GF20-0020-0

IBM Data Processing Techniques

Techniques for Processing Relocatable Lists in PL/I

This manual illustrates usage of PL/I list-processing facilities for processing relocatable data lists, pointer lists, and lists of lists. Relocatable lists are lists organized within an area of storage in a way that permits the area to be transmitted to and from an external storage medium without disturbing the linkage of list components in the area. Such organization also permits moving lists about in main storage.

The information in this manual concerning data lists assumes knowledge of *Introduction to the List Processing Facilities of PL/I* (GF20-0015) and *Techniques for Processing Data Lists in PL/I* (GF20-0018). Some of the information in this manual assumes knowledge of *Techniques for Processing Pointer Lists and Lists of Lists in PL/I* (GF20-0019). The audience for this manual is assumed to be the experienced programmer.

Illustrative programs were processed by the PL/I (F) Compiler (Version 5.1) under control of the IBM System/360 Operating System (Release 19).

Table of Contents

First Edition (August 1971)

Copies of this and other IBM publications can be obtained through IBM branch offices.

A form has been provided at the back of this publication for readers' comments. If this form has been removed, address comments to: IBM Corporation, Technical Publications Department, 1133 Westchester Avenue, White Plains, N. Y. 10604.

Preface

List processing in PL/I concerns programs which manipulate the storage addresses and the contents of based variables that are linked by contained locator variables. Techniques for using the list-processing facilities of PL/I to manipulate list components in main storage have been presented in *Techniques for Processing Data Lists in PL/I* (GF20-0018) and *Techniques for Processing Pointer Lists and Lists of Lists in PL/I* (GF20-0019).

This manual discusses methods used to convert list components linked within an area by absolute storage addresses to list components linked within an area by relative storage addresses. These latter addresses are relative to the beginning of the containing area. Such lists are called relocatable lists because they can be transferred about in main storage or transmitted to and from external storage for subsequent processing.

As indicated in the preceding manuals, the primary advantages of list-processing techniques are efficient main storage utilization and the ability to preserve the logical organization of complex data entities that do not lend themselves to convenient representation by conventional PL/I arrays and structures. Data of this type occurs in many nonnumeric applications, such as information storage and retrieval, system simulation, engineering design, computer-software production, text editing, and artificial intelligence. List processing preserves the natural structure of the data involved in such applications and reduces the complexity of related programs. The convenience of organizing data in list form is enhanced by the ability to transmit relocatable lists to and from external storage as needed by the application.

This manual illustrates the techniques involving relocatable lists with subroutine and function procedures that concentrate on specific aspects of creating and moving relocatable lists. Clarity of presentation has been emphasized rather than efficient programming techniques. No attempt has been made to produce "production" code. Suitable inline code may be preferred for many applications.

Because processing lists in PL/I is an advanced programming topic, this manual assumes that the reader is an experienced programmer with a knowledge of PL/I equivalent at least to that presented in *A PL/I Primer* (SC28-6808). Familiarity is assumed with array and structure organization and with methods for creating and invoking subroutines and functions. In addition, knowledge of the information contained in the following publications is assumed:

Introduction to the List Processing Facilities of PL/I (GF20-0015)
Techniques for Processing Data Lists in PL/I (GF20-0018)
Techniques for Processing Pointer Lists and Lists of Lists (GF20-0019)

Information on the F-level list-processing facilities appears in *IBM System/360: PL/I(F) Language Reference Manual* (GC28-8201) and *IBM System/360 Operating System: PL/I(F) Programmer's Guide* (GC28-6594).

Introduction

The techniques developed for creating and processing lists in *Techniques for Processing Data Lists in PL/I* and *Techniques for Processing Pointer Lists and Lists of Lists* do not consider moving a list about in storage or a list to and from a file. The ability to store a list in a file is important because it allows a list to be processed in stages by successive runs of either the same program or any other designed to retrieve the list.

One approach to the external transmission of a list is to disassemble the components of the list and to write their associated data values successively into a file. Then, when the list is to be processed further, the data values can be retrieved from the file, and the list can be reconstructed component by component. Transmission of a list in this manner may have merit with simple linear data lists but becomes complicated and inefficient when applied to nonlinear lists. A more desirable approach is to keep the list intact within its containing area and to write the area into the file as a unit; however, this method also presents a problem—pointer values within an area become invalid when the area is stored at a new location. Since there is no way of assuring that an area will occupy the same storage each time it is read from a file, the organization of a list contained in a retrieved area can generally be assumed to be destroyed because its linking pointers will be invalid at the new location.

PL/I overcomes this difficulty in the transmission of a list by providing special variables called offset variables, which are used in place of pointer variables and which remain valid when a list is moved to a new location. An offset variable is a storage address that is relative to the beginning of an area. This manual shows how offset variables can be used to organize data lists, pointer lists, and lists of lists into relocatable form and how such lists can be written into and read from a file.

Chapter 1. Organizing Relocatable Lists

The organization of relocatable lists depends mainly upon two PL/I facilities: offset variables, and the assignment statement applied to area variables. A detailed presentation of these facilities appears in the companion manual. *Introduction to the List Processing Facilities of PL/I* (GF20-0015). However, for the purposes of this chapter, the following discussions present a review of offset variables and area assignment and show how these facilities may be used to organize relocatable lists. The discussions also illustrate how relocatable lists can be moved to new storage locations and how they can be transmitted to and from files.

AREA ASSIGNMENT

An area variable is declared with the following attribute:

AREA [(size-expression)]

The size expression determines the number of bytes of storage reserved for the area. However, the size expression is optional; when it is not used, an implementation-defined size is assumed by the PL/I compiler. An asterisk (*) may be used in place of the size expression when the area variable appears as a parameter in either a subroutine or a function. The asterisk causes the area parameter to assume the size of the associated area argument.

An assignment statement can contain an area variable to the left of the equal sign provided the expression on the right is restricted to either another area variable or a function reference that possesses an area value. Execution of an assignment statement that contains area variables effectively frees all allocations in the receiving area and then assigns the contents of the source area to the receiving area. Free-storage gaps are retained during the assignment, so that allocations within the assigned area maintain their locations relative to each other.

Illustrations of area assignments appear in Figure 1.1. The shaded portions of the areas represent free storage that is available for further allocations of based variables within the areas. When the source area is smaller than the receiving area, the assigned area is, in effect, extended with free storage. Similarly, when the source area is larger than the receiving area, truncation of free storage occurs at the end of the assigned area. However, if the truncation involves allocated storage and not just free storage, the AREA ON-condition occurs, and the contents of the receiving area become undefined. If no ON-unit appears in an ON statement for the AREA condition, the operating system issues a comment and raises the ERROR condition. When an ON-unit is specified and normal return occurs from the ON-unit, program control returns to the point of interruption.

When an area variable is allocated, it is automatically given the empty state, which indicates that no storage has been allocated for based variables within the area. An area that is not empty can be made empty by assigning to it the value of an empty area or the value of the built-in function EMPTY. The effect of such an assignment is to free all allocations of based variables within the receiving area. Note that the area itself does not become free but retains its storage in reserve for further allocations of based variables.

A reference to the built-in function EMPTY uses no arguments and consists solely of the keyword EMPTY. A reference to EMPTY cannot appear in an operational expression; the value of EMPTY is used only to free storage allocated in a specified area.

Area assignment can be used to transmit any type of data from one area to another, but, as mentioned earlier, pointer values contained in the assigned area will generally be transmitted incorrectly. As a result, area assignment cannot be used to move a list linked by pointer variables; the addresses of the list components would not be known in the receiving area (see Figure 1.2). Note that assigning the head pointer in the source list to the intended head of the list in the receiving area would also be incorrect since the second head would specify the address of the first list component in the source area and not the address of the first list component in the receiving area. This difficulty in pointer transmission is overcome by replacing pointer variables with relocatable variables called offset variables.

OFFSET VARIABLES

An offset variable is a storage address that is relative to the beginning of an area. An offset variable must be declared explicitly with the OFFSET attribute, which has the following form:

OFFSET(area-variable)

The area variable in parentheses must also be declared explicitly and must be a based variable that is unsubscripted and has an implied or explicit level number of one.

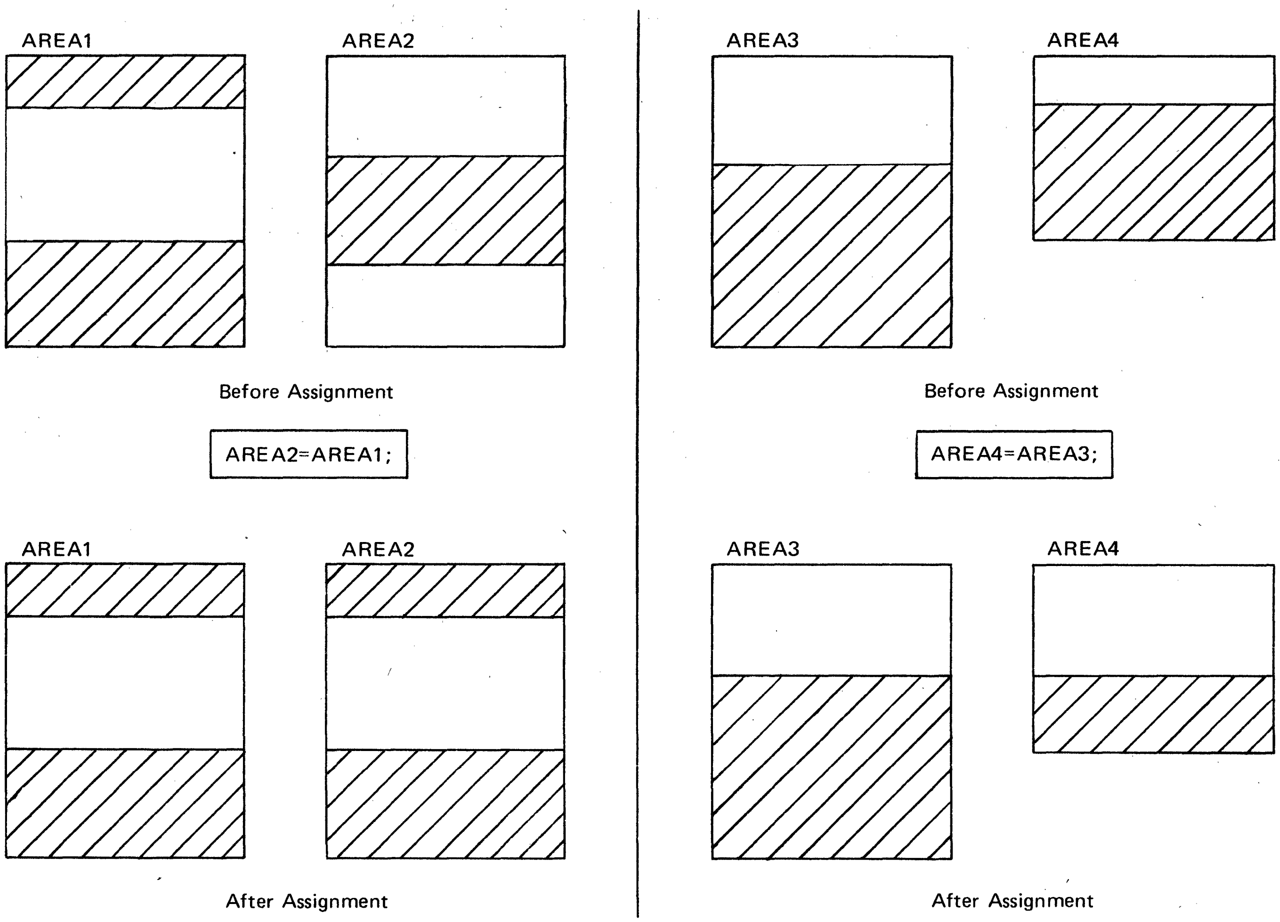
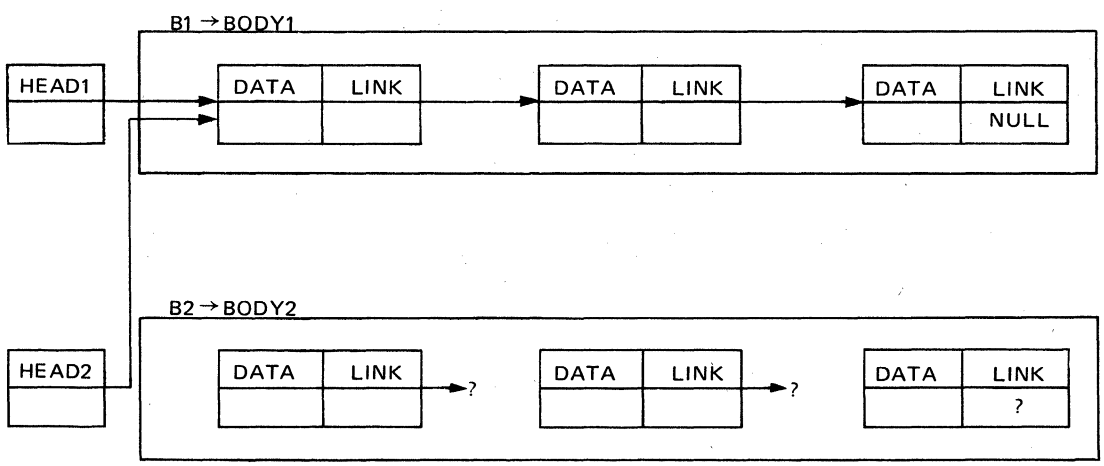


Figure 1.1. How areas are assigned



B2 → BODY2 = B1 → BODY1;
HEAD2 = HEAD1;

Figure 1.2. Incorrect use of area assignment to move a list

Examples:

```
DECLARE
    AREA1 AREA(2000) BASED(P1),
    O OFFSET(AREA1),
    (M,N) OFFSET(AREA1) EXTERNAL STATIC,
    SWITCH CONTROLLED OFFSET(AREA1),
    T(5) OFFSET(AREA1) INTERNAL,
    1 A, 2 X CHARACTER(15), 2 Y OFFSET(AREA1);
```

As shown in these examples, PL/I allows offset variables to be individual element variables or elements of arrays and structures. An offset variable can have any storage class and scope, and the usual default rules for these types of attributes also hold for an offset variable.

The value of an offset variable is always treated as a relative address and never as an absolute address. The offset value is relative to the beginning of the area specified in the associated OFFSET attribute. Assume, for example, that O1 is an offset variable and that AREA1 is an area variable declared as follows:

```
DECLARE
    AREA1 AREA BASED(A1),
    O1 (OFFSET(AREA1);
```

Assume further that the value of O1 is 75. The O1 specifies the 75th storage position (in bytes) from the beginning of AREA1.

Values are assigned to offset variables through the assignment statement. An offset variable can receive the value of another offset variable or the value of a pointer variable. When the value of a pointer variable is assigned to an offset variable, the assigned pointer value is automatically adjusted so that it becomes relative to the beginning of the area associated with the receiving offset variable. The address arithmetic performed automatically by the PL/I program to obtain the offset value is equivalent in effect to the following calculation:

Offset value = (Pointer value) - (Absolute address of area)

Similar but reverse address arithmetic is performed automatically when an offset value is assigned to a pointer variable. The offset value is added to the absolute address of the area specified in the associated OFFSET attribute:

Pointer value = (Offset value) + (Absolute address of area)

Note that these calculations are performed automatically by the PL/I program; the programmer cannot apply explicit arithmetic operations to offset variables in the source program.

When the value of an offset variable is assigned to another offset variable, no address arithmetic is performed; the assignment is direct, so that both offset variables have the same value.

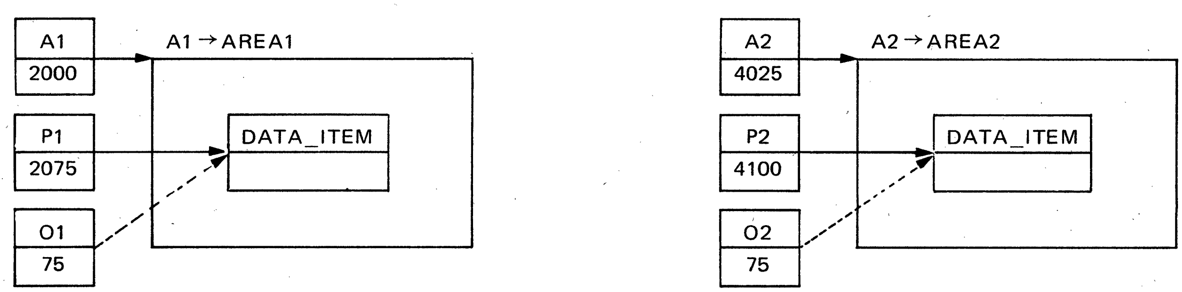
The following example shows how values are assigned to offset variables and how the absolute address of a data item is obtained in an assigned area:

```
DECLARE
    AREA1 AREA(500) BASED(A1),
    AREA2 AREA(500) BASED(A2),
    O1 OFFSET(AREA1),
    O2 OFFSET(AREA2),
    DATA_ITEM BASED (P1) CHARACTER(80);
      .          .
      .          .
      .          .
    ALLOCATE AREA1 SET(A1);
    ALLOCATE AREA2 SET(A2);
    ALLOCATE DATA_ITEM IN(A1->AREA1) SET(P1);
      .          .
      .          .
      .          .
    A2->AREA2 = A1->AREA1;
    O1 = P1;
    O2 = O1;
    P2 = O2;
      .          .
      .          .
      .          .
```

AREA1 and AREA2 are area variables, each of which reserves 500 bytes of based storage; O1 is an offset variable associated with AREA1, and O2 is an offset variable associated with AREA2. The based variable DATA_ITEM is a character string that requires 80 storage bytes.

When storage is allocated for AREA1, the absolute address of the allocation is assigned to pointer A1. Similarly, pointer A2 receives the absolute address of the storage allocated for AREA2. For example, Figure 1.3 assumes that AREA1 is allocated at location 2000 and that AREA2 is allocated at location 4025. The allocation of DATA_ITEM in A1->AREA1 is also assumed to occur at location 2075, which is assigned to pointer P1.

After A1->AREA1 is assigned to A2->AREA2, both areas contain equivalent storage configurations, as shown in Figure 1.3. Assignment of pointer P1 to offset O1 produces the relative address (75 = 2075 - 2000) of DATA_ITEM within A1->AREA1. This relative address remains unchanged when assigned to offset O2. Assignment of O2, in turn, to P2 produces the absolute address (4100 = 75 + 4025) of DATA_ITEM in A2->AREA2. Reference to DATA_ITEM in A2->AREA2 is then possible with the expression P2->DATA_ITEM. (Note that an offset variable cannot be used to qualify a based variable.) The broken lines in Figure 1.3 distinguish offset variables from pointer variables.




```
A2→AREA2 = A1→AREA1;     /* ASSIGN A1→AREA1 TO A2→AREA2. */
O1 = P1;                 /* SET O1 TO RELATIVE ADDRESS OF DATA_ITEM IN A1→AREA1. */
O2 = O1;                 /* SET O2 EQUAL TO O1. */
P2 = O2;                 /* SET P2 TO ABSOLUTE ADDRESS OF DATA_ITEM IN A2→AREA2. */
```

Figure 1.3. Obtaining the absolute address of a data item in an assigned area

Offset values and pointer values form a special type of program control data called the locator type. Locator data cannot be converted to any other type, nor can any other type of data be converted to locator type. Offset variables can receive offset and pointer values only; the same restriction applies to pointer variables.

A null offset value may be assigned to an offset variable through the built-in function NULLO, which uses no arguments and consists solely of the keyword NULLO. A reference to this function produces a null offset address, which does not specify any relative storage location.

Although pointer values may be assigned to offset variables and offset values may be assigned, in turn, to pointer variables, a null offset value cannot be assigned to a pointer variable, nor can a null pointer value be assigned to an offset variable. These restrictions apply not only to explicit references to NULL and NULLO but also to assigned variables that currently have null pointer or null offset values.

Assume, for example, that P is a pointer variable and O is an offset variable. Then P can be assigned to O provided that P does not have a null value. When the value of P may be null, an IF statement can be used to ensure proper assignment:

```
IF P = NULL
   THEN O = NULLO;
   ELSE O = P;
```

A similar statement governs the correct assignment of O to P:

```
IF O = NULLO
   THEN P = NULL;
   ELSE P = O;
```

As with pointer variables, the comparison operators equal (=) and not equal (¬=) are the only operators that can use offset variables as operands.

Offset variables, as well as pointer variables, can serve as arguments and parameters. An offset argument associated with an offset parameter or a pointer argument associated with a pointer parameter requires no conversion and therefore produces no dummy argument. But an offset argument associated with a pointer parameter or a pointer argument associated with an offset parameter does require conversion and will produce a dummy argument. Also, when an offset argument is associated with an offset parameter, both must be offset with respect to the same area for the argument-parameter association to be meaningful.

Although the area variable specified in the OFFSET attribute for an offset variable must be a based area, it is possible to associate an offset variable with an area that is not based. Consider the following statements:

```
DECLARE
   AREA1 AREA(2000),
   DUMMY_AREA AREA(2000) BASED
   (DUMMY_POINTER),
   O OFFSET(DUMMY_AREA);
   .        .
   .        .
   .        .
   DUMMY_POINTER = ADDR(AREA1);
```

AREA1 and DUMMY_AREA are area variables. AREA1 reserves automatic storage, and DUMMY_AREA reserves based storage. The OFFSET attribute for variable O uses DUMMY_AREA and thus satisfies the requirement that the area specified in an OFFSET attribute must be based.

When the address of AREA1 is assigned to DUMMY_POINTER, DUMMY_AREA becomes equivalent to AREA1. Subsequent references to offset variable O are then effectively associated with AREA1.

The size declared for DUMMY_AREA in the previous example does not have to be the same as the size of AREA1 and can even be zero. The only purpose of DUMMY_AREA is to provide a level-one based area variable for the OFFSET attribute of variable O, so that variable O can be made relative to the starting address of AREA1. The size of DUMMY_AREA is not important, because it does not affect the starting address assigned to DUMMY_AREA through DUMMY_POINTER.

RELOCATABLE ORGANIZATIONS FOR DATA LISTS, POINTER LISTS, AND LISTS OF LISTS

A relocatable list has the same organization as an absolute list except that offset variables rather than pointer variables are used to link the components of the relocatable list. This section presents illustrations of relocatable organizations for data lists, pointer lists, and lists of lists and shows how such lists may be moved from one location to another within internal storage. However, procedures for actually constructing relocatable lists are deferred to Chapter 2. The emphasis in this section is upon the use of offset variables as component links in relocatable lists.

Figure 1.4 illustrates the organization of a relocatable data list and shows the PL/I statements that can be used to move the list to a new location. Broken lines, instead of solid lines, indicate the offset links (OL's) and offset head of each list in the figure. Actual values (in decimal) are assumed for the offset variables to show that they remain unchanged when the list is moved.

