPL(I,J), J=1,5), I=1,2)
PRINT 20,NLIM
REWIND 6
CALL PLICAL(SEARCH,NLIM,IPLDV,TESTDV)
< = 0
DO I J = 1,5
DO I I = 1,2
K = K + 1
IF(IPL(I,J) .EQ. 0) GO TO 2
PRINT 30,K,IPL(I,J),JCHR(K)
GO TO 1
2 PRINT 40,K,IPL(I,J),TEST(K)
1 CONTINUE
STOP
C
10 FORMAT (16/(5I10))
20 FORMAT('I DELIMITER',16)
30 FORMAT ('0',15,110,A10)
40 FORMAT ('0',15,110,10X,Z10)
C
END

```

PL/I

```

SEARCH: PROC (LIMIT,IPL,TEST);
      DCL (LIMIT,IPL(*,*),P)FIXED BIN(31),
          TEST BIT(320),DTEST(10) BIT(32) PACKED
          DEFINED TEST;
      DCL CHR CHAR(60) STATIC EXT, ICHR(5) CHAR(4)
          PACKED DEFINED CHR,JCHR(10) CHAR(4)
          PACKED DEFINED CHR POSITION (21);
      DCL INTGR(15) FIXED BIN(31) STATIC EXT, IA(2)
          FIXED BIN(31) DEFINED INTGR, IB(2,5)
          FIXED BIN(31) DEFINED INTGR (ISUB+2*2SUB);
      DCL TEMPORARY CHAR(14), NUMBER CHAR(4)
          DEFINED TEMPORARY POSITION (11);
      PUT SKIP EDIT(' BEGIN SEARCH') (A);
      IB = IPL;
      IA = 0;
      LJ: DO J = 1 TO 5;
      LI: DO I = 1 TO 2;
          P = 1 - 2 + 2*J;
          IF IPL(I,J) < LIMIT THEN GO TO L2;
          DTEST(P) = UNSPEC(IPL(I,J));
          IPL(I,J) = 0;
          IA(2) = IA(2) + 1;
          GO TO LI;
      L2: TEMPORARY = IPL(I,J);
          JCHR(P) = NUMBER;
          IPL(I,J) = 1;
          IA(1) = IA(1) + 1;
      LI: END LI;
      END LJ;
      PUT EDIT (' END SEARCH') (SKIP(3),A);
      RETURN;
      END SEARCH;

```

TABLE 1. FORTRAN-to-PL/I Interface Routine