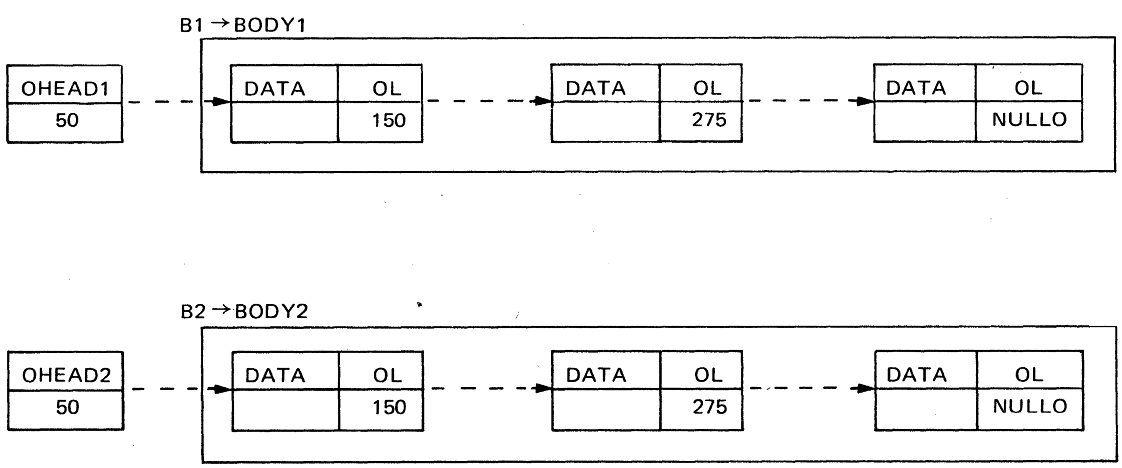
Two assignment statements perform the relocation of the data list in Figure 1.4:

```
B2->BODY2 = B1->BODY1;
OHEAD2 = OHEAD1;
```

The based area B1->BODY1 contains the body of the list, which is assigned to the based area B2->BODY2. The offset variable OHEAD2 receives the offset head OHEAD1 of the list. The effect of these assignments is to produce a separate copy of the original data list.

Since an offset variable cannot be used to qualify a based variable, it is not possible to refer to a component of a relocatable list unless the absolute address of the component is used. As an example, the following statements show how to obtain the absolute address of the last component in each of the relocatable data lists illustrated in Figure 1.4:

```
DECLARE
   BODY1  AREA(500)  BASED(B1),
   BODY2  AREA(500)  BASED(B2),
   1 COMPONENT1 BASED(P1),
      2 DATA CHARACTER(1),
      2 OL     OFFSET(BODY1),
   1 COMPONENT2 BASED(P2),
      2 DATA CHARACTER(1),
      2 OL     OFFSET(BODY2),
```



B2 → BODY2 = B1 → BODY1;
OHEAD2 = OHEAD1;

Figure 1.4. Assigning a relocatable data list to a new area

```
OHEAD1 OFFSET(BODY1),
OHEAD2 OFFSET(BODY2);
   .        .
   .        .
   .        .
P1 = OHEAD1;
P1 = P1->COMPONENT1.OL;
P1 = P1->COMPONENT1.OL;
P2 = OHEAD2;
P2 = P2->COMPONENT2.OL;
P2 = P2->COMPONENT2.OL;
   .        .
   .        .
   .        .
```

These statements specify that the area variables BODY1 and BODY2 each reserve 500 bytes of based storage and that the based variable DATA associated with each list component is a character string that contains one character. The offset links (OL'S) in B1->BODY1 and the offset head OHEAD1 are declared to be offset with respect to BODY1. Similarly, the offset links in B2->BODY2 and the offset head OHEAD2 are declared to be offset with respect to BODY2. After the above statements are executed, pointer P1 contains the absolute address of the last component in B1->BODY1, and pointer P2 contains the absolute address of the last component in B2->BODY2. The data elements of these two components can then be referred to with the following expressions:

```
P1->COMPONENT1.DATA
P2->COMPONENT2.DATA
```

Figure 1.5 illustrates the organization of a relocatable pointer list and shows how the list can be moved to a new area. Three assignment statements are used to move the list:

```
OHEAD2 = OHEAD1;
B2->BODY2 = B1->BODY1;
D2->DATA_AREA2 = D1->DATA_AREA1;
```

Note that each component of the list contains two offset variables: the offset link OL and the offset data pointer ODP. Both elements must use offset values because pointer

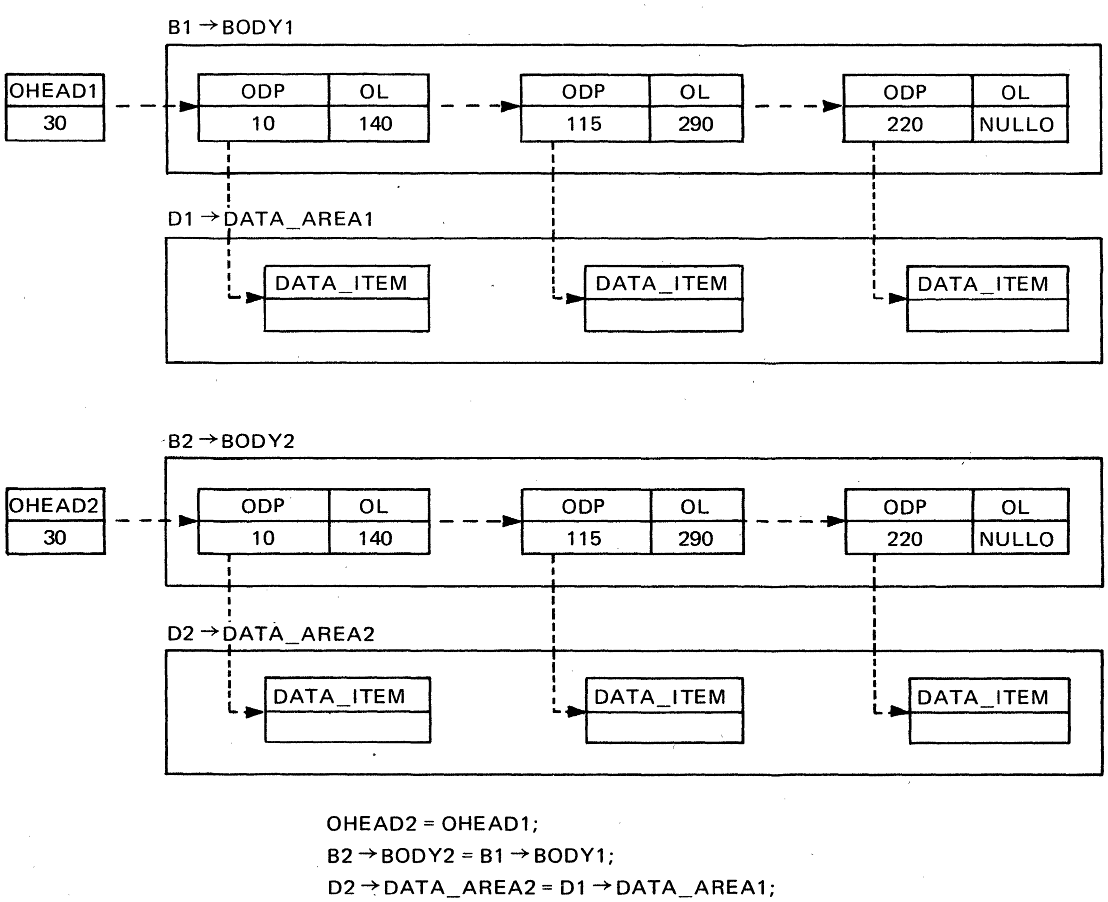


Figure 1.5. Assigning a relocatable pointer list to a new area

values become invalid when moved by area assignment. Movement of the data area of a relocatable pointer list is optional; if the data area is not moved, it can be shared between the old and new versions of the list body.

As with relocatable data lists, offset variables cannot appear as qualifying pointers in references to the based components of a relocatable pointer list; absolute addresses must serve as the qualifying pointers. The following statements show how to obtain the absolute address of the last data item in each of the relocatable pointer lists illustrated in Figure 1.5:

```
DECLARE
   BODY1  AREA(500)  BASED(B1),
   BODY2  AREA(500)  BASED(B1),
   DATA_AREA1  AREA(500)  BASED(D1),
   DATA_AREA2  AREA(500)  BASED(D2),
   1 COMPONENT1 BASED(P1),
      2 ODP OFFSET(DATA_AREA1),
      2 OL  OFFSET(BODY1),
   1 COMPONENT2 BASED(P2),
      2 ODP OFFSET(DATA_AREA2),
      2 OL  OFFSET(BODY2),
   OHEAD1  OFFSET(BODY1),
   OHEAD2  OFFSET(BODY2),
   DATA_ITEM BASED(DATA1) CHARACTER(1),
   (DATA1,DATA2) POINTER;
      .        .
      .        .
      .        .
   P1 = OHEAD1;
   P1 = P1->COMPONENT1.OL;
   P1 = P1->COMPONENT1.OL;
   DATA1 = P1->COMPONENT1.ODP;
   P2 = OHEAD2;
   P2 = P2->COMPONENT2.OL;
   P2 = P2->COMPONENT2.OL;
   DATA2 = P2->COMPONENT2.ODP;
      .        .
      .        .
      .        .
```

These statements specify that the area variables BODY1, BODY2, DATA_AREA1, and DATA_AREA2 each reserve 500 bytes of based storage and that the based variable DATA_ITEM associated with each list component is a character string that contains one character. The offset links (OL's) in B1->BODY1 and the offset head OHEAD1 are declared to be offset with respect to BODY1. Since the data items associated with the list are located in D1->DATA_AREA1, the offset data pointers (ODP's) in B1->BODY1 are offset with respect to DATA_AREA1. Similarly, the offset links (OL's) in B2->BODY2 and the offset head OHEAD2 are declared to be offset with respect to BODY2, and the offset data pointers (ODP's) in B2->BODY2 are offset with respect to DATA_AREA2.

After the above statements are executed, pointer DATA1 contains the absolute address of the data item associated with the last component in B1->BODY1, and pointer DATA2 contains the absolute address of the data item associated with the last component in B2->BODY2. The following expressions can then be used to refer to these two data items:

```
DATA1->DATA_ITEM
DATA2->DATA_ITEM
```

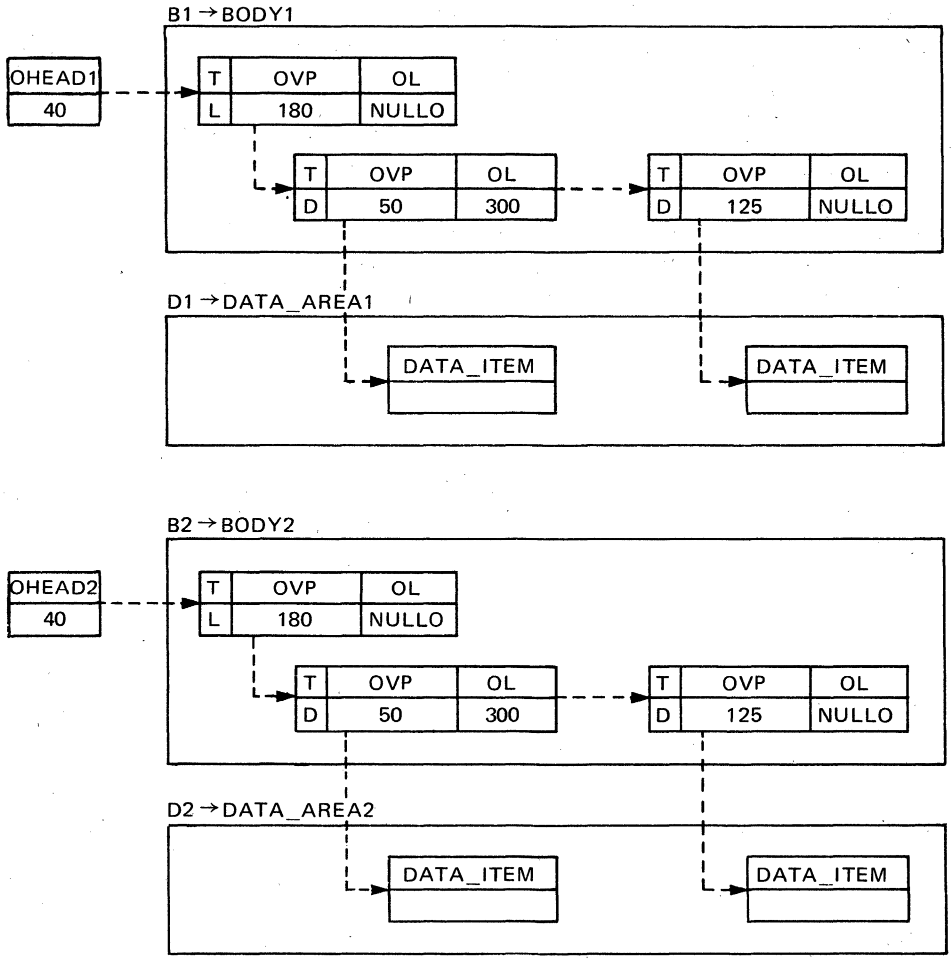
Figure 1.6 illustrates the organization of a relocatable list of lists and shows how the list can be moved to a new area. As with relocatable pointer lists, three assignment statements are used to move the relocatable list of lists:

```
OHEAD2 = OHEAD1;
B2->BODY2 = B1->BODY1;
D2->DATA_AREA2 = D1->DATA_AREA1;
```

Again, each component of the list contains two offset variables: the offset link OL and the offset value pointer OVP. A third element, however, appears in each component of a relocatable list of lists, namely, the type code T, which determines whether the offset value pointer (OVP) specifies the offset address of another list component (type code L) or the offset address of a data item (type code D).

The following statements show declaration of type code T and how to obtain the absolute address of the last data item in each of the relocatable lists of lists illustrated in Figure 1.6:

```
DECLARE
   BODY1  AREA(500)  BASED(B1),
   BODY2  AREA(500)  BASED(B2),
   DATA_AREA1  AREA(500)  BASED(D1),
   DATA_AREA2  AREA(500)  BASED(D2),
   1  D_COMPONENT1  BASED(P1),
      2    T      CHARACTER(1),
      2    PAD    CHARACTER(3),
      2    OVP    OFFSET(DATA_AREA1),
      2    OL     OFFSET(BODY1),
   /*THE PAD ELEMENTS ALIGN THE OFFSETS ON
   FOUR-BYTE BOUNDARIES*/
   1  L_COMPONENT1  BASED(P1),
      2    T      CHARACTER(1),
      2    PAD    CHARACTER(3),
      2    OVP    OFFSET(BODY1),
      2    OL     OFFSET(BODY1),
   1  D_COMPONENT2  BASED(P2),
      2    T      CHARACTER(1),
      2    PAD    CHARACTER(3),
```




```
OHEAD2 = OHEAD1;
B2 →BODY2 = B1 →BODY1;
D2 →DATA_AREA2 = D1 →DATA_AREA1;
```

Figure 1.6. Assigning a relocatable list of lists to a new area

```
   2     OVP     OFFSET(DATA_AREA2),
   2     OL      OFFSET(BODY2),
1  L_COMPONENT2 BASED(P2),
   2     T       CHARACTER(1),
   2     PAD     CHARACTER(3),
   2     OVP     OFFSET(BODY2),
   2     OL      OFFSET(BODY2),
OHEAD1 OFFSET(BODY1),
OHEAD2 OFFSET(BODY2),
DATA_ITEM BASED(DATA1) CHARACTER(1),
(DATA1, DATA2) POINTER;
   .        .
   .        .
   .        .
P1 = OHEAD1;
P1 = P1–>L_COMPONENT1.OVP;
P1 = P1–>D_COMPONENT1.OL;
DATA1 = P1–>D_COMPONENT1.OVP;
P2 = OHEAD2;
P2 = P2–>L_COMPONENT2.OVP;
P2 = P2–>D_COMPONENT2.OL;
DATA2 = P2–>D_COMPONENT2.OVP;
   .        .
   .        .
   .        .
```

These statements specify that the area variables BODY1, BODY2, DATA_AREA1, and DATA_AREA2 each reserve 500 bytes of based storage and that the based variable DATA_ITEM associated with each list component is a character string that contains one character. Because area B1–>BODY1 contains two types of components

(D-components and L-components), separate declarations (D_COMPONENT1 and L_COMPONENT1) are given for each type. The distinction between the two types of components is that the offset value pointer (OVP) in D_COMPONENT1 is offset with respect to DATA_AREA1, while OVP in L_COMPONENT1 is offset with respect to BODY1. However, the offset link (OL) in each type of component is offset with respect to BODY1, and both use a single character for the type code (T). Similarly, the two types of components in B2–>BODY2 are declared as D_COMPONENT2 and L_COMPONENT2. The offset value pointer (OVP) in D_COMPONENT2 is offset with respect to DATA_AREA2, and OVP in L_COMPONENT2 is offset with respect to BODY2. Also, the offset link (OL) for each component type in B2–>BODY2 is offset with respect to BODY2, and both types of components use a single character for the type code (T).

INPUT AND OUTPUT STATEMENTS FOR RELOCATABLE LISTS

The preceding discussions describe how to move a relocatable list from one location to another within internal storage. The main reason, however, for organizing a list in relocatable form is to allow it to be recorded on an external storage medium, such as magnetic tape or magnetic disk, from which it can be retrieved for further processing at a later time. Transmission of a relocatable list to and from an external file requires input and output statements for reading and writing the list. Since PL/I does not permit stream-oriented input and output statements (such as GET and PUT) to read and write the values of pointer variables and offset variables, record-oriented statements (such as READ and LOCATE) must be used to transmit a relocatable list to and from a file. The following discussions describe the effect of the LOCATE and READ statements upon relocatable lists.

The LOCATE Statement

Output transmission of a relocatable list is performed with the LOCATE statement, which has the following form:

```
LOCATE based-variable
   FILE (file-name)
   [SET (pointer-variable)] ;
```

This statement processes sequentially accessed files that are buffered, and allocates within an output buffer (automatically provided for the file) the next available storage position for the specified based variable. The location of the allocated storage is assigned to the pointer variable given in the SET option. The pointer variable allows proper qualification of references to the based variable in the buffer. A SET option, however, need not appear in the LOCATE statement; when it does not, an implied SET is assumed, which uses the pointer variable in the BASED attribute of the specified based variable. After the LOCATE statement has been executed, values can be assigned to the based variable in the buffer. If the based variable is a structure, it may require padding elements for boundary alignment.

Successive executions of the LOCATE statement produce successive allocations of storage in the buffer. An attempt to execute a LOCATE statement when the buffer has become full, momentarily suspends execution of the LOCATE statement and automatically causes the contents of the buffer to be transmitted as a block to the associated file. The buffer is then cleared, and storage is allocated at the beginning of the buffer for the suspended LOCATE statement.

The following statements show how the LOCATE statement may be used to write a relocatable data list into a file:

```
DECLARE
   OHEAD  OFFSET(BODY),
   BODY  AREA(500)  BASED(B),
   1  LIST_RECORD  BASED(RECORD_POINTER),
      2    R_HEAD   OFFSET(DUMMY_BODY),
      2    PADDING CHARACTER(4),
   /*PADDING ALIGNS R_BODY ON AN EIGHT-BYTE
   BOUNDARY IN THE OUTPUT BUFFER*/
      2    R_BODY   AREA(500),
   DUMMY_BODY AREA  BASED(DUMMY_POINTER),
   OUTFILE  FILE RECORD  OUTPUT;
      .         .
      .         .
      .         .
   LOCATE  LIST_RECORD
      FILE(OUTFILE)  SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
     (RECORD_POINTER->R_BODY);
   RECORD_POINTER->R_HEAD = OHEAD;
   RECORD_POINTER->R_BODY = B->BODY;
```

Figure 1.7 illustrates the effect of these statements. B–>BODY and OHEAD form the body area and offset head of the relocatable data list that is transmitted to the output file OUTFILE. Each record in the file is formed from the based variable LIST_RECORD, which contains two elements: R_HEAD and R_BODY. R_HEAD receives the value of OHEAD, and R_BODY receives the contents of B–>BODY.

Observe that R_HEAD is declared to be offset with respect to the based area DUMMY_BODY. Actually, R_HEAD should be offset with respect to based area R_BODY because R_HEAD contains the relative address of the first list component in R_BODY. But R_BODY has a level number of two and, therefore, does not satisfy the

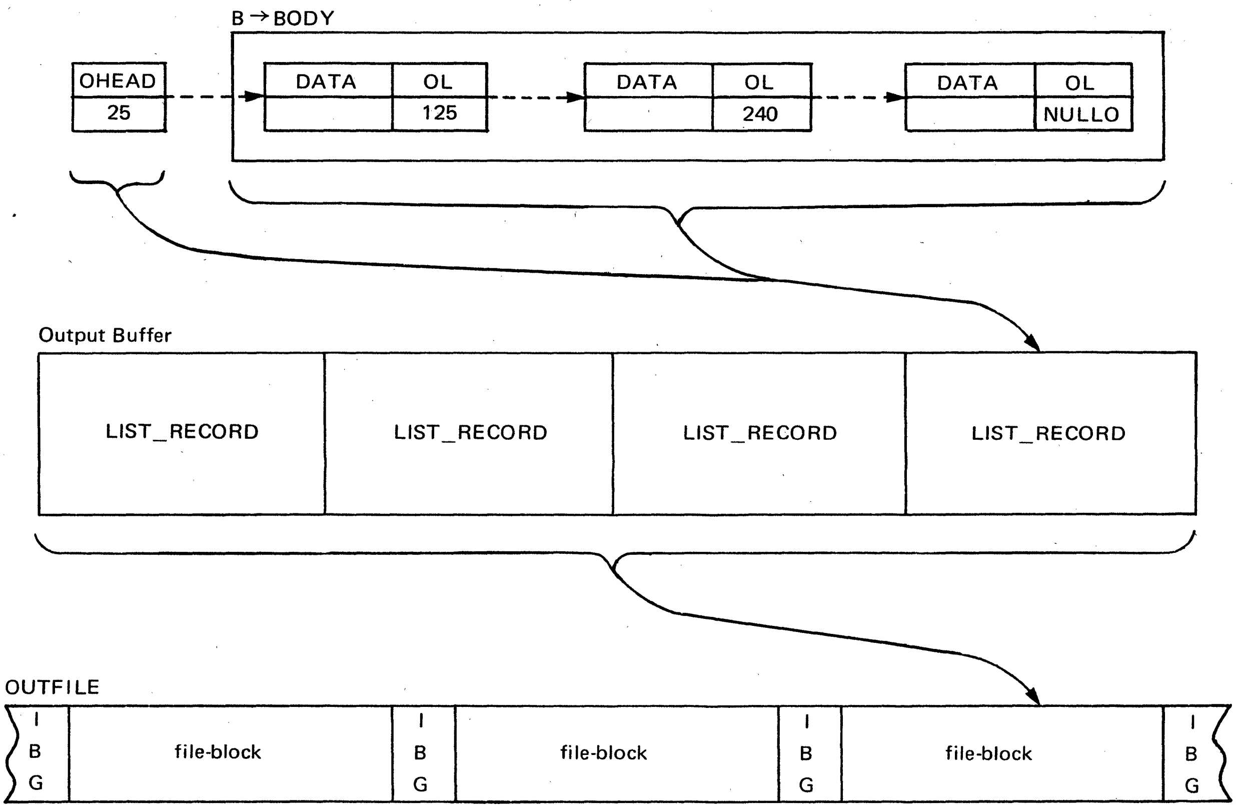


Figure 1.7. How a relocatable data list is transmitted as a logical record to a file

requirement that the based area in an OFFSET attribute must have a level number of one. However, R_HEAD becomes effectively offset with respect to R_BODY when DUMMY_BODY and R_BODY are made to occupy the same location. This overlay is achieved by assigning the address of R_BODY to the pointer variable DUMMY_POINTER associated with DUMMY_AREA.

This example assumes that environmental information, such as record type, record size, block size, input/output device type, unit number, and recording density, is specified in a data definition (DD) statement within the job step that calls for execution of the program under the operating system. The block size determines the size of the buffer, which in Figure 1.7 is assumed to contain storage for four allocations of LIST_RECORD. Also note that, when the size of LIST_RECORD is given in the appropriate DD statement, the size must include additional storage for the internal control information associated with R_BODY. For example, the F-level version of the PL/I compiler adds 16 bytes of internal control information to each area variable. Additional information on this point appears in *IBM System/360 Operating System: PL/I(F) Programmer's Guide* (GC28-6594).

When the contents of the output buffer are transmitted to the file, they are written as a block (also called a physical record). Figure 1.7 shows successive blocks recorded in OUTFILE, which is assumed to be on magnetic tape. Each block is separated by an interblock gap (IBG) and contains up to four logical records (that is, four allocations of LIST_RECORD). The number of logical records in a block can be changed by specifying a different block size in the associated DD statement.

The transmission of a relocatable data list to an external file has been discussed. The following discussion pertains to the transmission of relocatable pointer lists and lists of lists to an external file.

To write a relocatable pointer list or list of lists into a file, it is necessary to transmit the data area of the list along with its head and body. The following statements show how the head, body, and data area can be combined into a single logical record:

```
DECLARE
   OHEAD OFFSET(BODY),
   BODY AREA(500) BASED(B),
   DATA_AREA AREA(500) BASED(D),
   1 LIST_RECORD BASED(RECORD_POINTER),
      2 R_HEAD OFFSET(DUMMY_BODY),
      2 PAD1 CHARACTER(4),
      2 R_BODY AREA(500),
```

```
   2 PAD2 CHARACTER(4),
   2 R_DATA_AREA AREA(500),
DUMMY_BODY AREA BASED(DUMMY_POINTER),
OUTFILE FILE RECORD OUTPUT;
   .        .
   .        .
   .        .
LOCATE LIST_RECORD
   FILE(OUTFILE) SET(RECORD_POINTER);
DUMMY_POINTER = ADDR(RECORD_POINTER->
   R_BODY);
RECORD_POINTER->R_HEAD = OHEAD;
RECORD_POINTER->R_BODY = B->BODY;
RECORD_POINTER->R_DATA_AREA = D->
   DATA_AREA;
   .        .
   .        .
   .        .
```

These statements apply to both pointer lists and lists of lists because each type of list contains a head, a body, and a data area. The statements are also similar to those of the preceding example except that the data area of the list is included in the record transmitted to the file.

Inclusion of the data area in the logical record, however, may cause the record size to become too large and thus require additional buffer storage. A more convenient record size can be obtained by splitting the list into two logical records. The first record can contain the head and body of the list, and the second record can contain the data area. This type of transmission is obtained with the following statements:

```
DECLARE
   OHEAD OFFSET(BODY),
   BODY AREA(500) BASED(B),
   DATA_AREA AREA(500) BASED(D),
   1 HEAD_BODY_RECORD BASED(RECORD_
      POINTER),
      2 R_HEAD OFFSET(DUMMY_BODY),
      2 PAD CHARACTER(4),
      2 R_BODY AREA(500),
   DUMMY_BODY AREA BASED(DUMMY_POINTER),
   OUTFILE FILE RECORD OUTPUT;
      .        .
      .        .
      .        .
   LOCATE HEAD_BODY_RECORD
      FILE(OUTFILE) SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR(RECORD_POINTER->
      R_BODY);
   RECORD_POINTER->R_HEAD = OHEAD;
   RECORD_POINTER->R_BODY = B->BODY;
   LOCATE DATA_AREA
      FILE(OUTFILE) SET(RECORD_POINTER);
   RECORD_POINTER->DATA_AREA = D->
      DATA_AREA;
      .        .
      .        .
      .        .
```

Figure 1.8 illustrates the effect of these statements on a relocatable list of lists. The first LOCATE statement obtains storage in the output buffer for the logical record HEAD_BODY_RECORD, which receives the head and body of the relocatable list. The second LOCATE statement allocates storage in the output buffer for DATA_AREA, which is written as an individual logical record. The buffer in Figure 1.8 contains four logical records (for two lists), with HEAD_BODY_RECORD and DATA_AREA occupying alternate positions. When the buffer becomes full it is automatically written into OUTFILE and cleared for further transmission.

The READ Statement

After relocatable lists have been written into a file, they can be retrieved from the file for additonal processing. Retrieval is accomplished with a READ statement:

```
READ FILE(file-name) SET(pointer-variable);
```

This statement obtains the location of the next logical record in an input buffer associated with the specified file and assigns the location to the pointer variable given in the SET option. A based variable qualified by the same pointer will then relate to the fields of the logical record; the based variable is effectively overlaid on the logical record in the buffer.

The following statements demonstrate how a relocatable data list can be read from a file:

```
DECLARE
   OHEAD OFFSET(BODY),
   BODY AREA(500) BASED(B),
   1 LIST_RECORD BASED(RECORD_POINTER),
      2 R_HEAD OFFSET(DUMMY_BODY),
      2 PAD CHARACTER(4),
      2 R_BODY AREA(500),
   DUMMY_BODY AREA BASED(DUMMY_POINTER),
   INFILE FILE RECORD INPUT;
      .        .
      .        .
      .        .
```

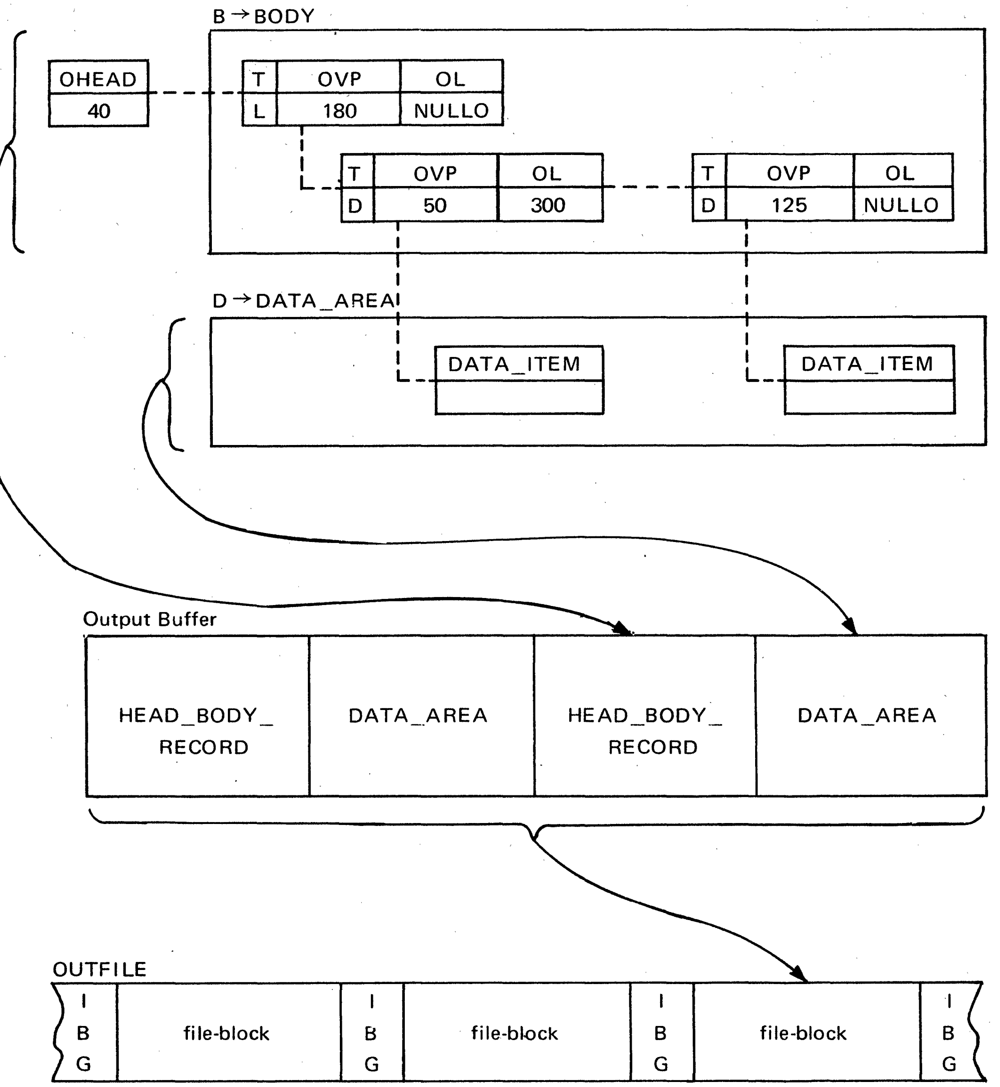


Figure 1.8. How a relocatable list of lists is transmitted as two logical records to a file

```
READ FILE(INFILE) SET(RECORD_POINTER);
DUMMY_POINTER = ADDR
   (RECORD_POINTER->R_BODY);
OHEAD = RECORD_POINTER->R_HEAD;
B->BODY = RECORD_POINTER->R_BODY);
   .      .
   .      .
   .      .
```

Figure 1.9 illustrates the effect of these statements. The READ statement obtains the address of the next occurrence of LIST_RECORD in the input buffer associated with INFILE and assigns the address to RECORD_POINTER. The head and body of the relocatable list are then assigned to OHEAD and B–>BODY by the following statements:

```
OHEAD = RECORD_POINTER->R_HEAD;
B->BODY = RECORD_POINTER->R_BODY;
```

Each execution of the READ statement advances the value of RECORD_POINTER to the location of the next logical record in the buffer. When the end of the buffer is reached and an attempt is made to read another logical record, the program automatically refills the buffer with the next block from INFILE and assigns the address of the first logical record in the buffer to RECORD_POINTER. This process is repeated until the end of the file is reached.

When a relocatable pointer list or list of lists is read from a file, the data area of the list must also be retrieved along with the head and body of the list. The following statements show how to read a relocatable pointer list or list of

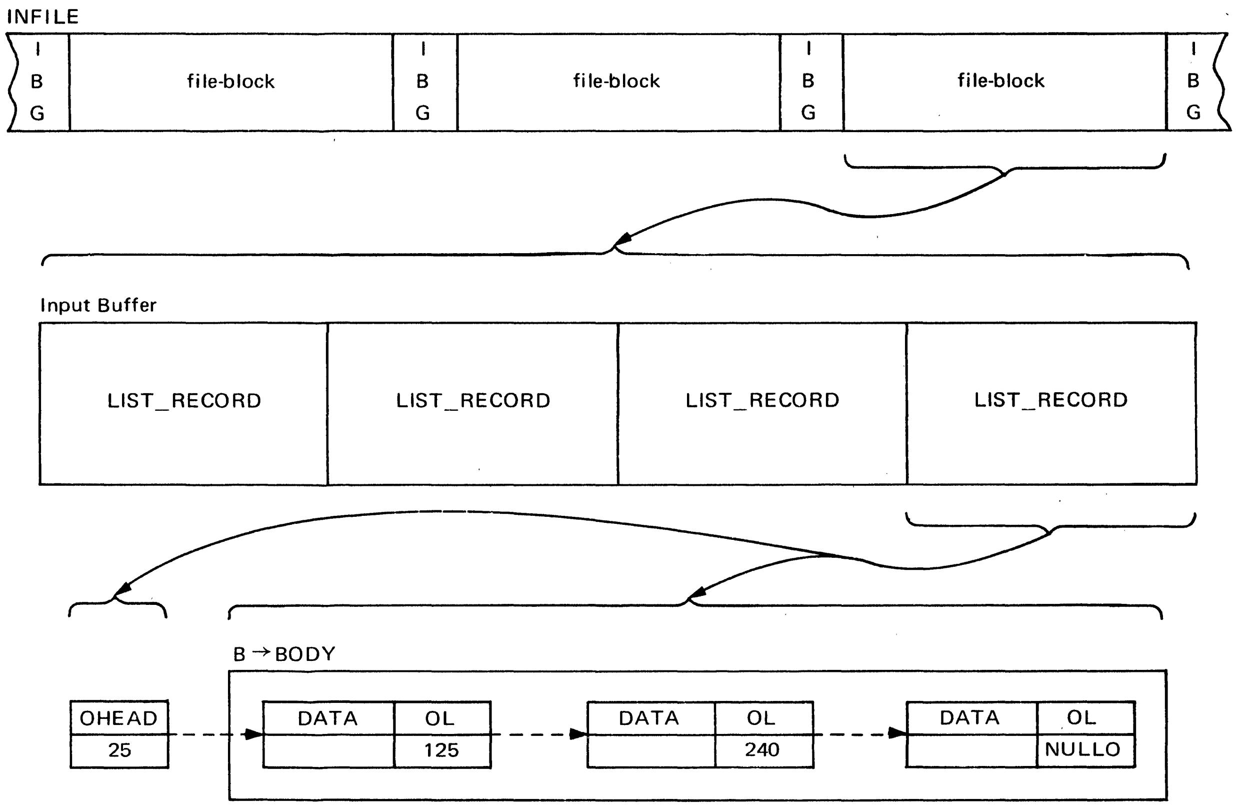


Figure 1.9. How a relocatable data list is retrieved as a logical record from a file

lists when the head, body, and data area are contained in a single logical record:

```
DECLARE
   OHEAD  OFFSET(BODY),
   BODY  AREA(500)  BASED(B),
   DATA_AREA  AREA(500)  BASED(D),
   1  LIST_RECORD BASED(RECORD_POINTER),
      2  R_HEAD  OFFSET(DUMMY_BODY),
      2  PAD1  CHARACTER(4),
      2  R_BODY  AREA(500),
      2  PAD2  CHARACTER(4),
      2  R_DATA_AREA  AREA(500),
   DUMMY_BODY  AREA  BASED(DUMMY_POINTER),
   INFILE  FILE  RECORD  INPUT;
      .          .
      .          .
      .          .
   READ FILE(INFILE) SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
      (RECORD_POINTER->R_BODY);
   OHEAD = RECORD_POINTER->R_HEAD;
   B->BODY = RECORD_POINTER->
      R_BODY;
   D->DATA_AREA = RECORD_POINTER->
      R_DATA_AREA;
      .          .
      .          .
      .          .
```

These statements retrieve either a relocatable pointer list or a relocatable list of lists because both types contain a head, a body, and a data area. This example is similar to the preceding example except that, in this example, the retrieved record contains a data area.

Had the list originally been split and recorded in the file as two logical records, one for the head and body, the other for the data area, then the following statements could be used to retrieve the list:

```
DECLARE
   OHEAD  OFFSET(BODY),
   BODY  AREA(500)  BASED(B),
   DATA_AREA  AREA(500)  BASED(D),
   1  HEAD_BODY_RECORD BASED(RECORD_
      POINTER),
      2  R_HEAD  OFFSET(DUMMY_BODY),
      2  PAD  CHARACTER(4),
```

```
    2  R_BODY  AREA(500),
DUMMY_BODY AREA BASED(DUMMY_POINTER),
INFILE  FILE  RECORD  INPUT;
    .          .
    .          .
    .          .
READ FILE(INFILE)  SET(RECORD_POINTER);
DUMMY_POINTER = ADDR(RECORD_POINTER->
   R_BODY);
OHEAD = RECORD_POINTER->R_HEAD;
B->BODY = RECORD_POINTER->
   R_BODY;
READ FILE(INFILE) SET(RECORD_POINTER);
D->DATA_AREA = RECORD_POINTER->
   DATA_AREA;
    .          .
    .          .
    .          .
```

Figure 1.10 illustrates the effect of these statements on a relocatable list of lists. The first READ statement obtains the location of the next logical record (HEAD_BODY_RECORD) in the input buffer associated with INFILE and assigns the location to RECORD_POINTER. The head and body of the list are then assigned to OHEAD and B->BODY. The second READ statement obtains the address of the next logical record (DATA_AREA) in the input buffer and assigns the address to RECORD_POINTER. The data area is then moved from the buffer to D->DATA_AREA.

Self-Defining Records

So far, the LOCATE and READ statements have been restricted to fixed-length records, but it is also possible to apply these statements to self-defining records. Such records contain a specification of their own size, which per-

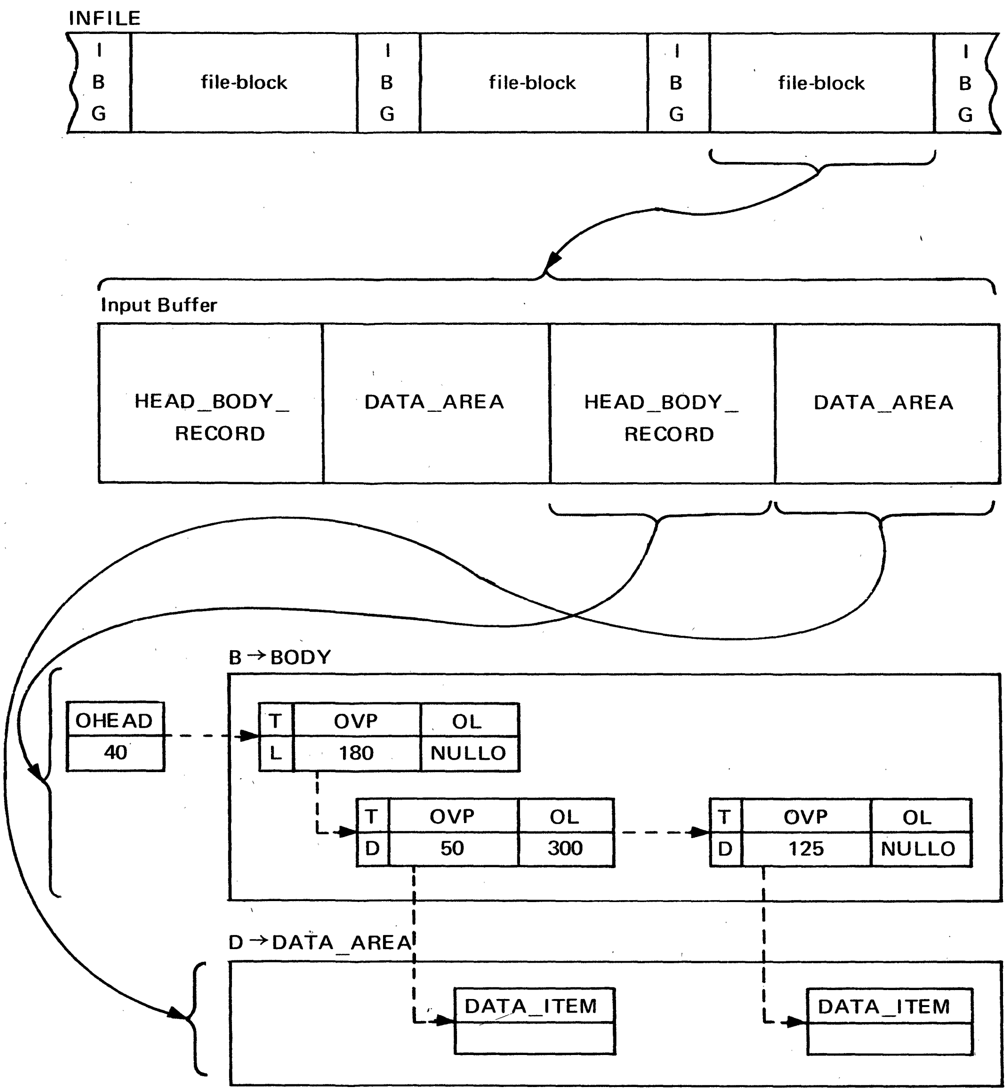


Figure 1.10. How a relocatable list of lists is retrieved as two logical records from a file

mits them to vary in length. They prove useful in handling the varying storage requirements associated with list-processing techniques.

The declaration of a self-defining record must be made with a based structure that contains an adjustable string length, adjustable area size, or adjustable array bound, the value of which is maintained by a variable within the structure. This variable, however, cannot possess a value until storage has been allocated for the containing based structure; otherwise, there would be no storage to hold the value of the variable. Since the amount of storage to be allocated depends on the value of this variable, a facility is needed for associating a value with the variable before allocation.

PL/I provides this facility through the REFER option, which has the following general format:

```
element-variable REFER(element-variable)
```

Both element variables in the option must be unsubscripted fixed-point binary variables having the same precision. The variable to the right of the keyword REFER must be an element of the self-defining based structure, but the variable to the left must be declared outside the structure. The option itself must appear as a string length, area size, or array bound within the structure. As an example, consider the following DECLARE statement:

```
DECLARE
   DUMMY_BODY AREA BASED (DUMMY_POINTER),
   BINARY_BODY_SIZE FIXED BINARY(16,0),
   1  LIST_RECORD BASED  (RECORD_POINTER),
      2  R_HEAD  OFFSET(DUMMY_BODY),
      2  R_BODY_SIZE  FIXED BINARY(16,0),
      2  R_BODY AREA(BINARY_BODY_SIZE
            REFER(R_BODY_SIZE));
```

LIST_RECORD is declared to be a self-defining based structure, which contains three components: R_HEAD, R_BODY_SIZE, and R_BODY. This declaration can be used to generate a self-defining record for a relocatable data list, in which R_HEAD serves as the offset head of the list and R_BODY contains the relocatable components of the list. The area attribute for R_BODY uses a REFER option to specify the size of the area:

```
BINARY_BODY_SIZE REFER
   (R_BODY_SIZE)
```

When storage is allocated for LIST_RECORD, the size of R_BODY is obtained from BINARY_BODY_SIZE (which is declared outside LIST_RECORD) and is automatically assigned to R_BODY_SIZE (which is declared inside LIST_RECORD). It is the programmer's responsibility to assign the proper value to BINARY_BODY_SIZE before storage is allocated for LIST_RECORD. By changing the value of BINARY_BODY_SIZE, the programmer can vary the size of R_BODY within each generation of LIST_RECORD.