```

PLICAL  START
*
* CALLING SEQUENCE CALL PLICAL(PLPROC,ARG1,ARG2,...)
*          WHERE PLPROC: PROCEDURE(ARG1,ARG2,...);
          ENTRY IHEMAIN,STRGDV,ARRAYDV
          EXTRN IHEERRA
          STM 14,12,12(13)          SAVE REGISTERS
          BALR 10,0
          USING *,10          ESTABLISH BASE REGISTER
BASE    ST 13,SAVE+4
          LA 13,SAVE
          L 15,ADDR
          BR 15          LINK PL/I

```

TABLE I (Contd.)

RESUME	LA	10,BASE	RESET BASE REGISTER
	L	13,SAVE+4	
	L	1,24(13)	
	L	15,0(1)	LOAD ADDRESS OR POINTER
	CLI	0(15),X'47'	TEST FOR ADDRESS OR INSTRUCTION
	BC	8,*+8	
	L	15,0(15)	
	LA	1,4(1)	SET PL/I PARAMETER LIST POINTER
	LA	13,SAVE	
	BALR	14,15	BRANCH TO PL/I PROCEDURE
	L	13,SAVE+4	
	LM	14,12,12(13)	RESTORE REGISTERS
	BR	14	RETURN TO FORTRAN
*			
* CALLING SEQUENCE CALL STRGDV(String,NCHAR,SDV)			
*			
	CNOP	0,4	
STRGDV	STM	2,5,28(13)	SAVE REGISTERS
	LM	2,4,0(1)	LOAD PARAMETER LIST
	ST	2,0(4)	STORE STRING ADDRESS IN D.V.
	L	5,0(3)	
	STH	5,4(4)	STORE STRING LENGTH IN D.V.
	STH	5,6(4)	
	LM	2,5,28(13)	RESTORE REGISTERS
	BR	14	RETURN TO FORTRAN
*			
* CALLING SEQUENCE CALL ARAYDV(Array,ADV,N,LENGTH,D1,...,DN)			
*			
	CNOP	0,4	
ARAYDV	STM	2,10,28(13)	
	L	10,POINTER-ARAYDV(15)	
* COMPUTE AND STORE MULTIPLIERS			
	L	3,8(1)	
	L	3,0(3)	PICK UP N FOR LOOP
	ST	3,N	
	L	2,4(1)	PICK UP DOPE VECTOR ADDRESS
	LA	4,8(1)	SET FACTOR POINTER
	SR	8,8	ZERO SUM
	LH	7,ONE	PUT 1 IN PRODUCT
LOOP1	AH	4,FOUR	
	AH	2,FOUR	
	L	6,0(4)	PICK UP FACTOR ADDRESS
	M	6,0(6)	DOPE VECTOR MULTIPLIER IN 7
	AR	8,7	COMPUTE ADDRESS DECREMENT
	ST	7,0(2)	STORE MULTIPLIER IN DOPE VECTOR
	BCT	3,LOOP1	
* END LOOP1			
* COMPUTE AND STORE ORIGIN ADDRESS			
	LM	4,5,0(1)	
	SR	4,8	COMPUTE ORIGIN ADDRESS
	ST	4,0(5)	STORE ADDRESS IN DOPE VECTOR

TABLE 1 (Contd.)

* STORE ARRAY DIMENSIONS			PICK UP LOOP MAXIMUM
	L	3,N	SET DIMENSION POINTER
	LA	4,12(1)	
LOOP2	AH	4,FOUR	
	AH	2,FOUR	
	L	5,0(4)	PICK UP UPPER BOUND ADDRESS
	MVC	0(2,2),2(5)	MOVE UPPER BOUND INTO DOPE VECT
	MVI	3(2),X'01'	MOVE LOWER BOUND OF 1 INTO D.V.
	BCT	3,LOOP2	
* END LOOP2			
	LM	2,10,28(13)	RESTORE REGISTERS
	BR	14	RETURN TO FORTRAN
* STORAGE SECTION			
SAVE	DS	18F	
N	DS	F	
ADDR	DC	V(IHESAPB)	
IHEMAIN	DC	A(RESUME)	
POINTER	DC	A(BASE)	
FOUR	DC	H'4'	
ONE	DC	H'1'	
*			
END			

APPENDIX B

PL/I-to-FORTRAN Programming Example

The following PL/I procedure inputs a matrix A and calls, through FORTRAN, the FORTRAN routine MATINV, which calculates the inverse of this matrix. MATINV is a standard matrix inverse routine and is not listed here, except for the following:

```
SUBROUTINE MATINV (A,N,B,M,DETERM,NMAX)
  DIMENSION A(NMAX,N), B(NMAX,M)
  DOUBLE PRECISION A,B,DETERM
```

Note the defining of A, B, and C onto ABASE, BBASE, and CBASE, respectively, with iSUB variables. The base arrays are used in the parameter list as required by PL/I. The CLOSE statement is used to ensure that all output to this point will appear if an error occurs in MATINV.

PL/I