The following example shows how LIST_RECORD may acquire different lengths when used to write two relocatable data lists into a file:

```
DECLARE
   OHEAD1  OFFSET(BODY1),
   BODY1  AREA(500)  BASED(B),
   OHEAD2  OFFSET(BODY2),
   BODY2  AREA(750)  BASED(B),
   DUMMY_BODY AREA BASED(DUMMY_POINTER),
   BINARY_BODY_SIZE  FIXED BINARY(16,0),
   OUTFILE  FILE RECORD OUTPUT,
   1 LIST_RECORD BASED(RECORD_POINTER),
      2  R_HEAD  OFFSET(DUMMY_BODY),
      2  R_BODY_SIZE FIXED BINARY(16,0),
      2  R_BODY AREA(BINARY_BODY_SIZE
            REFER(R_BODY_SIZE));
   .        .
   .        .
   .        .
   BINARY_BODY_SIZE = 500;
   LOCATE  LIST_RECORD
      FILE(OUTFILE)  SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
      (RECORD_POINTER->R_BODY);
   RECORD_POINTER->R_HEAD = OHEAD1;
   RECORD_POINTER->R_BODY = B->BODY1;
   .        .
   .        .
   .        .
   BINARY_BODY_SIZE = 750;
   LOCATE LIST_RECORD
      FILE(OUTFILE)  SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
      (RECORD_POINTER->R_BODY);
   RECORD_POINTER->R_HEAD = OHEAD2;
   RECORD_POINTER->R_BODY = B->BODY2;
   .        .
   .        .
   .        .
```

OHEAD1 and B->BODY1 form the offset head and body area of the first list, while OHEAD2 and B->BODY2 serve as the corresponding parts of the second list. The two body areas have different storage sizes: BODY1 contains 500 bytes, and BODY2 contains 750 bytes. Before the first list is transmitted to OUTFILE, the value 500 is assigned to BINARY_BODY_SIZE. Execution of the LOCATE statement for LIST_RECORD causes 500 bytes of buffer storage to be allocated for R_BODY within LIST_RECORD and this size to be assigned automatically to

R_BODY_SIZE. The following statements then fill LIST_RECORD with the offset head and body area of the first list:

```
RECORD_POINTER->R_HEAD = OHEAD1;
RECORD_POINTER->R_BODY = B->BODY1;
```

The same process is used to write the second list into OUTFILE, but BINARY_BODY_SIZE is set equal to 750 before storage is allocated for LIST_RECORD. This value causes the size of area R_BODY to change from 500 bytes to 750 bytes.

Retrieval of these two lists is illustrated by the following example:

```
DECLARE
   OHEAD1  OFFSET(BODY1),
   BODY1  AREA(500)  BASED(B),
   OHEAD2  OFFSET(BODY2),
   BODY2  AREA(750)  BASED(B),
   DUMMY_BODY AREA BASED(DUMMY_POINTER),
   BINARY_BODY_SIZE FIXED BINARY(16,0),
   (BODY_SIZE1, BODY_SIZE2) FIXED DECIMAL(5),
   INFILE  FILE RECORD  INPUT,
   1  LIST_RECORD  BASED(RECORD_POINTER),
      2  R_HEAD  OFFSET(DUMMY_BODY),
      2  R_BODY_SIZE FIXED BINARY(16,0),
      2  R_BODY AREA(BINARY_BODY_SIZE
            REFER(R_BODY_SIZE));
      .          .
      .          .
      .          .
   READ FILE(INFILE) SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
      (RECORD_POINTER->R_BODY);
   OHEAD1 = RECORD_POINTER->R_HEAD;
   B->BODY1 = RECORD_POINTER->R_BODY;
   BODY_SIZE1 = RECORD_POINTER->
      R_BODY_SIZE;
      .          .
      .          .
      .          .
```

The first READ statement retrieves a logical record from INFILE and assigns the location of the record to RECORD_POINTER. This assignment causes the based structure LIST_RECORD to be overlaid on the record. The example assumes that the retrieved record contains the offset head, body size, and body area for a relocatable data list that is to be assigned to OHEAD1 and B->BODY1.

The REFER option in LIST_RECORD indicates that the size of area R_BODY can vary and is automatically determined by the value of R_BODY_SIZE. Execution of the following statements causes the head and body of the retrieved list to be assigned to OHEAD1 and B->BODY1:

```
OHEAD1 = RECORD_POINTER->R_HEAD;
B->BODY1 = RECORD_POINTER->R_BODY;
```

Note that, although BINARY_BODY_SIZE appears at the left of the REFER option, its value is not used or changed in any way by the READ statement. Only a LOCATE statement could make use of BINARY_BODY_SIZE. In this example, the size of the first retrieved body area is assigned, for further use, to the variable BODY_SIZE1 by the statement:

```
BODY_SIZE1 = RECORD_POINTER->
   R_BODY_SIZE;
```

Similar steps are used to read the next relocatable data list from INFILE and to assign it to OHEAD2 and B->BODY2. The size of the body area for this second list is assigned to BODY_SIZE2.

PL/I(F) allows one REFER option in the declaration of a self-defining based structure. When the REFER option specifies a string length or an area size, the string or area must be an element variable and must be the last element in the structure declaration. If the REFER option appears as an array bound, the bound must be the upper bound of the leftmost dimension in the array declaration, and the REFER option must also belong to the last array variable in the self-defining structure or to a minor structure that contains the last element of the self-defining structure.

Earlier examples, illustrated in Figures 1.8 and 1.10, showed how to write and read body areas and data areas as separate logical records that are not self-defining. The following discussion shows how those examples can be modified to handle self-defining records. Consider the following example:

```
DECLARE
   OHEAD1  OFFSET(BODY1)
   BODY1  AREA(500)  BASED(B),
   DATA1  AREA(1000)  BASED(D),
   OHEAD2  OFFSET(BODY2),
   BODY2  AREA(750)  BASED(B),
   DATA2  AREA(1000)  BASED(D),
   DUMMY_BODY  AREA  BASED(DUMMY_POINTER),
   BINARY_SIZE  FIXED BINARY(16,0),
   OUTFILE  FILE  RECORD  OUTPUT,
   1 HEAD_BODY_RECORD BASED
      (RECORD_POINTER),
      2  R_HEAD  OFFSET(DUMMY_BODY),
      2  R_BODY_SIZE FIXED BINARY(16,0),
      2  R_BODY  AREA
         (BINARY_SIZE REFER(R_BODY_SIZE)),
   1 DATA_RECORD  BASED(RECORD_POINTER),
```

```
      2 R_DATA_SIZE FIXED BINARY(16,0),
      2 PAD CHARACTER(4),
      2 R_DATA AREA(BINARY_SIZE
        REFER(R_DATA_SIZE));
      .       .
      .       .
      .       .
   BINARY_SIZE = 500;
   LOCATE HEAD_BODY_RECORD
      FILE(OUTFILE) SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
      (RECORD_POINTER->R_BODY);
   RECORD_POINTER->R_HEAD = OHEAD1;
   RECORD_POINTER->R_BODY = B->BODY1;
      .       .
      .       .
      .       .
   BINARY_SIZE = 1000;
   LOCATE DATA_RECORD
      FILE(OUTFILE) SET(RECORD_POINTER);
   RECORD_POINTER->R_DATA = D->DATA1;
      .       .
      .       .
      .       .
   BINARY_SIZE = 750;
   LOCATE HEAD_BODY_RECORD
      FILE(OUTFILE) SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
      (RECORD_POINTER->R_BODY);
   RECORD_POINTER->R_HEAD = OHEAD2;
   RECORD_POINTER->R_BODY = B->BODY2;
      .       .
      .       .
      .       .
   BINARY_SIZE = 1000;
   LOCATE DATA_RECORD
      FILE(OUTFILE) SET(RECORD_POINTER);
   RECORD_POINTER->R_DATA = D->DATA2;
      .       .
      .       .
      .       .
```

This example applies to relocatable pointer lists and lists of lists. It uses the self-defining based structure HEAD_BODY_RECORD for the offset head and body area of each list, and the self-defining based structure DATA_RECORD for the data area. Two lists are written. The offset head, body area, and data area of the first list are specified by OHEAD1, BODY1, and DATA1, while OHEAD2, BODY2, and DATA2 denote the corresponding parts of the second list. BODY1 and DATA1 contain 500 and 1000 bytes each, and BODY2 and DATA2 contain 750 and 1000 bytes each. These sizes are transmitted with the associated self-defining records.

Retrieval of these two lists is illustrated by the following example:

```
DECLARE
   OHEAD1 OFFSET(BODY1),
   BODY1 AREA(500) BASED(B),
   DATA1 AREA(1000) BASED(D),
   OHEAD2 OFFSET(BODY2),
   BODY2 AREA(750) BASED(B),
   DATA2 AREA(1000) BASED(D),
   DUMMY_BODY AREA BASED(DUMMY_POINTER),
   BINARY_SIZE FIXED BINARY(16,0),
   (SIZE1, SIZE2, SIZE3, SIZE4) FIXED DECIMAL(5),
   INFILE FILE RECORD INPUT,
   1 HEAD_BODY_RECORD BASED
      (RECORD_POINTER),
      2 R_HEAD OFFSET(DUMMY_BODY),
      2 R_BODY_SIZE FIXED BINARY(16,0),
      2 R_BODY AREA(BINARY_SIZE REFER
         (R_BODY_SIZE)),
   1 DATA_RECORD BASED(RECORD_POINTER),
      2 R_DATA_SIZE FIXED BINARY(16,0),
      2 PAD CHARACTER(4),
      2 R_DATA AREA(BINARY_SIZE REFER
         (R_DATA SIZE));
      .       .
      .       .
      .       .
   READ FILE(INFILE) SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
      (RECORD_POINTER->R_BODY);
   OHEAD1 = RECORD_POINTER->R_HEAD;
   B->BODY1 = RECORD_POINTER->R_BODY;
   SIZE1 = RECORD_POINTER->R_BODY_SIZE;
      .       .
      .       .
      .       .
   READ FILE(INFILE) SET(RECORD_POINTER);
   D->DATA1 = RECORD_POINTER->R_DATA;
   SIZE2 = RECORD_POINTER->R_DATA_SIZE;
      .       .
      .       .
      .       .
   READ FILE(INFILE) SET(RECORD_POINTER);
   DUMMY_POINTER = ADDR
      (RECORD_POINTER->R_BODY);
   OHEAD2 = RECORD_POINTER->R_HEAD;
   B->BODY2 = RECORD_POINTER->R_BODY;
   SIZE3 = RECORD_POINTER->R_BODY_SIZE;
      .       .
      .       .
      .       .
   READ FILE(INFILE) SET(RECORD_POINTER);
   D->DATA2 = RECORD_POINTER->R_DATA;
   SIZE4 = RECORD_POINTER->R_BODY_SIZE;
      .       .
      .       .
      .       .
```

This example uses the same self-defining based structure HEAD_BODY_RECORD and DATA_RECORD as the preceding example. It retrieves two lists. The parts of the first list are assigned to OHEAD1, BODY1, and DATA1, and those of the second list are assigned to OHEAD2, BODY2, and DATA2. The sizes of BODY1 and DATA1 are assigned to SIZE1 and SIZE2 for possible use by other statements. SIZE3 and SIZE4 receive the sizes of BODY2 and DATA2.

Chapter 2. Processing Relocatable Lists

The following discussion develop subroutines that use the relocation facilities described in the preceding chapter. No attempt is made, however, at creating a collection of procedures for relocatable lists. Instead, it is assumed that lists will usually be created and manipulated in absolute form and then converted to relocatable form when they are to be moved to new locations or transmitted to files. This approach restricts the procedures needed for relocatable lists to five categories:

1. Converting absolute lists to relocatable form
2. Converting relocatable lists to absolute form
3. Moving relocatable lists
4. Writing relocatable lists
5. Reading relocatable lists

Each category contains subroutines for three types of lists: data lists, pointer lists, and lists of lists.

The subroutines in these categories are designed to process an arbitray number of relocatable lists in each area and are not limited to areas that contain a single list. The heads of all the relocatable lists in an area are passed to each subroutine as an array of offset variables. This convention permits the number of offset heads in the array (and consequently, the number of lists in the area) to vary while at the same time allowing the number of arguments in each invocation to remain constant. As an example, Figure 2.1 shows the area B–>BODY_AREA with three relocatable data lists, the heads of which are individual offset variables. The offset head OAVAIL is assumed to identify the relocatable list of available storage components in the area. The same area and lists appear in Figure 2.2, but the offset heads of the lists have been assigned to the array OHEAD_ARRAY. The subroutines in the following discussions assume an arbitrary size for the array of offset heads and transmit the array as a self-defining record.

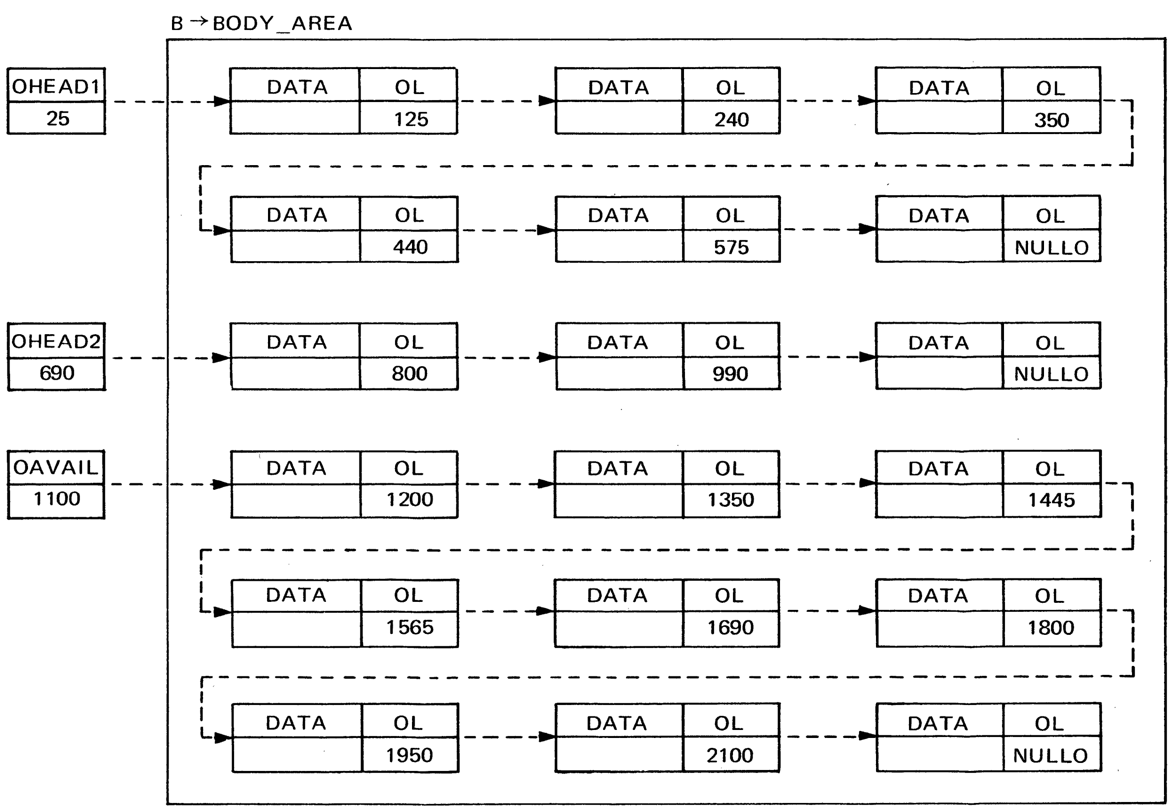


Figure 2.1. Relocatable lists with individual offset heads

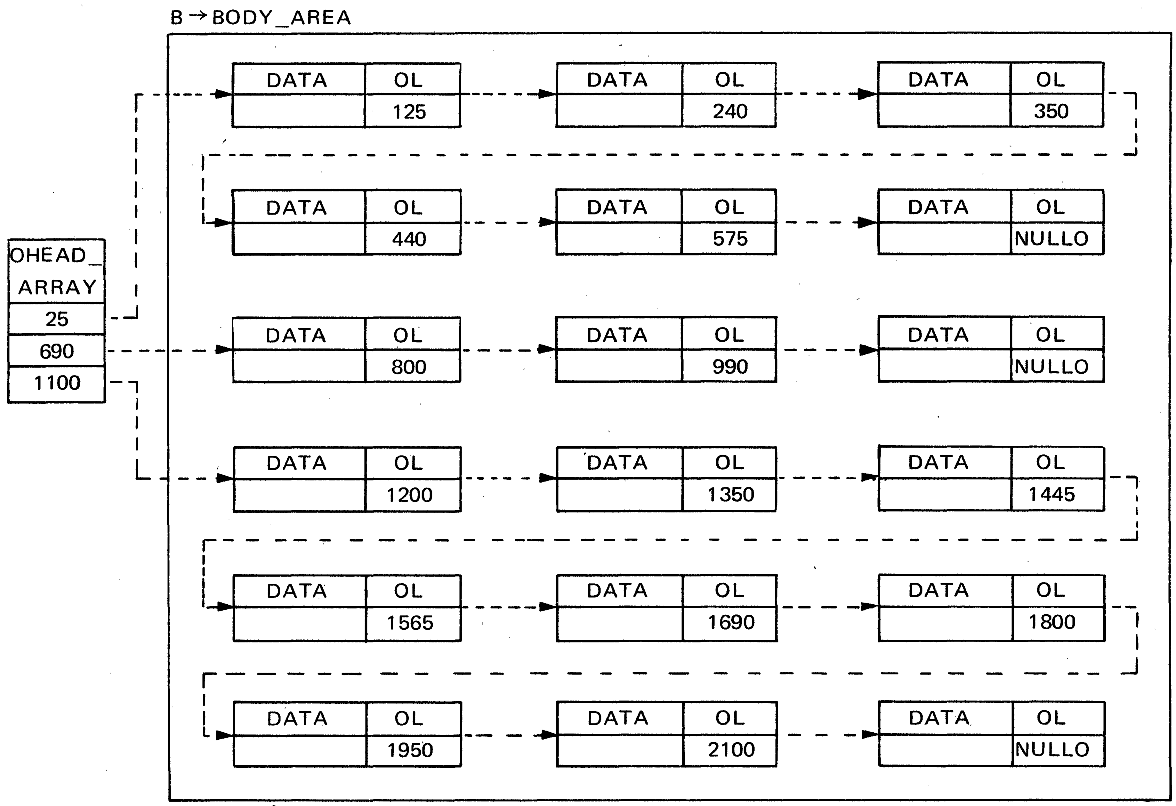


Figure 2.2. Relocatable lists with their offset heads stored in an array

CONVERTING ABSOLUTE LISTS TO RELOCATABLE FORM

The following discussions develop three subroutines for converting the absolute lists in one area to relocatable lists in another area:

1. CON_DAR, which converts data lists from absolute to relocatable form
2. CON_PAR, which converts pointer lists from absolute to relocatable form
3. CON_LAR, which converts lists of lists from absolute to relocatable form

These subroutines are used after the absolute lists have been constructed and processed by other routines and are ready to be moved to new storage locations or to be written into files.

CON_DAR Subroutine

Figures 2.3A and 2.3B present the CON_DAR subroutine, which converts absolute data lists in one area to relocatable data lists in another area. The subroutine uses four arguments: the body area and head array of the absolute data lists being converted, and the body area and head array that are to receive the relocatable data lists during conversion.

CON_DAR Subroutine

Purpose

To convert data lists from absolute to relocatable form

Reference

CON_DAR(BODY_AREA1, HEAD_ARRAY, BODY_AREA2, OHEAD_ARRAY)

Entry-Name Declaration

```
DECLARE CON_DAR
      ENTRY(AREA(*),(*)POINTER, AREA(*),
      (*)OFFSET(DUMMY_BODY_AREA));
```

Meaning of Arguments

BODY_AREA1	–	the area that contains the bodies of the absolute lists being converted to relocatable form
HEAD_ARRAY	–	the array that contains the pointer heads of the absolute lists in BODY_AREA1
BODY_AREA2	–	the area that receives the bodies of the lists after they have been converted to relocatable form
OHEAD_ARRAY	–	the array that receives the offset heads of the relocatable lists in BODY_AREA2

Remarks

BODY_AREA1 and BODY_AREA2 can have any storage class and be of arbitrary and unequal size. If BODY_AREA2 is not large enough to receive the converted components of BODY_AREA1, or if OHEAD_ARRAY is smaller than HEAD_ARRAY, or if HEAD_ARRAY is completely null, then OHEAD_ARRAY is filled with null offset values, and the content of BODY_AREA2 becomes undefined.

Other Programmer-Defined Procedures Required

None

Method

Each absolute list in BODY_AREA1 is reconstructed, component by component, as a relocatable list in BODY_AREA2. The data element of each component is a single character.

Figure 2.3A. Description of the CON_DAR subroutine for converting data lists from absolute to relocatable form

```
CON_DAR:
        PROCEDURE(BODY_AREA1, HEAD_ARRAY,
        BODY_AREA2, OHEAD_ARRAY);
 DECLARE
        (DUMMY_POINTER,C1,C2)POINTER,
        (BODY_AREA1, BODY_AREA2) AREA(*),
        DUMMY_BODY_AREA BASED
        (DUMMY_POINTER) AREA,
        (HEAD_ARRAY(*), SAVE) POINTER,
        OHEAD_ARRAY(*) OFFSET
        (DUMMY_BODY_AREA),
        1 COMPONENT1 BASED(C1),
        2 DATA CHARACTER(1),
        2 LINK POINTER,
        1 COMPONENT2 BASED(C2),
        2 DATA CHARACTER(1),
        2 OLINK OFFSET(DUMMY_BODY_AREA);
        /* IF AREA CONDITION OCCURS,
        BODY_AREA2 IS TOO SMALL TO RECEIVE
        CONTENTS OF BODY_AREA1.  GO TO
        NULL_LIST.*/
        ON AREA
      GO TO
        NULL_LIST;
        /* IF OHEAD_ARRAY IS SMALLER THAN
        HEAD_ARRAY, GO TO NULL_LIST */
   IF
        DIM(OHEAD_ARRAY,1)<DIM(HEAD_ARRAY,1)
    THEN
      GO TO
        NULL_LIST;
        /* ASSOCIATE OFFSETS OHEAD_ARRAY AND
        OLINK WITH BODY_AREA2.*/
        DUMMY_POINTER = ADDR(BODY_AREA2);
        /* CONVERT SUCCESSIVE DATA LISTS IN
        BODY_AREA1 TO RELOCATABLE DATA LISTS
        IN BODY_AREA2.*/
        J=LBOUND(OHEAD_ARRAY,1)-1;
BEGIN_CONVERT_LOOP:
  DO
        I=LBOUND(HEAD_ARRAY,1) TO HBOUND
        (HEAD_ARRAY,1);
        J = J+1;
        /* IF I-TH POINTER IN HEAD_ARRAY IS
        NULL, SET J-TH OFFSET IN OHEAD_ARRAY
        TO NULLO, AND CONVERT NEXT LIST IN
        BODY_AREA1.*/
   IF
        HEAD_ARRAY(I) = NULL
    THEN
  DO;
        OHEAD_ARRAY(J) = NULLO;
      GO TO
        END_CONVERT_LOOP;
END;
        /* ALLOCATE COMPONENT2 IN
        BODY_AREA2, AND ASSIGN TO THE
        ALLOCATION THE DATA VALUE OF THE
        FIRST COMPONENT IN THE I-TH LIST IN
        BODY_AREA1.*/
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        OHEAD_ARRAY(J), SAVE = C2;
        C1 = HEAD_ARRAY(I);
        C2->COMPONENT2.DATA = C1->
        COMPONENT1.DATA;
        /* PERFORM SUCCESSIVE ALLOCATIONS OF
        COMPONENT2 IN BODY_AREA1, AND ASSIGN
        TO THE ALLOCATIONS THE DATA VALUE OF
        SUCCESSIVE COMPONENTS IN THE I-TH
        LIST WITHIN BODY_AREA1.*/
        C1 = C1->LINK;
  DO
        WHILE (C1¬=NULL);
        ALLOCATE COMPONENT2 IN (BODY_AREA2)
        SET(C2);
        SAVE->OLINK, SAVE = C2;
        C2->COMPONENT2.DATA = C1->
        COMPONENT1.DATA;
        C1 = C1->LINK;
END;
        /* ASSIGN A NULL OFFSET VALUE TO
        OLINK IN LAST COMPONENT OF J-TH LIST
        IN BODY_AREA2.*/
        SAVE->OLINK = NULLO;
        /* CONVERT NEXT LIST IN BODY_AREA1
        BY EXECUTING NEXT CYCLE OF CONVERT
        LOOP.*/
END_CONVERT_LOOP:
END;
        /* THIS POINT IS REACHED WHEN ALL
        DATA LISTS IN BODY_AREA1 HAVE BEEN
        CONVERTED TO RELOCATABLE DATA LISTS
        IN BODY_AREA2.  THEREFORE, RETURN
        SUBROUTINE CONTROL TO POINT OF
        INVOCATION.*/
        RETURN;
        /* IF THIS POINT IS REACHED, ASSIGN
        A NULL OFFSET VALUE TO EACH ELEMENT
        OF OHEAD_ARRAY.*/
NULL_LIST:
        OHEAD_ARRAY = NULLO;
END
        CON_DAR;
```

Figure 2.3B. The CON_DAR subroutine used to convert data lists from absolute to relocatable form

CON_PAR Subroutine

Figures 2.4A and 2.4B present the CON_PAR subroutine, which converts absolute pointer lists to relocatable form. The subroutine uses five arguments: the body area and head array of the absolute pointer lists being converted, the body area and head array that are to receive the relocatable pointer lists during conversion, and the data area, which is shared by both the absolute and relocatable forms of the pointer lists.