```
INVRT: PROC OPTIONS(MAIN);
  DCL (ABASE(20,20),A(20,20) DEFINED ABASE(2SUB,1SUB),
       BBASE(20,20),B(20,20) DEFINED BBASE(2SUB,1SUB),
       CBASE(20,20),C(20,20) DEFINED CBASE(2SUB,1SUB),
       S,DETERM) FLOAT DECIMAL(16);
  DCL (M,N,NMAX) FIXED BIN(31);
  DCL FLAG FIXED BIN(31),MATINV EXT ENTRY;
  DCL P CHAR(1);
  GET EDIT (N,P) (F(6),X(73),A(1));
  GET EDIT ( ((A(1,J) DO J=1 TO N) DO I=1 TO N) ) (8 E(10,5));
  PUT PAGE EDIT (' INPUT MATRIX') (A);
  PUT SKIP EDIT ( ((A(1,J) DO J=1 TO N) DO I=1 TO N) ) (R(F1));
SI: DO I = 1 TO N;
  DO J = 1 TO N;
    B(1,J) = A(1,J);
  END SI;
  FLAG = 0;
  M = 0;
  NMAX = 20;
  CLOSE FILE (SYSPRINT);
  CALL FORTRAN(MATINV,FLAG,ABASE(1,1),N,BBASE(1,1),M,DETERM,
              NMAX);
  PUT PAGE EDIT (' INVERSE MATRIX') (A);
  PUT SKIP EDIT ( ((A(1,J) DO J=1 TO N) DO I=1 TO N) ) (R(F1));
LI: DO I = 1 TO N;
  DO J = 1 TO N;
    S = 0;
  L2: DO K = 1 TO N;
    S = S + A(1,K)*B(K,J);
  END L2;
  C(1,J) = S;
  END LI;
  PUT PAGE EDIT (' IDENTITY MATRIX COMPUTED') (A);
  PUT SKIP EDIT ( ((C(1,J) DO J=1 TO N) DO I=1 TO N) ) (R(F2));
F1: FORMAT (SKIP,8 E(15,5));
F2: FORMAT (SKIP,8 F(15,5));
END INVRT;
```

TABLE 11. PL/I-to-FORTRAN Interface Routine

```

FORTRAN START
*
* CALLING SEQUENCE  CALL FORTRAN(FTNRTN,FLAG,A,B,C,...)
*                   FLAG < 0  IMPLIES NO PARAMETERS
*                   FLAG = 0  IMPLIES NO STRING DATA IN PARM. LIST
*                   FLAG = N  IMPLIES FIRST N PARAMETERS ARE STRINGS
*
      STM 14,12,12(13)          SAVE REGISTERS
      BALR 12,0
      USING *,12
BASE  ST 13,SAVE+4
      LA 13,SAVE
      L 15,FADDR
      BAL 14,64(15)
      LA 12,BASE
      L 13,SAVE+4
      L 1,24(13)
      L 2,0(1)
      L 15,0(2)                LOAD ADDRESS OF FORTRAN ROUTINE
      LA 13,SAVE
* TEST FLAG FOR PARAMETERS
      L 3,4(1)
      L 3,0(3)
      LTR 3,3
      BC 4,CALL                FLAG < 0 NO PARAMETERS
      LA 1,8(1)                SET PARAMETER LIST POINTER
      BC 8,CALL                FLAG = 0 NO STRING DATA
* OBTAIN STRING ADDRESS FROM DOPE VECTOR
      SR 6,6
LOOP  L 4,0(6,1)
      L 2,0(4)                LOAD SDV
      ST 2,0(6,1)            STORE STRING ADDRESS
      AH 6,FOUR
      BCT 3,LOOP
*
CALL  BALR 14,15                BRANCH TO FORTRAN ROUTINE
      L 13,SAVE+4
      LM 14,12,12(13)          RESTORE REGISTERS
      BR 14                    RETURN TO PL/I
*
* STORAGE SECTION
SAVE  DS 18F
FADDR DC V(1BCOM#)
FOUR  DC H'4'
*
      END

```

BIBLIOGRAPHY

IBM System/360 PL/I Reference Manual, Form No. C28-8201.

IBM System/360 Operating System PL/I(F) Programmer's Guide, Form No. C28-6594.

IBM System/360 Operating System PL/I Subroutine Library Program Logic Manual, Form No. Y28-6801.

IBM System/360 Operating System FORTRAN IV(H) Programmer's Guide, Form No. C28-6602.