CON_PAR Subroutine

Purpose

To convert pointer lists from absolute to relocatable form

Reference

```
CON_PAR (BODY_AREA1, HEAD_ARRAY
         BODY_AREA2, OHEAD_ARRAY,
         DATA_AREA)
```

Entry-Name Declaration

```
DECLARE CON_PAR
    ENTRY(AREA(*), (*) POINTER,
          AREA(*), (*) OFFSET
          (DUMMY_BODY_AREA), AREA(*));
```

Meaning of Arguments

BODY_AREA1	–	the area that contains the bodies of the absolute lists being converted to relocatable form
HEAD_ARRAY	–	the array that contains the pointer heads of the absolute lists in BODY_AREA1
BODY_AREA2	–	the area that receives the bodies of the lists after they have been converted to relocatable form
OHEAD_ARRAY	–	the array that receives the offset heads of the relocatable lists in BODY_AREA2
DATA_AREA	–	the area that contains the data values of the lists before and after conversion

Remarks

BODY_AREA1, BODY_AREA2, and DATA_AREA can have any storage class and be of arbitrary size. If BODY_AREA2 is not large enough to receive the converted components of BODY_AREA1, or if OHEAD_ARRAY is smaller than HEAD_ARRAY, or if HEAD_ARRAY is completely null, then OHEAD_ARRAY is filled with null offset values, and the content of BODY_AREA2 becomes undefined.

Other Programmer-Defined Procedures Required

None

Method

Each absolute list in BODY_AREA1 is reconstructed, component by component, as are relocatable lists in BODY_AREA2. After conversion, both types of lists share DATA_AREA.

Figure 2.4A. Description of the CON_PAR subroutine for converting pointer lists from absolute to relocatable form

```
CON_PAR:
        PROCEDURE(BODY_AREA1, HEAD_ARRAY,
        BODY_AREA2, OHEAD_ARRAY,
        DATA_AREA);
 DECLARE
        (BODY_AREA1, BODY_AREA2, DATA_AREA)
        AREA(*),
        DUMMY_BODY_AREA BASED(DUMMY_POINTER1)
        AREA,
        DUMMY_DATA_AREA BASED(DUMMY_POINTER2)
        AREA,
        (HEAD_ARRAY(*), SAVE) POINTER,
        OHEAD_ARRAY(*)
        OFFSET(DUMMY_BODY_AREA),
        1 COMPONENT1 BASED(C1),
        2 DATA POINTER,
        2 LINK POINTER,
        1 COMPONENT2 BASED(C2),
        2 ODATA OFFSET(DUMMY_DATA_AREA),
        2 OLINK OFFSET(DUMMY_DATA_AREA);
        /* IF AREA CONDITION OCCURS,
        BODY_AREA2 IS TOO SMALL TO
        RECEIVE CONTENTS OF BODY_AREA1. GO
        TO NULL_LIST. */
        ON AREA
      GO TO
        NULL_LIST;
        /* IF HEAD_ARRAY IS NULL, GC TC
        NULL_LIST. */
  DO I = LBOUND(HEAD_ARRAY,1)
        TO HBOUND(HEAD_ARRAY,1);
  IF HEAD_ARRAY(I) ¬= NULL
  THEN GO TO GO2;
END;
      GO TO
        NULL_LIST;
GO2:
        /* IF CHEAD_ARRAY IS SMALLER THAN
        HEAD_ARRAY, GO TO NULL_LIST. */
   IF
        DIM(OHEAD_ARRAY,1)<DIM(HEAD_ARRAY,1)
    THEN
      GO TO
        NULL_LIST;
        /* ASSOCIATE OFFSETS OHEAD_ARRAY AND
        OLINK WITH BODY_AREA2 AND OFFSET
        ODATA WITH DATA_AREA. */
        DUMMY_POINTER1 = ADDR(BODY_AREA2);
        DUMMY_POINTER2 = ADDR(DATA_AREA);
        /* CONVERT SUCCESSIVE POINTER LISTS
        IN BODY_AREA1 TO RELOCATABLE
        POINTER LISTS IN BODY_AREA2. */
        J = LBOUND(OHEAD_ARRAY,1)-1;
BEGIN_CONVERT_LOOP:
  DO
        I = LBOUND(HEAD_ARRAY,1)
        TO HBOUND(HEAD_ARRAY,1);
        J = J + 1;
        /* IF I-TH POINTER IN HEAD_ARRAY IS
        NULL, SET J-TH OFFSET IN OHEAD _ARRAY
        TO NULLC, AND CONVERT NEXT LIST IN
        BODY_AREA1. */
   IF
        HEAD_ARRAY(I) = NULL
     THEN
   DO;
        OHEAD_ARRAY(J) = NULLO;
      GO TO
        END_CONVERT_LOOP;
END;
        /* ALLOCATE COMPONENT2 IN
        BODY_AREA2, AND ASSIGN TO THE
        ALLOCATION THE DATA POINTER OF THE
        FIRST COMPONENT IN THE I-TH LIST IN
        BODY_AREA1. */
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        OHEAD_ARRAY(J), SAVE = C2;
        C1 = HEAD_ARRAY(I);
    IF
        DATA = NULL
     THEN
        ODATA = NULLC;
     ELSE
        ODATA = DATA;
        /* PERFORM SUCCESSIVE ALLOCATIONS
        OF COMPONENT2 IN BODY_AREA2, AND
        ASSIGN TO THE ALLOCATIONS THE DATA
        POINTER OF SUCCESSIVE COMPONENTS IN
        THE I-TH LIST WITHIN BODY_AREA1. */
        C1 = C1->LINK;
  DO
        WHILE(C1¬=NULL);
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE->OLINK,SAVE = C2;
    IF
        DATA = NULL
     THEN
        ODATA = NULLO;
     ELSE
        ODATA = DATA;
        C1 = C1->LINK;
END;
        /* ASSIGN A NULL OFFSET VALUE TC
        OLINK IN LAST COMPONENT OF J-TH LIST
        IN BODY_AREA2. */
        SAVE->OLINK = NULLO;
        /* CONVERT NEXT LIST IN BODY_AREA1
        BY EXECUTING NEXT CYCLE OF CONVERT
        LOOP. */
END_CONVERT_LOOP:
END;
        /* THIS POINT IS REACHED WHEN ALL
        POINTER LISTS IN BODY_AREA1 HAVE
        BEEN CONVERTED TO RELOCATABLE
        POINTER LISTS IN BODY_AREA2,
        THEREFORE, RETURN SUBROUTINE CONTROL
        TO POINT OF INVOCATION. */
        RETURN;
        /* IF THIS POINT IS REACHED, ASSIGN
        A NULL OFFSET VALUE TO EACH ELEMENT
        OF OHEAD_ARRAY. */
NULL_LIST:
        OHEAD_ARRAY = NULLO;
END
        CON_PAR;
```

Figure 2.4B. The CON_PAR subroutine used to convert absolute pointer lists to relocatable form

CON_LAR Subroutine

Figures 2.5A and 2.5B present the CON_LAR subroutine, which converts absolute lists of lists to relocatable form. The subroutine uses six arguments: the body area and head array of the absolute lists of lists being converted, the body area and head array that are to receive the relocatable lists of lists during conversion, the data area, which is shared by both the absolute and relocatable forms of the lists of lists, and the number of sublists.

The code in CON_LAR indicates an optional use of the recursive function procedure CONV shown in the Appendix. CONV examines the type code in each list component and takes appropriate conversion action. CONV returns an offset value.

CON_LAR Subroutine

Purpose

To convert lists of lists from absolute to relocatable form

Reference

```
CON_LAR(BODY_AREA1,HEAD_ARRAY,
        BODY_AREA2, OHEAD_ARRAY,
        DATA_AREA,#SUBS)
```

Entry-Name Declaration

```
DECLARE CON_LAR
     ENTRY(AREA(*), (*)POINTER,
          AREA(*), (*)OFFSET
          (DUMMY_BODY_AREA), AREA(*),
          FIXED DECIMAL);
```

Meaning of Arguments

BODY_AREA1	–	the area that contains the bodies of the absolute lists being converted to relocatable form
HEAD_ARRAY	–	the array that contains the pointer heads of the absolute lists in BODY_AREA1
BODY_AREA2	–	the area that receives the bodies of the lists after they have been converted to relocatable form
OHEAD_ARRAY	–	the array that receives the offset heads of the relocatable lists in BODY_AREA2
DATA_AREA	–	the area that contains the data values of the lists before and after conversion
#SUBS	–	the number of sublists

Remarks

BODY_AREA1, BODY_AREA2, and DATA_AREA can have any storage class and be of arbitrary size. If BODY_AREA2 is not large enough to receive the converted components of BODY_AREA1, or if OHEAD_ARRAY is smaller than HEAD_ARRAY, or if HEAD_ARRAY is completely null, then OHEAD_ARRAY is filled with null offset values, and the content of BODY_AREA2 becomes undefined.

Other Programmer-Defined Procedures Required

CONV (optional)

Method

Each absolute list in BODY_AREA1 is reconstructed, component by component, as a relocatable list in BODY_AREA2. After conversion, both types of lists share DATA_AREA.

Figure 2.5A. Description of the CON_LAR subroutine for converting lists of lists from absolute to relocatable form

```
CON_LAR:
        PROCEDURE(BODY_AREA1,HEAD_ARRAY,
        BODY_AREA2, OHEAD_ARRAY, DATA_AREA,
        #SUBS);
 DECLARE
        #SUBS FIXED DECIMAL,
        (SAVE, KEEP, PA(#SUBS)) POINTER,
        /* PARAMETER #SUBS IS NOT NECESSARY
        WHEN FUNCTION CONV IS USED */
        (DUMMY_BODY_POINTER,
        DUMMY_DATA_POINTER,C1,C2)POINTER,
        (BODY_AREA1,BODY_AREA2,DATA_AREA)
        AREA(*),
        DUMMY_BODY_AREA
        BASED(DUMMY_BODY_POINTER) AREA,
        DUMMY_DATA_AREA
        BASED(DUMMY_DATA_POINTER) AREA,
        HEAD_ARRAY(*) POINTER,
        OHEAD_ARRAY(*)
        OFFSET(DUMMY_BODY_AREA),
        1 COMPONENT1 BASED(C1),
        2 TYPE CHARACTER(1),
        2 VALUE POINTER,
        2 LINK POINTER,
        1 D_COMPONENT2 BASED(C2),
        2 D_OTYPE CHARACTER(1),
        2 D_OVALUE OFFSET(DUMMY_DATA_AREA),
        2 D_OLINK OFFSET(DUMMY_BODY_AREA),
        1 L_COMPONENT2 BASED(C2),
        2 L_OTYPE CHARACTER(1),
        2 L_OVALUE OFFSET(DUMMY_BODY_AREA),
        2 L_OLINK OFFSET(DUMMY_BODY_AREA);
        /* IF AREA CONDITION OCCURS,
        BODY_AREA2 IS TOO SMALL TO RECEIVE
        CONTENTS OF BODY_AREA1. GO TO
        NULL_LIST. */
        ON AREA
      GO TO
        NULL_LIST;
        /* IF OHEAD_ARRAY IS SMALLER THAN
        HEAD_ARRAY, GO TO NULL_LIST. */
   IF
        DIM(OHEAD_ARRAY,1)<DIM(HEAD_ARRAY,1)
    THEN
      GO TO
        NULL_LIST;
        /* ASSOCIATE OFFSETS OHEAD_ARRAY,
        D_OLINK, L_OLINK, AND L_OVALUE WITH
        BODY_AREA2, AND OFFSET D_OVALUE WITH
        DATA_AREA. */
        DUMMY_BODY_POINTER =
        ADDR(BODY_AREA2);
        DUMMY_DATA_POINTER =
        ADDR(DATA_AREA);
        /* CONVERT SUCCESSIVE LISTS OF
        LISTS IN BODY_AREA1 TO
        RELOCATABLE LISTS OF LISTS IN
        BODY_AREA2. */
        PA = NULL;
        J = LBOUND(OHEAD_ARRAY,1)-1;
BEGIN_CONVERT_LOOP:
  DO
        I = LBOUND(HEAD_ARRAY,1)
        TO HBOUND(HEAD_ARRAY,1);
        J = J + 1;
        K = 1;
        C1 = HEAD_ARRAY(I);
        IF C1 = NULL THEN DO;
        OHEAD_ARRAY(J) = NULLO;
        GO TO END_CONVERT_LOOP;
        END;
/* OPTION */
/* TEST */ IF #SUBS ¬= 1 THEN GO TO NO_CONV;
    USE_CONV:
        /* USE THE FOLLOWING CODE
        TO EMPLOY FUNCTION CONV
        FOR CONVERSIONS */
        OHEAD_ARRAY(J) = CONV(HEAD_ARRAY(I),
        BODY_AREA1, BODY_AREA2, DATA_AREA);
        GO TO END_CONVERT_LOOP;
/* END OF OPTION */
    NO_CONV:
        ALLOCATE L_COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE, KEEP, OHEAD_ARRAY(J) = C2;
        C2->L_OTYPE = 'L';
        PA(K) = C1->VALUE;
        C1 = C1->LINK;
        DO WHILE(C1 ¬= NULL);
        K = K + 1;
        ALLOCATE L_COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE->L_OLINK = C2;
        SAVE = C2;
        C2->L_OTYPE = 'L';
        PA(K) = C1->VALUE;
        C1 = C1->LINK;
        END;
        SAVE->L_OLINK = NULLO;
  D_LIST:
        DO L = 1 TO #SUBS;
        C1 = PA(L);
        IF C1 = NULL THEN GOTO END_D_LIST;
        ALLOCATE D_COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE = C2;
        KEEP->L_OVALUE = C2;
        KEEP = KEEP->L_OLINK;
        C2->D_OTYPE = 'D';
        IF C1->VALUE = NULL
        THEN C2->D_OVALUE = NULLO;
        ELSE C2->D_OVALUE = C1->VALUE;
        C1 = C1->LINK;
        DO WHILE(C1 ¬= NULL);
        ALLOCATE D_COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE->D_OLINK = C2;
        SAVE = C2;
        C2->D_OTYPE = 'D';
        IF C1->VALUE = NULL
        THEN C2->D_OVALUE = NULLO;
        ELSE C2->D_OVALUE = C1->VALUE;
        C1 = C1->LINK;
        END;
        SAVE->D_OLINK = NULLO;
   END_D_LIST: END;
END_CONVERT_LOOP:
END;
        /* WHEN THIS POINT IS REACHED, ALL
        LISTS OF LISTS IN BODY_AREA1 HAVE
        BEEN CONVERTED TO RELOCATABLE LISTS
        OF LISTS IN BODY_AREA2. THEREFORE,
        RETURN SUBROUTINE CONTROL TO POINT
        OF INVOCATION. */
        RETURN;
        /* IF THIS POINT IS REACHED, ASSIGN
        A NULL OFFSET VALUE TO EACH
        ELEMENT OF OHEAD_ARRAY. */
NULL_LIST:
        OHEAD_ARRAY = NULLO;
END
        CON_LAR;
```

Figure 2.5B. The CON_LAR subroutine used to convert absolute lists of lists to relocatable form

CONVERTING RELOCATABLE LISTS TO ABSOLUTE FORM

The following discussions develop three subroutines for converting the relocatable lists in one area to absolute lists in another area:

1. CON_DRA, which converts data lists from relocatable to absolute form
2. CON_PRA, which converts pointer lists from relocatable to absolute form
3. CON_LRA, which converts lists of lists from relocatable to absolute form

These subroutines are used after the relocatable lists have been retrieved from files or moved to new storage locations. Conversion of the lists to absolute form permits them to be processed by routines that accept only absolute lists.

CON_DRA Subroutine

Figures 2.6A and 2.6B present the CON_DRA subroutine, which converts data lists from relocatable to absolute form. The subroutine uses four arguments: the body area and head array of the relocatable lists being converted, and the body area and head array that are to receive the absolute lists during conversion.

CON_DRA Subroutine

Purpose

To convert data lists from relocatable to absolute form

Reference

```
CON_DRA(BODY_AREA1,OHEAD_ARRAY,
        BODY_AREA2,HEAD_ARRAY)
```

Entry-Name Declaration

```
DECLARE CON_DRA
     ENTRY(AREA(*),(*)OFFSET
     (DUMMY_BODY_AREA),AREA(*),
     (*)POINTER);
```

Meaning of Arguments

BODY_AREA1	–	the area that contains the bodies of the relocatable lists being converted to absolute form
OHEAD_ARRAY	–	the array that contains the offset heads of the relocatable lists in BODY_AREA1
BODY_AREA2	–	the area that receives the bodies of the lists after they have been converted to absolute form
HEAD_ARRAY	–	the array that receives the pointer heads of the absolute lists in BODY_AREA2

Remarks

BODY_AREA1 and BODY_AREA2 can have any storage class and be of arbitrary and unequal size. If BODY_AREA2 is not large enough to receive the converted components of BODY_AREA1, or if HEAD_ARRAY is smaller than OHEAD_ARRAY, or if all positions of OHEAD_ARRAY contain null offset values, then HEAD_ARRAY is filled with null pointer values, and the content of BODY_AREA2 becomes undefined.

Other Programmer-Defined Procedures Required

None

Method

Each relocatable list in BODY_AREA1 is reconstructed, component by component, as an absolute list in BODY_AREA2. The data element of each component is a single character.

Figure 2.6A. Description of the CON_DRA subroutine for converting data lists from relocatable to absolute form

```
CON_DRA:
        PROCEDURE (BODY_AREA1,OHEAD_ARRAY,
        BODY_AREA2,HEAD_ARRAY);
 DECLARE
        (DUMMY_POINTER,C1,C2)POINTER,
        (BODY_AREA1,BODY_AREA2)AREA(*),
        DUMMY_BODY_AREA BASED
        (DUMMY_POINTER)AREA,
        OHEAD_ARRAY(*) OFFSET
        (DUMMY_BODY_AREA),
        (HEAD_ARRAY(*), SAVE, TEMP) POINTER,
        1 COMPONENT1 BASED(C1),
        2 DATA CHARACTER(1),
        2 OLINK OFFSET(DUMMY_BODY_AREA),
        1 COMPONENT2 BASED(C2),
        2 DATA CHARACTER(1),
        2 LINK POINTER;
        /* IF AREA CONDITION OCCURS,
        BODY_AREA2 IS TOO SMALL TO RECEIVE
        CONTENTS OF BODY_AREA1. GO TO
        NULL_LIST.*/
        ON AREA
      GO TO
        NULL_LIST;
        /* IF HEAD_ARRAY IS SMALLER THAN
        OHEAD_ARRAY, GO TO NULL_LIST.*/
   IF
        DIM(HEAD_ARRAY,1)<DIM(OHEAD_ARRAY,1)
    THEN
      GO TO
        NULL_LIST;
        /* ASSOCIATE OFFSETS OHEAD_ARRAY AND
        OLINK WITH BODY_AREA1.*/
        DUMMY_POINTER = ADDR(BODY_AREA1);
        /* CONVERT EACH RELOCATABLE DATA
        LIST IN BODY_AREA1 TO AN ABSOLUTE
        DATA LIST IN BODY_AREA2.*/
        J = LBOUND(HEAD_ARRAY,1)-1;
BEGIN_CONVERT_LOOP:
  DO
        I = LBOUND(OHEAD_ARRAY,1) TO HBOUND
        (OHEAD_ARRAY,1);
        J = J+1;
        /* IF I-TH OFFSET IN OHEAD_ARRAY IS
        NULLO, SET J-TH POINTER IN
        HEAD_ARRAY TO NULL, AND CONVERT NEXT
        LIST IN BODY_AREA1.*/
   IF
        OHEAD_ARRAY(I) = NULLO
    THEN
  DO;
        HEAD_ARRAY(J) = NULL;
      GO TO
        END_CONVERT_LOOP;
END;
        /* ALLOCATE COMPONENT2 IN
        BODY_AREA2, AND ASSIGN TO THE
        ALLOCATION THE DATA VALUE OF THE
        FIRST COMPONENT IN THE I-TH LIST IN
        BODY_AREA1.*/
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET (C2);
        HEAD_ARRAY(J), SAVE = C2;
        TEMP = OHEAD_ARRAY(I);
        C2->COMPONENT2.DATA = TEMP->
        COMPONENT1.DATA;
        /* PERFORM SUCCESSIVE ALLOCATIONS OF
        COMPONENT2 IN BODY_AREA2, AND ASSIGN
        TO THE ALLOCATIONS THE DATA VALUES
        OF SUCCESSIVE COMPONENTS IN THE I-TH
        LIST WITHIN BODY_AREA1.*/
        C1 = TEMP->OLINK;
        DO WHILE(C1 ¬= NULL);
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE->LINK, SAVE = C2;
        C2->COMPONENT2.DATA = C1->
        COMPONENT1.DATA;
    IF
        C1->OLINK = NULLO
     THEN
        C1 = NULL;
     ELSE
        C1 = C1->OLINK;
END;
        /* ASSIGN A NULL VALUE TO LINK IN
        LAST COMPONENT OF J-TH LIST IN
        BODY_AREA2.*/
        SAVE->LINK = NULL;
        /* CONVERT NEXT LIST IN BODY_AREA1
        BY EXECUTING NEXT CYCLE OF CONVERT
        LOOP.*/
END_CONVERT_LOOP:
END;
        /* THIS POINT IS REACHED WHEN ALL
        RELOCATABLE LISTS IN BODY_AREA1
        HAVE BEEN CONVERTED TO ABSOLUTE
        LISTS IN BODY_AREA2.  THEREFORE,
        RETURN SUBROUTINE CONTROL TO POINT
        OF INVOCATION.*/
        RETURN;
        /* IF THIS POINT IS REACHED, ASSIGN
        A NULL VALUE TO EACH ELEMENT OF
        HEAD_ARRAY.*/
NULL_LIST:
        HEAD_ARRAY = NULL;
END
        CON_DRA;
```

Figure 2.6B. The CON_DRA subroutine used to convert data lists from relocatable to absolute form

CON_PRA Subroutine

Figures 2.7A and 2.7B present the CON_PRA subroutine, which converts relocatable pointer lists to absolute form. The subroutine uses five arguments: the body area and head array of the relocatable pointer lists being converted, the body area and head array that are to receive the absolute pointer lists during conversion, and the data area, which is shared by both the relocatable and absolute forms of the pointer lists.

CON_PRA Subroutine

Purpose

To convert pointer lists from relocatable to absolute form

Reference

```
CON_PRA(BODY_AREA1, OHEAD_ARRAY,
        BODY_AREA2, HEAD_ARRAY,
        DATA_AREA)
```

Entry-Name Declaration

```
DECLARE CON_PRA
    ENTRY(AREA(*), (*)OFFSET
    (DUMMY_BODY_AREA), AREA(*),
    (*)POINTER, AREA(*));
```

Meaning of Arguments

BODY_AREA1 – the area that contains the bodies of the relocatable lists being converted to absolute form

OHEAD_ARRAY – the array that contains the offset heads of the relocatable lists in BODY_AREA1

BODY_AREA2 – the area that receives the bodies of the lists after they have been converted to absolute form

HEAD_ARRAY – the array that receives the pointer heads of the absolute lists in BODY_AREA2

DATA_AREA – the area that contains the data values of the lists before and after conversion

Remarks

BODY_AREA1, BODY_AREA2, and DATA_AREA can have any storage class and be of arbitrary size. If BODY_AREA2 is not large enough to receive the converted components of BODY_AREA1, or if HEAD_ARRAY is smaller than OHEAD_ARRAY, or if all positions of OHEAD_ARRAY contain null offset values, then HEAD_ARRAY is filled with null pointer values, and the content of BODY_AREA2 becomes undefined.

Other Programmer-Defined Procedures Required

None

Method

Each relocatable list in BODY_AREA1 is reconstructed, component by component, as an absolute list in BODY_AREA2. After conversion, both types of lists share DATA_AREA.

Figure 2.7A. Description of the CON_PRA subroutine for converting pointer lists from relocatable to absolute form

```
CON_PRA:
        PROCEDURE(BODY_AREA1, OHEAD_ARRAY,
        BODY_AREA2, HEAD_ARRAY, DATA_AREA);
 DECLARE
        (BODY_AREA1,BODY_AREA2,DATA_AREA)
        AREA(*),
        DUMMY_BODY_AREA
        BASED(DUMMY_POINTER1) AREA,
        DUMMY_DATA_AREA
        BASED(DUMMY_POINTER2) AREA,
        (HEAD_ARRAY(*), SAVE, TEMP) POINTER,
        OHEAD_ARRAY(*)
        OFFSET(DUMMY_BODY_AREA),
        1 COMPONENT1 BASED(C1),
        2 ODATA OFFSET(DUMMY_DATA_AREA),
        2 OLINK OFFSET(DUMMY_BODY_AREA),
        1 COMPONENT2 BASED(C2),
        2 DATA POINTER,
        2 LINK POINTER;
        /* IF AREA CONDITION OCCURS,
        BODY_AREA2 IS TOO SMALL TO RECEIVE
        CONTENTS OF BODY_AREA1. GO TO
        NULL_LIST. */
        ON AREA
      GO TO
        NULL_LIST;
        /* IF ALL OFFSET VALUES IN
        OHEAD_ARRAY ARE NULLO, GO TO
        NULL_LIST. */
  DO I = LBOUND(OHEAD_ARRAY,1)
  TO HBOUND(OHEAD_ARRAY,1);
  IF OHEAD_ARRAY(I) ¬= NULLO
  THEN GO TO GO5;
END;
   GO TO NULL_LIST;
GO5:
        /* IF HEAD_ARRAY IS SMALLER THAN
        OHEAD_ARRAY GO TO NULL_LIST. */
   IF
        DIM(HEAD_ARRAY,1)<DIM(OHEAD_ARRAY,1)
    THEN
      GO TO
        NULL_LIST;
        /* ASSOCIATE OFFSETS OHEAD_ARRAY AND
        OLINK WITH BODY_AREA2 AND OFFSET
        ODATA WITH DATA_AREA. */
        DUMMY_POINTER1 = ADDR(BODY_AREA2);
        DUMMY_POINTER2 = ADDR(DATA_AREA);
        /* CONVERT EACH RELOCATABLE POINTER
        LIST IN BODY_AREA1 TO AN ABSOLUTE
        POINTER LIST IN BODY_AREA2. */
        J = LBOUND(HEAD_ARRAY,1)-1;
BEGIN_CONVERT_LOOP:
  DO
        I = LBOUND(OHEAD_ARRAY,1)
        TO HBOUND(OHEAD_ARRAY,1);
        J = J + 1;
        /* IF I-TH OFFSET IN OHEAD_ARRAY IS
        NULLO, SET J-TH POINTER IN
        HEAD_ARRAY TO NULL, AND CONVERT NEXT
        LIST IN BODY_AREA1. */
    IF
        OHEAD_ARRAY(I) = NULLO
     THEN
   DO;
        HEAD_ARRAY(J) = NULL;
      GO TO
        END_CONVERT_LOOP;
END;
        /* ALLOCATE COMPONENT2 IN BODY_AREA2,
        AND ASSIGN TO THE ALLOCATION THE
        DATA POINTER OF THE FIRST COMPONENT
        IN THE I-TH LIST IN BODY_AREA1 */
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        HEAD_ARRAY(J), SAVE = C2;
        TEMP = OHEAD_ARRAY(I);
        C2->DATA = TEMP->ODATA;
        /* PERFORM SUCCESSIVE ALLOCATIONS OF
        COMPONENT2 IN BODY_AREA2, AND ASSIGN
        TO THE ALLOCATIONS THE DATA POINTER
        OF SUCCESSIVE COMPONENTS IN THE I-TH
        LIST WITHIN BODY_AREA1. */
        C1 = TEMP->OLINK;
      DO WHILE
        (C1¬=NULL);
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE->LINK, SAVE = C2;
        C2->DATA = C1->ODATA;
    IF
        C1->OLINK = NULLO
     THEN
        C1 = NULL;
     ELSE
        C1 = C1->OLINK;
END;
        /* ASSIGN A NULL VALUE TO LINK IN
        LAST COMPONENT OF J-TH LIST IN
        BODY_AREA2. */
        SAVE->LINK = NULL;
        /* CONVERT NEXT LIST IN BODY_AREA1
        BY EXECUTING NEXT CYCLE OF CONVERT
        LOOP. */
END_CONVERT_LOOP:
END;
        /* THIS POINT IS REACHED WHEN ALL
        RELOCATABLE LISTS IN BODY_AREA1
        HAVE BEEN CONVERTED TO ABSOLUTE LISTS
        IN BODY_AREA2. THEREFORE, RETURN
        SUBROUTINE CONTROL TO POINT OF
        INVOCATION. */
        RETURN;
        /* IF THIS POINT IS REACHED,
        ASSIGN A NULL VALUE TO EACH
        ELEMENT OF HEAD_ARRAY. */
NULL_LIST:
        HEAD_ARRAY = NULL;
END
        CON_PRA;
```

Figure 2.7B. The CON_PRA subroutine used to convert relocatable pointer lists from relocatable to absolute form

CON_LRA Subroutine

Figures 2.8A and 2.8B present the CON_LRA subroutine, which converts relocatable lists of lists to absolute form. The subroutine uses six arguments: the body area and head array of the relocatable lists of lists being converted, the body area and head array that are to receive the absolute lists of lists during conversion, the data area, which is shared by both the relocatable and absolute forms of the lists of lists, and the number of sublists.

The code in CON_LRA indicates an optional use of the recursive function procedure CON shown in the Appendix. CON examines the type code in each list component and takes appropriate conversion action. CON returns a pointer value.

CON_LRA Subroutine

Purpose

To convert lists of lists from relocatable to absolute form

Reference

```
CON_LRA(BODY_AREA1, OHEAD_ARRAY,
        BODY_AREA2, HEAD_ARRAY,
        DATA_AREA,#SUBS)
```

Entry-Name Declaration

```
DECLARE CON_LRA
    ENTRY(AREA(*), (*)OFFSET
        (DUMMY_BODY_AREA), AREA(*),
        (*)POINTER, AREA(*),FIXED
        DECIMAL);
```

Meaning of Arguments

BODY_AREA1	– the area that contains the bodies of the relocatable lists being converted to absolute form
OHEAD_ARRAY	– the array that contains the offset heads of the relocatable lists in BODY_AREA1
BODY_AREA2	– the area that receives the bodies of the lists after they have been converted to absolute form
HEAD_ARRAY	– the array that receives the pointer heads of the absolute lists in BODY_AREA2
DATA_AREA	– the area that contains the data values of the lists before and after conversion
#SUBS	– the number of sublists

Remarks

BODY_AREA1, BODY_AREA2, and DATA_AREA can have any storage class and be of arbitrary size. If BODY_AREA2 is not large enough to receive the converted components of BODY_AREA1, or if HEAD_ARRAY is smaller than OHEAD_ARRAY, or if all positions of OHEAD_ARRAY contain null offset values, then HEAD_ARRAY is filled with null pointer values, and the content of BODY_AREA2 becomes undefined.

Other Programmer-Defined Procedures Required

CON (optional)

Method

Each relocatable list in BODY_AREA1 is reconstructed, component by component, as an absolute list in BODY_AREA2. After conversion, both types of lists share DATA_AREA.

Figure 2.8A. Description of the CON_LRA subroutine for converting lists of lists from relocatable to absolute form

```
CON_LRA:
        PROCEDURE(BODY_AREA1,OHEAD_ARRAY,
        BODY_AREA2, HEAD_ARRAY, DATA_AREA,
        #SUBS);
 DECLARE
        #SUBS FIXED DECIMAL,
        (SAVE, KEEP, PA(#SUBS)) POINTER,
        /* PARAMETER #SUBS IS NOT NECESSARY
        WHEN FUNCTION CON  IS USED */
        (BODY_AREA1,BODY_AREA2,DATA_AREA)
        AREA(*),
        DUMMY_BODY_AREA
        BASED(DUMMY_BODY_POINTER) AREA,
        DUMMY_DATA_AREA
        BASED(DUMMY_DATA_POINTER) AREA,
        HEAD_ARRAY(*) POINTER,
        (DUMMY_BODY_POINTER,
        DUMMY_DATA_POINTER,C1,C2)POINTER,
        OHEAD_ARRAY(*)
        OFFSET(DUMMY_BODY_AREA),
        1 D_COMPONENT1 BASED(C1),
        2 D_OTYPE CHARACTER(1),
        2 D_OVALUE OFFSET(DUMMY_DATA_AREA),
        2 D_OLINK OFFSET(DUMMY_BODY_AREA),
        1 L_COMPONENT1 BASED(C1),
        2 L_OTYPE CHARACTER(1),
        2 L_OVALUE OFFSET(DUMMY_BODY_AREA),
        2 L_OLINK OFFSET(DUMMY_BODY_AREA),
        1 COMPONENT2 BASED(C2),
        2 TYPE CHARACTER(1),
        2 VALUE POINTER,
        2 LINK POINTER;
        /* IF AREA CONDITION OCCURS,
        BODY_AREA2 IS TOO SMALL TO RECEIVE
        CONTENTS OF BODY_AREA1. GO TO
        NULL_LIST. */
        ON AREA
      GO TO
        NULL_LIST;
        /* IF HEAD_ARRAY IS SMALLER THAN
        OHEAD_ARRAY, GO TO NULL_LIST. */
   IF
        DIM(HEAD_ARRAY,1)<DIM(OHEAD_ARRAY,1)
    THEN
      GO TO
        NULL_LIST;
        /* ASSOCIATE OFFSETS OHEAD_ARRAY,
        D_OLINK, L_OVALUE AND L_OLINK WITH
        BODY_AREA1, AND OFFSET D_OVALUE WITH
        DATA_AREA. */
        DUMMY_BODY_POINTER =
        ADDR(BODY_AREA1);
        DUMMY_DATA_POINTER =
        ADDR(DATA_AREA);
        /* CONVERT EACH RELOCATABLE LIST OF
        LISTS IN BODY_AREA1 TO AN ABSOLUTE
        LIST OF LISTS IN BODY_AREA2. */
        PA = NULL;
        J = LBOUND(HEAD_ARRAY,1)-1;
BEGIN_CONVERT_LOOP:
  DO
        I = LBOUND(OHEAD_ARRAY,1)
        TO HBOUND(OHEAD_ARRAY,1);
        J = J + 1;
        K = 1;
        IF OHEAD_ARRAY(I) = NULLO
        THEN DO;
        HEAD_ARRAY(J) = NULL;
        GO TO END_CONVERT_LOOP;
        END;
/* OPTION */
/* TEST */ IF #SUBS ¬= 1 THEN GO TO NO_CON;
    USE_CON:
        /* USE THE FOLLOWING CODE
        TO EMPLOY FUNCTION CON
        FOR CONVERSIONS */
        HEAD_ARRAY(J) = CON(OHEAD_ARRAY(I),
        BODY_AREA1, BODY_AREA2, DATA_AREA);
        GO TO END_CONVERT_LOOP;
/* END OF OPTION */
    NO_CON:
        C1 = OHEAD_ARRAY(I);
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE, KEEP, HEAD_ARRAY(J) = C2;
        C2->TYPE = 'L';
        PA(K) = C1 -> L_OVALUE;
        IF C1->L_OLINK = NULLO
        THEN C1 = NULL;
        ELSE C1 = C1->L_OLINK;
         DO WHILE(C1 ¬= NULL);
        K = K + 1;
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE->LINK = C2;
        SAVE = C2;
        C2->TYPE = 'L';
        PA(K) = C1 -> L_OVALUE;
        IF C1->L_OLINK = NULLO
        THEN C1 = NULL;
        ELSE C1 = C1->L_OLINK;
        END;
        SAVE->LINK = NULL;
  D_LIST:
        DO L = 1 TO #SUBS;
        C1 = PA(L);
        IF C1=NULL THEN GOTO END_D_LIST;
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE = C2;
        C2->TYPE = 'D';
        KEEP->VALUE = C2;
        KEEP = KEEP->LINK;
        IF C1->D_OVALUE = NULLO
        THEN C2->VALUE = NULL;
        ELSE C2->VALUE = C1->D_OVALUE;
        IF C1->D_OLINK = NULLO
        THEN C1 = NULL;
        ELSE C1 = C1->D_OLINK;
        DO WHILE (C1 ¬= NULL);
        ALLOCATE COMPONENT2 IN(BODY_AREA2)
        SET(C2);
        SAVE->LINK = C2;
        SAVE = C2;
        C2->TYPE = 'D';
        IF C1->D_OVALUE = NULLO
        THEN C2->VALUE = NULL;
        ELSE C2->VALUE = C1->D_OVALUE;
        IF C1->D_OLINK = NULLO
        THEN C1 = NULL;
        ELSE C1 = C1->D_OLINK;
        END;
        SAVE->LINK = NULL;
  END_D_LIST: END;
END_CONVERT_LOOP:
END;
        /* WHEN THIS POINT IS REACHED, ALL
        RELOCATABLE LISTS OF LISTS IN
        BODY_AREA1 HAVE BEEN CONVERTED TO
        ABSOLUTE LISTS OF LISTS IN
        BODY_AREA2. THEREFORE, RETURN
        SUBROUTINE CONTROL TO POINT OF
        INVOCATION. */
        RETURN;
        /* IF THIS POINT IS REACHED,
        ASSIGN A NULL VALUE TO EACH
        ELEMENT OF HEAD_ARRAY. */
```

```
NULL_LIST:
        HEAD_ARRAY = NULL;
END
        CON_LRA;
```

Figure 2.8B. The CON_LRA subroutine used to convert relocatable lists of lists from relocatable to absolute form

MOVING RELOCATABLE LISTS

The following discussions develop two subroutines for moving relocatable lists from one storage area to another:

1. MOVE_RDL, which moves relocatable data lists
2. MOVE_RPL, which moves either relocatable pointer lists or relocatable lists of lists. This subroutine can be used with either type of list because both types have a head, a body area, and a data area.

MOVE_RDL Subroutine

Figures 2.9A and 2.9B present the MOVE_RDL subroutine, which moves relocatable data lists from one area to another. The subroutine uses four arguments: the body area and head array of the source lists, and the body area and head array that are to receive the relocatable lists when they are moved.

MOVE_RDL Subroutine

Purpose

To move relocatable data lists

Reference

```
MOVE_RDL(BODY_AREA1,OHEAD_ARRAY1,
         BODY_AREA2,OHEAD_ARRAY2)
```

Entry-Name Declaration

```
DECLARE MOVE_RDL
    ENTRY(AREA(*),(*)OFFSET
        (DUMMY_BODY_AREA1),AREA(*),
        (*)OFFSET(DUMMY_BODY_AREA2));
```

Meaning of Arguments

BODY_AREA1 – the area that contains the relocatable lists being moved

OHEAD_ARRAY1 – the array that contains the offset heads of the relocatable lists in BODY_AREA1

BODY_AREA2 – the area to which the relocatable lists are moved

OHEAD_ARRAY2 – the array that receives the offset heads of the relocatable lists in BODY_AREA2

Remarks

BODY_AREA1 and BODY_AREA2 can have any storage class and be of arbitrary and unequal size. If BODY_AREA2 is not large enough to receive the contents of BODY_AREA1, or if OHEAD_ARRAY2 is smaller than OHEAD_ARRAY1, then OHEAD_ARRAY2 is set to NULLO, and the content of BODY_AREA2 becomes undefined.

Other Programmer Defined Procedures Required

None

Method

Assignment statements are used to move BODY_AREA1 to BODY_AREA2 and OHEAD_ARRAY1 to OHEAD_ARRAY2.

Figure 2.9A. Description of the MOVE_RDL subroutine for moving relocatable data lists

```
MOVE_RDL:
        PROCEDURE
        (BODY_AREA1, OHEAD_ARRAY1,
        BODY_AREA2, OHEAD_ARRAY2);
 DECLARE
        (DUMMY_POINTER1, DUMMY_POINTER2)
        POINTER,
        (BODY_AREA1,BODY_AREA2)AREA(*),
        DUMMY_BODY_AREA1 BASED
        (DUMMY_POINTER1) AREA,
        DUMMY_BODY_AREA2 BASED
        (DUMMY_POINTER2) AREA,
        OHEAD_ARRAY1(*) OFFSET
        (DUMMY_BODY_AREA1),
        OHEAD_ARRAY2(*) OFFSET
        (DUMMY_BODY_AREA2);
        /* IF AREA CONDITION OCCURS,
        BODY_AREA2 IS TOO SMALL TO RECEIVE
        CONTENTS OF BODY_AREA1.  SET
        OHEAD_ARRAY2 TO NULLO, AND GO TO END
        OF SUBROUTINE.*/
        ON AREA
 BEGIN;
        OHEAD_ARRAY2 = NULLO;
      GO TO
        END_MOVE_RDL;
END;
        /* ASSOCIATE OHEAD_ARRAY1 AND
        OHEAD_ARRAY2 WITH BODY_AREA1 AND
        BODY_AREA2.*/
        DUMMY_POINTER1 = ADDR(BODY_AREA1);
        DUMMY_POINTER2 = ADDR(BODY_AREA2);
        /* IF OHEAD_ARRAY2 IS SMALLER THAN
        OHEAD_ARRAY1, THEN SET OHEAD_ARRAY2
        TO NULLO, AND GO TO END OF
        SUBROUTINE, OTHERWISE, ASSIGN
        OHEAD_ARRAY1 TO OHEAD_ARRAY2.*/
   IF
        DIM(OHEAD_ARRAY2,1)<DIM(OHEAD_ARRAY1,1)
   THEN DO;
        OHEAD_ARRAY2 = NULLO;
      GO TO
        END_MOVE_RDL;
END;
    ELSE
        OHEAD_ARRAY2 = OHEAD_ARRAY1;
        /* ASSIGN BODY_AREA1 TO BODY_AREA2.
        AREA CONDITION MAY OCCUR.*/
        BODY_AREA2 = BODY_AREA1;
END_MOVE_RDL:
END
        MOVE_RDL;
```

Figure 2.9B. The MOVE_RDL subroutine used to move relocatable data lists from one area to another

MOVE_RPL Subroutine

Figures 2.10A and 2.10B present the MOVE_RPL subroutine, which moves relocatable pointer lists and relocatable lists of lists to new storage locations. The subroutine uses six arguments: the body area, head array, and data area of the source lists, and the body area, head array, and data area that are to receive the relocatable lists when they are moved.

MOVE_RPL Subroutine

Purpose

To move relocatable pointer lists and lists of lists

Reference

```
MOVE_RPL(BODY_AREA1, OHEAD_ARRAY1,
         DATA_AREA1, BODY_AREA2,
         OHEAD_ARRAY2, DATA_AREA2)
```

Entry-Name Declaration

```
DECLARE MOVE_RPL
    ENTRY(AREA(*), (*)OFFSET
         (DUMMY_BODY_AREA1),
         AREA(*), AREA(*),
         (*)OFFSET(DUMMY_BODY_AREA2),
         AREA(*));
```

Meaning of Arguments

BODY_AREA1 – the area that contains the bodies of the relocatable lists being moved

OHEAD_ARRAY1 – the array that contains the offset heads of the relocatable lists in BODY_AREA1

DATA_AREA1 – the area that contains the data values of the lists in BODY_AREA1

BODY_AREA2 – the area that receives the contents of BODY_AREA1

OHEAD_ARRAY2 – the array that receives the offset heads of the relocatable lists in BODY_AREA2

DATA_AREA2 – the area that receives the data values of the lists in BODY_AREA2

Remarks

BODY_AREA1, BODY_AREA2, DATA_AREA1, and DATA_AREA2 can have any storage class and be of arbitrary size. If BODY_AREA2 is not large enough to receive the contents of BODY_AREA1, or if DATA_AREA2 is not large enough to receive the contents of DATA_AREA1, or if OHEAD_ARRAY2 is smaller than OHEAD_ARRAY1, then OHEAD_ARRAY2 is set to NULLO, and the content of BODY_AREA2 and DATA_AREA2 becomes undefined.

Other Programmer-Defined Procedures Required

None

Method

Assignment statements are used to move BODY_AREA1 to BODY_AREA2, OHEAD_ARRAY1 to OHEAD_ARRAY2, and DATA_AREA1 to DATA_AREA2.

Figure 2.10A. Description of the MOVE_RPL subroutine for moving relocatable pointer lists and lists of lists

```
MOVE_RPL:
        PROCEDURE(BODY_AREA1,OHEAD_ARRAY1,
        DATA_AREA1, BODY_AREA2, OHEAD_ARRAY2,
        DATA_AREA2);
 DECLARE
        (BODY_AREA1,DATA_AREA1,BODY_AREA2,
        DATA_AREA2) AREA(*),
        DUMMY_BODY_AREA1
        BASED(DUMMY_POINTER1) AREA,
        DUMMY_BODY_AREA2
        BASED(DUMMY_POINTER2) AREA,
        OHEAD_ARRAY1(*)
        OFFSET(DUMMY_BODY_AREA1),
        OHEAD_ARRAY2(*)
        OFFSET(DUMMY_BODY_AREA2);
        /* IF AREA CONDITION OCCURS,
        BODY_AREA2 OR DATA_AREA2 IS TOO
        SMALL TO RECEIVE CONTENTS OF
        BODY_AREA1 OR DATA_AREA1. SET
        OHEAD_ARRAY2 TO NULLO, AND GO TO END
        OF SUBROUTINE. */
        ON AREA
 BEGIN;
        OHEAD_ARRAY2 = NULLO;
      GO TO
        END_MOVE_RPL;
END;
        /* ASSOCIATE OHEAD_ARRAY1 AND
        OHEAD_ARRAY2 WITH BODY_AREA1 AND
        BODY_AREA2. */
        DUMMY_POINTER1 = ADDR(BODY_AREA1);
        DUMMY_POINTER2 = ADDR(BODY_AREA2);
        /* IF OHEAD_ARRAY2 IS SMALLER THAN
        OHEAD_ARRAY1, THEN SET OHEAD_ARRAY2
        TO NULLO, AND GO TO END OF
        SUBROUTINE. OTHERWISE ASSIGN
        OHEAD_ARRAY1 TO OHEAD_ARRAY2. */
   IF
        DIM(OHEAD_ARRAY2,1)<
        DIM(OHEAD_ARRAY1,1)
    THEN
  DO;
        OHEAD_ARRAY2 = NULLO;
      GO TO
        END_MOVE_RPL;
END;
    ELSE
        OHEAD_ARRAY2 = OHEAD_ARRAY1;
        /* ASSIGN BODY_AREA1 TO BODY_AREA2
        AND DATA_AREA1 TO DATA_AREA2. AREA
        CONDITION MAY OCCUR. */
        BODY_AREA2 = BODY_AREA1;
        DATA_AREA2 = DATA_AREA1;
END_MOVE_RPL:
END
        MOVE_RPL;
```

Figure 2.10B. The MOVE_RPL subroutine used to move relocatable pointer lists and relocatable lists of lists to new storage locations

WRITING RELOCATABLE LISTS

The following discussions develop two subroutines for writing relocatable lists into a file:

1. WRITE_RDL, which writes relocatable data lists
2. WRITE_RPL, which writes either relocatable pointer lists or relocatable lists of lists. This subroutine can be used with either type of list because both types have a head, a body area, and a data area.

WRITE_RDL Subroutine

Figures 2.11A and 2.11B present the WRITE_RDL subroutine, which writes relocatable data lists into a file. The subroutine uses four arguments: the file that receives the lists, the head array and body area of the relocatable lists, and the size of the body area in bytes. The head array and body area are written as separate self-defining records in that order.

WRITE_RDL Subroutine

Purpose

To write relocatable data lists into a file

Reference

```
WRITE_RDL(DFILE, OHEAD_ARRAY,
          BODY_AREA, BODY_SIZE)
```

Entry-Name Declaration

```
DECLARE WRITE_RDL
     ENTRY(FILE RECORD OUTPUT,
          (*)OFFSET(DUMMY_BODY1),
          AREA(*), FIXED DECIMAL(5));
```

Meaning of Arguments

DFILE	– the file into which the relocatable lists are written
OHEAD_ARRAY	– the array that contains the offset heads of the relocatable lists
BODY_AREA	– the area that contains the bodies of the relocatable lists
BODY_SIZE	– the size of BODY_AREA in bytes

Remarks

DFILE must be a sequentially buffered output file. OHEAD_ARRAY and BODY_AREA can be of any storage class and have arbitrary size, and are written as separate logical records in that order. The records are self-defining: OHEAD_ARRAY is preceded by a count of its elements, and BODY_AREA is preceded by its storage size (BODY_SIZE), which does not include the control storage internally associated with an area.

Other Programmer-Defined Procedures Required

None

Method

Separate LOCATE statements are executed for each of the following record descriptions:

```
1 OHEAD_RECORD BASED(OUTPOINTER1),
  2 OUT_OHEAD_SIZE FIXED BINARY(16,0),
  2 OUT_OHEAD_ARRAY
     (BINARY_OHEAD_SIZE
     REFER(OUT_OHEAD_SIZE))OFFSET
     (DUMMY_BODY2),
1 BODY_RECORD BASED(OUTPOINTER2),
  2 PADDING2 CHARACTER(4),
  2 OUT_BODY_SIZE FIXED BINARY(16,0),
  2 OUT_BODY_AREA AREA
     (BINARY_BODY_SIZE
     REFER(OUT_BODY_SIZE)),
```

Figure 2.11A. Description of the WRITE_RDL subroutine for writing relocatable data lists into a file

```
WRITE_RDL:
        PROCEDURE(OUTFILE, OHEAD_ARRAY,
        BODY_AREA,BODY_SIZE);
 DECLARE
        (DUMMY_POINTER1, DUMMY_POINTER2,
        OUTPOINTER1, OUTPOINTER2)
        POINTER,
        DUMMY_BODY1 AREA
        BASED(DUMMY_POINTER1),
        DUMMY_BODY2 AREA
        BASED(DUMMY_POINTER2),
        OHEAD_ARRAY(*)
        OFFSET(DUMMY_BODY1),
        BODY_AREA AREA (*),
        BODY_SIZE FIXED DECIMAL(5),
        BINARY_OHEAD_SIZE FIXED BINARY(16,0),
        BINARY_BODY_SIZE FIXED BINARY( 16,0),
        OUTFILE FILE RECORD OUTPUT,
        1 OHEAD_RECORD BASED(OUTPOINTER1),
        2 OUT_OHEAD_SIZE FIXED BINARY( 16,0),
        2 OUT_OHEAD_ARRAY(BINARY_OHEAD_SIZE
        REFER(OUT_OHEAD_SIZE))
        OFFSET(DUMMY_BODY2),
        1 BCDY_RECORD BASED(OUTPOINTER2),
        2 PADDING2 CHARACTER(4),
        2 OUT_BODY_SIZE FIXED BINARY(16,0),
        2 OUT_BODY_AREA AREA
        (BINARY_BODY_SIZE
        REFER(OUT_BODY_SIZE));
        /* ASSOCIATE OHEAD_ARRAY AND
        OUT_OHEAD_ARRAY WITH BODY_AREA. */
        DUMMY_POINTER1,DUMMY_POINTER2 =
        ADDR(BODY_AREA);
        /* INITIALIZE SIZE OF
        OUT_OHEAD_ARRAY */
        BINARY_OHEAD_SIZE =
        DIM(OHEAD_ARRAY,1);
        /* INITIALIZE SIZE OF
        OUT_BODY_AREA */
        BINARY_BODY_SIZE = BODY_SIZE;
        /* LOCATE STORAGE IN OUTPUT BUFFER
        FOR OHEAD_RECORD, AND ASSIGN ADDRESS
        OF LOCATION TO OUTPOINTER1. */
        LOCATE OHEAD_RECORD FILE(OUTFILE)
        SET(OUTPOINTER1);
        /* ASSIGN OHEAD_ARRAY TO
        OUT_OHEAD_ARRAY IN OHEAD_RECORD. */
        OUTPOINTER1->OUT_OHEAD_ARRAY =
        OHEAD_ARRAY;
        /* LOCATE STORAGE IN OUTPUT BUFFER
        FOR BODY_RECORD, AND ASSIGN ADDRESS
        OF LOCATION TO OUTPOINTER2. */
        LOCATE BODY_RECORD FILE(OUTFILE)
        SET(OUTPOINTER2);
        /* ASSIGN BODY_AREA TO OUT_BODY_AREA
        IN BODY_RECORD. */
        OUTPOINTER2->OUT_BODY_AREA =
        BODY_AREA;
END
        WRITE RDL;
```

Figure 2.11B. The WRITE_RDL subroutine used to write relocatable data lists into a file

WRITE_RPL Subroutine

Figures 2.12A and 2.12B present the WRITE_RPL subroutine, which writes relocatable pointer lists and relocatable lists of lists into a file. The subroutine uses six arguments: the file that receives the relocatable lists, the head array of the lists, the containing body area and its size, and the associated data area and its size. The head array, the body area, and the data area are written as separate self-defining records in that order.

WRITE_RPL Subroutine

Purpose

To write relocatable pointer lists and lists of lists into a file

Reference

```
WRITE_RPL(LFILE, OHEAD_ARRAY,
          BODY_AREA, BODY_SIZE,
          DATA_AREA, DATA_SIZE)
```

Entry-Name Declaration

```
DECLARE WRITE_RPL
    ENTRY(FILE RECORD OUTPUT, (*)OFFSET
          (DUMMY_BODY1), AREA(*), FIXED
          DECIMAL(5), AREA(*), FIXED
          DECIMAL(5));
```

Meaning of Arguments

LFILE	– the file into which the relocatable lists are written
OHEAD_ARRAY	– the array that contains the offset heads of the relocatable lists
BODY_AREA	– the area that contains the bodies of the relocatable lists
BODY_SIZE	– the size of BODY_AREA in bytes
DATA_AREA	– the area that contains the data values of the relocatable lists
DATA_SIZE	– the size of DATA_AREA in bytes

Remarks

LFILE must be a sequentially buffered output file. OHEAD_ARRAY, BODY_AREA, and DATA_AREA can be of any storage class and have arbitrary size, and are written as separate logical records in that order. The records are self-defining: OHEAD_ARRAY is preceded by a count of its elements, and BODY_AREA and DATA_AREA are preceded by their storage sizes, which do not include the control storage internally associated with the areas.

Other Programmer-Defined Procedures Required

None

Method

Separate LOCATE statements are executed for each of the following record descriptions:

```
1 OHEAD_RECORD BASED(OUTPOINTER),
  2 OUT_OHEAD_SIZE FIXED BINARY(16,0),
  2 OUT_OHEAD_ARRAY
     (BINARY_OHEAD_SIZE
     REFER(OUT_OHEAD_SIZE)) OFFSET
     (DUMMY_BODY2),
1 BODY_RECORD BASED(OUTPOINTER),
  2 PADDING2 CHARACTER(4),
  2 OUT_BODY_SIZE FIXED BINARY(16,0),
  2 OUT_BODY_AREA AREA
     (BINARY_BODY_SIZE
     REFER(OUT_BODY_SIZE)),
1 DATA_RECORD BASED(OUTPOINTER),
  2 PADDING3 CHARACTER(4),
  2 OUT_DATA_SIZE FIXED BINARY(16,0),
  2 OUT_DATA_AREA AREA
     (BINARY_DATA_SIZE
     REFER(OUT_DATA_SIZE)),
```

Figure 2.12A. Description of the WRITE_RPL subroutine for writing relocatable pointer lists and lists of lists into a file

```
WRITE_RPL:
        PROCEDURE(OUTFILE,OHEAD_ARRAY,
        BODY_AREA,BODY_SIZE,DATA_AREA,
        DATA_SIZE);
 DECLARE
        (DUMMY_POINTER1, DUMMY_POINTER2,
        OUTPOINTER) POINTER,
        DUMMY_BODY1 AREA
        BASED(DUMMY_POINTER1),
        DUMMY_BODY2 AREA
        BASED(DUMMY_POINTER2),
        OHEAD_ARRAY(*) OFFSET(DUMMY_BODY1),
        BODY_AREA AREA (*),
        BODY_SIZE FIXED DECIMAL(5),
        DATA_AREA AREA (*),
        DATA_SIZE FIXED DECIMAL(5),
        OUTFILE FILE RECORD OUTPUT,
        BINARY_OHEAD_SIZE FIXED BINARY(16,0),
        BINARY_BODY_SIZE FIXED BINARY(16,0),
        BINARY_DATA_SIZE FIXED BINARY(16,0),
        1 OHEAD_RECORD BASED(OUTPOINTER),
        2 OUT_OHEAD_SIZE FIXED BINARY(16,0),
        2 OUT_OHEAD_ARRAY(BINARY_OHEAD_SIZE
        REFER(OUT_OHEAD_SIZE))
        OFFSET(DUMMY_BODY2),
        1 BODY_RECORD BASED(OUTPOINTER),
        2 PADDING2 CHARACTER(4),
        2 OUT_BODY_SIZE FIXED BINARY(16,0),
        2 OUT_BODY_AREA AREA
        (BINARY_BODY_SIZE REFER
        (OUT_BODY_SIZE)),
        1 DATA_RECORD BASED(OUTPOINTER),
        2 PADDING3 CHARACTER(4),
        2 OUT_DATA_SIZE FIXED BINARY(16,0),
        2 OUT_DATA_AREA AREA
        (BINARY_DATA_SIZE
        REFER(OUT_DATA_SIZE));
        /* ASSOCIATE OHEAD_ARRAY AND
        OUT_OHEAD_ARRAY WITH BODY_AREA. */
        DUMMY_POINTER1,DUMMY_POINTER2 =
        ADDR(BODY_AREA);
        /* INITIALIZE SIZE OF HEAD ARRAY IN
        OHEAD_RECORD, SIZE OF BODY AREA IN
        BODY_RECORD, AND SIZE OF DATA
        AREA IN DATA_RECORD. */
        BINARY_OHEAD_SIZE =
        DIM(OHEAD_ARRAY,1);
        BINARY_BODY_SIZE = BODY_SIZE;
        BINARY_DATA_SIZE = DATA_SIZE;
        /* LOCATE STORAGE IN OUTPUT BUFFER
        FOR OHEAD_RECORD, AND ASSIGN
        OHEAD_ARRAY TO OUT_OHEAD_ARRAY. */
        LOCATE OHEAD_RECORD FILE(OUTFILE)
        SET(OUTPOINTER);
        OUTPOINTER->OUT_OHEAD_ARRAY =
        OHEAD_ARRAY;
        /* LOCATE STORAGE IN OUTPUT BUFFER
        FOR BODY_RECORD, AND ASSIGN
        BODY_AREA TO OUT_BODY_AREA. */
        LOCATE BODY_RECORD FILE(OUTFILE)
        SET(OUTPOINTER);
        OUTPOINTER->OUT_BODY_AREA =
        BODY_AREA;
        /* LOCATE STORAGE IN OUTPUT BUFFER
        FOR DATA_RECORD, AND ASSIGN DATA
        AREA TO OUT_DATA_AREA. */
        LOCATE DATA_RECORD FILE(OUTFILE)
        SET(OUTPOINTER);
        OUTPOINTER->OUT_DATA_AREA =
        DATA_AREA;
 END
        WRITE_RPL;
```

Figure 2.12B. The WRITE_RPL subroutine used to write relocatable pointer lists and relocatable lists of lists into a file

READING RELOCATABLE LISTS

The following discussions develop two subroutines for reading relocatable lists from a file:

1. READ_RDL, which reads relocatable data lists
2. READ_RPL, which reads either relocatable pointer lists or relocatable lists of lists. This subroutine can be used with either type of list because both types have a head, a body area, and a data area.

READ_RDL Subroutine

Figures 2.13A and 2.13B present the READ_RDL subroutine, which reads relocatable data lists from a file. The subroutine uses four arguments: the file that contains the relocatable lists, the head array and body area that are to receive the lists, and a variable that receives the size of the body area. The head array and body area are assumed to be contained in separate self-defining records, which are read in that order.

READ_RDL Subroutine

Purpose

To read relocatable data lists from a file

Reference

```
READ_RDL(DFILE, OHEAD_ARRAY,
         BODY_AREA, BODY_SIZE)
```

Entry-Name Declaration

```
DECLARE READ_RDL
    ENTRY(FILE RECORD INPUT, (*)OFFSET
          (DUMMY_BODY1),AREA(*), FIXED
          DECIMAL(5));
```

Meaning of Arguments

DFILE	–	the file from which the relocatable data lists are read
OHEAD_ARRAY	–	the array that receives the offset heads of the relocatable lists
BODY_AREA	–	the area that receives the bodies of the relocatable lists
BODY_SIZE	–	the size of BODY_AREA in bytes

Remarks

DFILE must be a sequentially buffered input file. OHEAD_ARRAY and BODY_AREA can be of any storage class and have arbitrary size; their values are read as separate logical records in that order. The records are self-defining: the record for OHEAD_ARRAY is preceded by a count of its offset values, and the record for BODY_AREA is preceded by the size of the area in bytes. The size of BODY_AREA is assigned to BODY_SIZE. An attempt to read past the end of DFILE assigns a zero value to BODY_SIZE and returns control to the invoking procedure.

Other Programmer-Defined Procedures Required

None

Method

Separate READ statements are executed for each of the following record descriptions:

```
1 OHEAD_RECORD BASED(INPOINTER),
   2 IN_OHEAD_SIZE FIXED BINARY(16,0),
   2 IN_OHEAD_ARRAY
      (BINARY_OHEAD_SIZE
      REFER(IN_OHEAD_SIZE))
      OFFSET(DUMMY_BODY2),
1 BODY_RECORD BASED(INPOINTER),
   2 PADDING2 CHARACTER(4),
   2 IN_BODY_SIZE FIXED BINARY(16,0),
   2 IN_BODY_AREA AREA
      (BINARY_BODY_SIZE
      REFER(IN_BODY_SIZE)),
```

Figure 2.13A. Description of the READ_RDL subroutine for reading relocatable data lists from a file

```
READ_RDL:
        PROCEDURE(INFILE, OHEAD_ARRAY,
        BODY_AREA,BODY_SIZE);
 DECLARE
        (DUMMY_POINTER1, DUMMY_POINTER2,
        INPOINTER) POINTER,
        DUMMY_BODY1 AREA
        BASED(DUMMY_POINTER1),
        DUMMY_BODY2 AREA
        BASED(DUMMY_POINTER2),
        OHEAD_ARRAY(*) OFFSET(DUMMY_BODY1),
        BODY_SIZE FIXED DECIMAL(5),
        BODY_AREA AREA (*),
        INFILE FILE RECORD INPUT,
        BINARY_OHEAD_SIZE FIXED BINARY(16,0),
        BINARY_BODY_SIZE FIXED BINARY(16,0),
        1 OHEAD_RECORD BASED(INPOINTER),
        2 IN_OHEAD_SIZE FIXED BINARY(16,0),
        2 IN_OHEAD_ARRAY(BINARY_OHEAD_SIZE
        REFER(IN_OHEAD_SIZE))
        OFFSET(DUMMY_BODY2),
        1 BODY_RECORD BASED(INPOINTER),
        2 PADDING2 CHARACTER(4),
        2 IN_BODY_SIZE FIXED BINARY(16,0),
        2 IN_BODY_AREA AREA
        (BINARY_BODY_SIZE
        REFER(IN_BODY_SIZE));
        /* AT END OF INFILE, SET BODY_SIZE TO
        ZERO, AND END SUBROUTINE. */
        ON ENDFILE(INFILE)
 BEGIN;
        BODY_SIZE = 0;
      GO TO
        END_READ_RDL;
END;
        /* ASSOCIATE OHEAD_ARRAY AND
        IN_OHEAD_ARRAY WITH BODY_AREA. */
        DUMMY_POINTER1,DUMMY_POINTER2 =
        ADDR(BODY_AREA);
        /* READ NEXT LOGICAL OHEAD_RECORD
        FROM INFILE, AND SET INPOINTER TO
        LOCATION OF OHEAD_RECORD IN INPUT
        BUFFER. */
        READ FILE(INFILE) SET(INPOINTER);
        /* ASSIGN IN_OHEAD_ARRAY WITHIN
        OHEAD_RECORD TO OHEAD_ARRAY. */
        OHEAD_ARRAY =
        INPOINTER->IN_OHEAD_ARRAY;
        /* READ NEXT LOGICAL BODY_RECORD
        FROM INFILE, AND SET INPOINTER TO
        LOCATION OF BODY_RECORD IN INPUT
        BUFFER. */
        READ FILE(INFILE) SET(INPOINTER);
        /* ASSIGN IN_BODY_AREA WITHIN
        BODY_RECORD TO BODY_AREA. */
        BODY_AREA =
        INPOINTER->IN_BODY_AREA;
END_READ_RDL:
END
        READ_RDL;
```

Figure 2.13B. The READ_RDL subroutine used to read relocatable data lists from a file

READ_RPL Subroutine

Figures 2.14A and 2.14B present the READ_RPL subroutine, which reads relocatable pointer lists and relocatable lists of lists from a file. The subroutine uses six arguments: the file that contains the relocatable lists, an array to receive the heads of the lists, an area and a variable to receive the body and the body size of the lists, and an area and a variable to receive the data area and the data-area size of the lists. The head array, the body area, and the data area are assumed to be contained in separate self-defining records, which are read in that order.

READ_RPL Subroutine

Purpose

To read relocatable pointer lists and lists of lists from a file

Reference

```
READ_RPL(LFILE, OHEAD_ARRAY,
         BODY_AREA, BODY_SIZE,
         DATA_AREA, DATA_SIZE)
```

Entry-Name Declaration

```
DECLARE READ_RPL
    ENTRY(FILE RECORD INPUT, (*)OFFSET
         (DUMMY_BODY1), AREA(*), FIXED
         DECIMAL(5), AREA(*), FIXED
         DECIMAL(5));
```

Meaning of Arguments

LFILE	–	the file from which the relocatable lists are read
OHEAD_ARRAY	–	the array that receives the offset heads of the relocatable lists
BODY_AREA	–	the area that receives the bodies of the relocatable lists
BODY_SIZE	–	the size of BODY_AREA in bytes
DATA_AREA	–	the area that receives the data values of the relocatable lists
DATA_SIZE	–	the size of DATA_AREA in bytes

Remarks

LFILE must be a sequentially buffered input file. OHEAD_ARRAY, BODY_AREA, and DATA_AREA can be of any storage class and have arbitrary size; their values are read as separate logical records in that order. The records are self-defining: the record for OHEAD_ARRAY is preceded by a count of its offset values, and the records for BODY_AREA and DATA_AREA are preceded by their storage sizes. The size of BODY_AREA is assigned to BODY_SIZE, and DATA_SIZE receives the size of DATA_AREA. An attempt to read past the end of LFILE assigns zero values to BODY_SIZE and DATA_SIZE and returns control to the invoking procedure.

Other Programmer-Defined Procedures Required

None

Method

Separate READ statements are executed for each of the following record descriptions:

```
1 OHEAD_RECORD BASED(INPOINTER),
  2 IN_OHEAD_SIZE FIXED BINARY(16,0),
  2 IN_OHEAD_ARRAY
     (BINARY_OHEAD_SIZE
     REFER(IN_OHEAD_SIZE)) OFFSET
     (DUMMY_BODY2),
1 BODY_RECORD BASED(INPOINTER),
  2 PADDING2 CHARACTER(4),
  2 IN_BODY_SIZE FIXED BINARY(16,0),
  2 IN_BODY_AREA AREA
     (BINARY_BODY_SIZE
     REFER(IN_BODY_SIZE)),
1 DATA_RECORD BASED(INPOINTER),
  2 PADDING3 CHARACTER(4),
  2 IN_DATA_SIZE FIXED BINARY(16,0),
  2 IN_DATA_AREA AREA
     (BINARY_DATA_SIZE
     REFER(IN_DATA_SIZE)),
```

Figure 2.14A. Description of the READ_RPL subroutine for reading relocatable pointer lists and lists of lists from a file.

```
READ_RPL:
        PROCEDURE(INFILE,OHEAD_ARRAY,
        BODY_AREA,BODY_SIZE,DATA_AREA,
        DATA_SIZE);
 DECLARE
        (DUMMY_POINTER1, DUMMY_POINTER2,
        INPOINTER) POINTER,
        DUMMY_BODY1 AREA
        BASED(DUMMY_POINTER1),
        DUMMY_BODY2 AREA
        BASED(DUMMY_POINTER2),
        OHEAD_ARRAY(*)
        OFFSET(DUMMY_BODY1),
        BODY_AREA AREA (*),
        BODY_SIZE FIXED DECIMAL(5),
        DATA_AREA AREA (*),
        DATA_SIZE FIXED DECIMAL(5),
        INFILE FILE RECORD INPUT,
        BINARY_OHEAD_SIZE FIXED BINARY(16,0),
        BINARY_BODY_SIZE FIXED BINARY(16,0),
        BINARY_DATA_SIZE FIXED BINARY(16,0),
        1 OHEAD_RECORD BASED(INPOINTER),
        2 IN_OHEAD_SIZE FIXED BINARY(16,0),
        2 IN_OHEAD_ARRAY(BINARY_OHEAD_SIZE
        REFER(IN_OHEAD_SIZE))
        OFFSET(DUMMY_BODY2),
        1 BODY_RECORD BASED(INPOINTER),
        2 PADDING2 CHARACTER(4),
        2 IN_BODY_SIZE FIXED BINARY(16,0),
        2 IN_BODY_AREA AREA(BINARY_BODY_SIZE
        REFER(IN_BODY_SIZE)),
        1 DATA_RECORD BASED(INPOINTER),
        2 PADDING3 CHARACTER(4),
        2 IN_DATA_SIZE FIXED BINARY(16,0),
        2 IN_DATA_AREA AREA(BINARY_DATA_SIZE
        REFER(IN_DATA_SIZE));
        /* AT END OF INFILE, SET BODY_SIZE
        AND DATA_SIZE TO ZERO, AND END
        SUBROUTINE. */
        ON ENDFILE(INFILE)
 BEGIN;
        BODY_SIZE,DATA_SIZE = 0;
    GO TO
        END_READ_RPL;
END;
        /* ASSOCIATE OHEAD_ARRAY AND
        IN_OHEAD_ARRAY WITH BODY_AREA. */
        DUMMY_POINTER1,DUMMY_POINTER2 =
        ADDR(BODY_AREA);
        /* READ NEXT LOGICAL OHEAD_RECORD
        FROM INFILE. */
        READ FILE(INFILE) SET(INPOINTER);
        /* ASSIGN IN_OHEAD_ARRAY WITHIN
        OHEAD_RECORD TO OHEAD_ARRAY. */
        OHEAD_ARRAY =
        INPOINTER->IN_OHEAD_ARRAY;
        /* READ NEXT LOGICAL BODY_RECORD
        FROM INFILE. */
        READ FILE(INFILE) SET(INPOINTER);
        /* ASSIGN IN_BODY_AREA WITHIN
        BODY_RECORD TO BODY_AREA. */
        BODY_AREA =
        INPOINTER->IN_BODY_AREA;
        /* READ NEXT LOGICAL DATA_RECORD
        FROM INFILE. */
        READ FILE(INFILE) SET(INPOINTER);
        /* ASSIGN IN_DATA_AREA WITHIN
        DATA_RECORD TO DATA_AREA. */
        DATA_AREA =
        INPOINTER->IN_DATA_AREA;
END_READ_RPL:
END
        READ_RPL;
```

Figure 2.14B. The READ_RPL subroutine used to read relocatable pointer lists and relocatable lists of lists from a file

Organizing a list in relocatable form permits the list to be stored in a file. Transmission of relocatable lists to and from files allows programs to be run in stages and also allows libraries of list organizations to be created and maintained for use by other programs.

The following discussions present two examples of list transmission. The first example processes relocatable data lists, and the second processes relocatable lists of lists. Both examples use the previously developed subroutines for converting, writing, and reading relocatable lists.

AN EXAMPLE THAT TRANSMITS RELOCATABLE DATA LISTS

Figures 3.1A through 3.1F present the TRANS_D program, which provides a simple illustration of how relocatable data lists may be constructed and then transmitted to and from a file. The program performs little actual processing of the lists and concentrates mainly on showing how relocatable data lists can be written into and read from a file.

TRANS_D begins by allocating list components in the storage area ABSOLUTE_BODY_AREA and linking the components into an absolute list of available storage components called FREE. Each component contains a single character as its data element. The program then uses the components in FREE to create two additional lists, LIST1 and LIST2, which contain ten components each.

```
TRANS_D:
PROCEDURE;
DECLARE
        (LIST1, LIST2, FREE, WORK_POINTER,
        DP, P) POINTER,
        ABSOLUTE_HEAD_ARRAY(2) POINTER,
        RELOCATABLE_HEAD_ARRAY(2)
        OFFSET(DUMMY_RELOCATABLE_AREA),
        ABSOLUTE_BODY_AREA AREA(240),
        RELOCATABLE_BODY_AREA AREA (240),
        BODY_SIZE FIXED DECIMAL(5)
        INITIAL (240),
        DUMMY_RELOCATABLE_AREA
        AREA BASED(DP),
        DFILE FILE RECORD /* RECFM = V */,
        1 LIST_COMPONENT BASED(P),
        2 DATA CHARACTER(1),
        2 LINK POINTER,
        BLANKS CHARACTER(70);
        /* WHEN FREE LIST HAS BEEN FORMED,
        GO TO NULL_LINK. */
        ON AREA
      GO TO
        NULL_LINK;
```

```
    ON ENDFILE (SYSIN)
      BEGIN;
      CLOSE FILE(DFILE);
      OPEN FILE(DFILE) INPUT;
      GO TO OUTPUT;
      END;
START:
        /* INITIALIZE. */
        ABSOLUTE_HEAD_ARRAY = NULL;
        RELOCATABLE_HEAD_ARRAY = NULLO;
        ABSOLUTE_BODY_AREA,
        RELOCATABLE_BODY_AREA = EMPTY;
        /* ASSOCIATE RELOCATABLE_HEAD_ARRAY
        WITH RELOCATABLE_BODY_AREA. */
        DP = ADDR(RELOCATABLE_BODY_AREA);
        /* FORM ABSOLUTE LIST OF FREE
        STORAGE COMPONENTS, AND SET DATA
        ELEMENT OF EACH COMPONENT TO
        BLANK. */
        ALLOCATE LIST_COMPONENT
        IN(ABSOLUTE_BODY_AREA) SET(FREE);
        WORK_POINTER = FREE;
REPEAT:
        ALLOCATE LIST_COMPONENT
        IN(ABSOLUTE_BODY_AREA) SET(P);
        WORK_POINTER->LINK = P;
        WORK_POINTER->DATA = ' ';
        WORK_POINTER = P;
    GO TO
        REPEAT;
NULL_LINK:
        /* IN LAST COMPONENT OF FREE LIST,
        SET LINK POINTER TO NULL AND DATA
        ELEMENT TO BLANK. */
        WORK_POINTER->LINK = NULL;
        WORK_POINTER->DATA = ' ';
        /* FORM ABSOLUTE LISTS, LIST1 AND
        LIST2, WITH TEN COMPONENTS EACH
        FROM FREE LIST. */
        LIST1, WORK_POINTER = FREE;
  DO
        I = 1 TO 9;
        WORK_POINTER =
        WORK_POINTER->LINK;
END;
        LIST2 = WORK_POINTER->LINK;
        WORK_POINTER->LINK = NULL;
        WORK_POINTER = LIST2;
  DO
        I = 1 TO 9;
        WORK_POINTER = WORK_POINTER->LINK;
END;
        FREE = WORK_POINTER->LINK;
        WORK_POINTER->LINK = NULL;
        /* ASSIGN HEAD POINTERS LIST1
        AND LIST2 TO ABSOLUTE_HEAD_ARRAY */
        ABSOLUTE_HEAD_ARRAY(1) = LIST1;
        ABSOLUTE_HEAD_ARRAY(2) = LIST2;
        OPEN FILE(DFILE) OUTPUT;
INPUT:
        /* READ TWO INPUT CARDS AND INSERT
        THE FIRST TEN COLUMNS OF FIRST CARD
        INTO LIST1 AND THE FIRST TEN
        COLUMNS OF SECOND CARD INTO LIST2.*/
  DO
```

```
        WORK_POINTER = LIST1,LIST2;
  DO
        I = 1 TO 10;
     GET
        EDIT(WORK_POINTER->DATA)(A(1));
        WORK_POINTER = WORK_POINTER->LINK;
END;
     GET EDIT(BLANKS)(A(70));
END;
        /* EMPTY RELOCATABLE_BODY_AREA. */
        RELOCATABLE_BODY_AREA = EMPTY;
        /* CONVERT DATA LISTS LIST1
        AND LIST2 FROM ABSOLUTE TO
        RELOCATABLE FORM. */
        CALL CON_DAR(ABSOLUTE_BODY_AREA,
        ABSOLUTE_HEAD_ARRAY,
        RELOCATABLE_BODY_AREA,
        RELOCATABLE_HEAD_ARRAY);
        /* WRITE RELOCATABLE DATA LISTS */
        CALL WRITE_RDL(DFILE,
        RELOCATABLE_HEAD_ARRAY,
        RELOCATABLE_BODY_AREA, BODY_SIZE);
        /* PROCESS NEXT TWO INPUT CARDS. */
      GO TO
        INPUT;
OUTPUT:
        /* READ HEAD ARRAY AND BODY AREA FOR
        NEXT SET OF RELOCATABLE LISTS */
        CALL READ_RDL(DFILE,
        RELOCATABLE_HEAD_ARRAY,
        RELOCATABLE_BODY_AREA,BODY_SIZE);
   IF
        BODY_SIZE = 0
    THEN
      GO TO
        END_TRANS_D;
        /* EMPTY ABSOLUTE_BODY_AREA. */
        ABSOLUTE_BODY_AREA = EMPTY;
        /* CONVERT DATA LISTS FROM
        RELOCATABLE TO ABSOLUTE FORM. */
        CALL CON_DRA(RELOCATABLE_BODY_AREA,
        RELOCATABLE_HEAD_ARRAY,
        ABSOLUTE_BODY_AREA,
        ABSOLUTE_HEAD_ARRAY);
        /* ASSIGN POINTER VALUES OF
        ABSOLUTE_HEAD_ARRAY TO LIST1
        AND LIST2. */
        LIST1 = ABSOLUTE_HEAD_ARRAY(1);
        LIST2 = ABSOLUTE_HEAD_ARRAY(2);
        /* PRINT DATA VALUES IN LIST1 IN
        SUCCESSIVE POSITIONS ON ONE LINE
        AND THOSE OF LIST2 ON NEXT LINE. */
PRINT:
  DO
        WORK_POINTER= LIST1,LIST2;
  DO
        I = 1 TO 10;
     PUT
        EDIT(WORK_POINTER->DATA)(A(1));
        WORK_POINTER = WORK_POINTER->LINK;
END;
     PUT SKIP;
END_PRINT:END;
     PUT
        LIST('*****');
     PUT
        SKIP(2);
        /* PROCESS NEXT SET OF
        RELOCATABLE LISTS IN DFILE. */
      GO TO
        OUTPUT;
        END_TRANS_D:
        CLOSE FILE(SYSPRINT);
        CLOSE FILE(DFILE);
END
        TRANS_D;
```

Figure 3.1A. The TRANS_D program, which illustrates the construction of a relocatable data list and its transmission to and from a file

LIST1 and LIST2 obtain their data values from the standard system-input file, SYSIN. LIST1 receives the characters in cc 1 through 10 of the first input card. Similarly, the second input card supplies the data values for LIST2. Sample input cards appear in Figure 3.1B, and Figure 3.1C shows how the characters from the first two input cards are arranged in LIST1 and LIST2.

```
CARD1 -----
CARD2 -----
CARD3 -----
CARD4 -----
CARD5 -----
CARD6 -----
CARD7 -----
CARD8 -----
CARD9 -----
CARD10 ----
```

Figure 3.1B. Sample input from SYSIN file

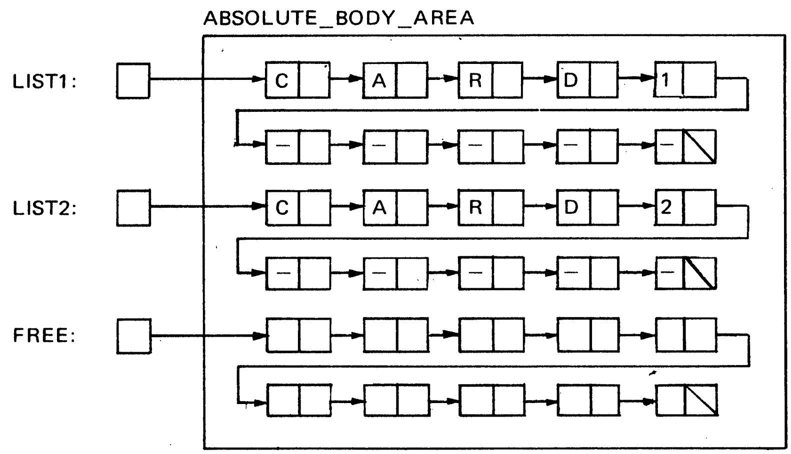


Figure 3.1C. Examples of absolute data lists

Next, the program assigns the head pointers of LIST1 and LIST2 to the pointer array ABSOLUTE_HEAD_ARRAY and invokes the subroutine CON_DAR, which was discussed in Chapter 2. This subroutine converts the absolute lists LIST1 and LIST2 in ABSOLUTE_BODY_AREA to relocatable data lists in RELOCATABLE_BODY_AREA, as illustrated by Figure 3.1D. The array RELOCATABLE_HEAD_ARRAY contains the offset heads of the relocatable lists. At this point, subroutine WRITE_RDL writes the relocatable lists into the file, DFILE, as shown in Figure 3.1E.

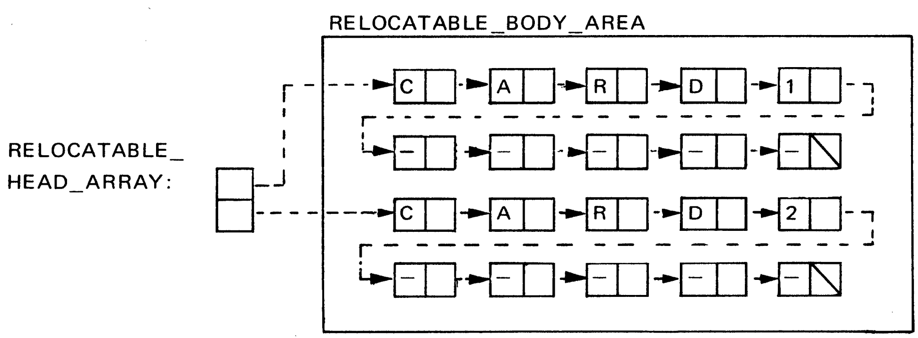


Figure 3.1D. Examples of relocatable data lists

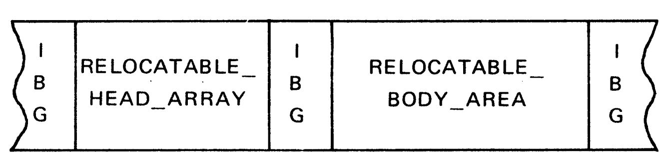


Figure 3.1E. Sample content of DFILE (unblocked)

TRANS_D continues processing pairs of input cards and generating relocatable output for DFILE. When the end of the SYSIN file is reached, DFILE is closed and reopened as an input file. The previous processing steps are now reversed. Subroutine READ_RDL retrieves each relocatable head array and body area from DFILE, and subroutine CON_DRA converts the retrieved lists from relocatable to absolute form. The data values of absolute LIST 1 are then printed in successive positions on a line in the standard system-output file, SYSPRINT. Similarly, the data values of LIST2 appear on the next line, as shown in Figure 3.1F. Each pair of output lines is followed by a line of five asterisks and a blank line. The program terminates when the end of DFILE is reached.

```
CARD1 - - - - -
CARD2 - - - - -
* * * * *

CARD3 - - - - -
CARD4 - - - - -
* * * * *

CARD5 - - - - -
CARD6 - - - - -
* * * * *

CARD7 - - - - -
CARD8 - - - - -
* * * * *

CARD9 - - - - -
CARD10  - - - -
* * * * *
```

Figure 3.1F. Sample output to SYSPRINT file

AN EXAMPLE THAT TRANSMITS RELOCATABLE LISTS OF LISTS

Figures 3.2A through 3.2F present the TRANS_L program, which illustrates how relocatable lists of lists may be transmitted to and from a file. This program resembles the previous program, TRANS_D, except that it processes relocatable lists of lists.

```
TRANS_L:
PROCEDURE;
 DECLARE
        (LIST1,LIST2,FREE,WORK_POINTER,
        SUB1, SUB2, DB, P, D) POINTER,
        ABSOLUTE_HEAD_ARRAY(2) POINTER,
        RELOCATABLE_HEAD_ARRAY(2)
        OFFSET(DUMMY_RELOCATABLE_BODY),
        (ABSOLUTE_BODY_AREA,
        RELOCATABLE_BODY_AREA,
        DATA_AREA) AREA (400),
        DATA_SIZE FIXED DECIMAL(5)
        INITIAL (400),
        BODY_SIZE FIXED DECIMAL(5)
        INITIAL (400),
        DUMMY_RELOCATABLE_BODY
        AREA BASED(DB),
        1 LIST_COMPONENT BASED(P),
        2 TYPE CHARACTER(1),
        2 VALUE POINTER,
        2 LINK POINTER,
        DATA_ITEM CHARACTER(1) BASED(D),
        LFILE FILE RECORD /* RECFM = V */,
        COUNT FIXED DECIMAL INIT(0),
        #SUBS FIXED DECIMAL INIT(2),
        BLANKS CHARACTER (80),
        /* WHEN FREE LIST HAS BEEN FORMED,
        GO TO NULL_LINK. */
        ON AREA
      GO TO
        NULL_LINK;
        /* WHEN ALL INPUT CARDS HAVE BEEN
        READ FROM SYSIN FILE, CLOSE LFILE
        AND REOPEN IT AS AN INPUT FILE.
        THEN GO TO OUTPUT. */
      ON ENDFILE (SYSIN)
 BEGIN;
 CLOSE FILE (
        LFILE);
 OPEN FILE (
        LFILE ) INPUT;
      GO TO
        OUTPUT;
END;
START:
        /* INITIALIZE. */
        ABSOLUTE_HEAD_ARRAY = NULL;
        RELOCATABLE_HEAD_ARRAY = NULLO;
        ABSOLUTE_BODY_AREA = EMPTY;
        RELOCATABLE_BODY_AREA = EMPTY;
        /* ASSOCIATE RELOCATABLE_HEAD_ARRAY
        WITH RELOCATABLE_BODY_AREA. */
        DB = ADDR(RELOCATABLE_BODY_AREA);
        /* FORM ABSOLUTE LIST OF FREE
        STORAGE COMPONENTS. IN EACH
        COMPONENT, SET TYPE CODE TO 'D' AND
        VALUE POINTER TO NULL. */
        ALLOCATE LIST_COMPONENT
        IN(ABSOLUTE_BODY_AREA) SET(FREE);
        WORK_POINTER = FREE;
REPEAT:
        ALLOCATE LIST_COMPONENT
        IN(ABSOLUTE_BODY_AREA) SET(P);
        WORK_POINTER->LINK = P;
        WORK_POINTER->VALUE = NULL;
        WORK_POINTER->TYPE = 'D';
```

```
        WORK_POINTER = P;
      GO TO
        REPEAT;
NULL_LINK:
        /* IN LAST COMPONENT OF FREE LIST,
        SET LINK ELEMENT TO NULL, VALUE
        ELEMENT TO NULL, AND TYPE CODE TO
        'D'.*/
        WORK_POINTER->LINK = NULL;
        WORK_POINTER->VALUE = NULL;
        WORK_POINTER->TYPE = 'D';
        /* FORM ABSOLUTE LISTS OF LISTS,
        LIST1 AND LIST2, FROM FREE LIST BY
        ASSIGNING 12 LIST COMPONENTS AT THE
        TOP LEVEL OF EACH LIST. */
        LIST1 = FREE;
        WORK_POINTER = LIST1;
  DO
        I = 1 TO 11;
        WORK_POINTER = WORK_POINTER->LINK;
END;
        LIST2 = WORK_POINTER->LINK;
        WORK_POINTER->LINK = NULL;
        WORK_POINTER = LIST2;
  DO
        I = 1 TO 11;
        WORK_POINTER = WORK_POINTER->LINK;
END;
        FREE = WORK_POINTER->LINK;
        WORK_POINTER->LINK = NULL;
        /* ASSIGN HEAD POINTERS LIST1
        AND LIST2 TO ABSOLUTE_HEAD_ARRAY */
        ABSOLUTE_HEAD_ARRAY(1) = LIST1;
        ABSOLUTE_HEAD_ARRAY(2) = LIST2;
        /* ORGANIZE THE 12 COMPONENTS IN
        LIST1 SO THAT THE TOP LEVEL
        CONTAINS TWO SUBLISTS WITH FIVE
        COMPONENTS EACH. DO THE SAME FOR
        LIST2. */
  DO
        I = 1 TO 2;
        WORK_POINTER =
        ABSOLUTE_HEAD_ARRAY(I);
        WORK_POINTER->TYPE = 'L';
        WORK_POINTER = WORK_POINTER->LINK;
        WORK_POINTER->TYPE = 'L';
        SUB1 = WORK_POINTER->LINK;
        WORK_POINTER->LINK = NULL;
        WORK_POINTER = SUB1;
  DO
        J = 1 TO 4;
        WORK_POINTER = WORK_POINTER->LINK;
END;
        SUB2 = WORK_POINTER->LINK;
        WORK_POINTER->LINK = NULL;
        WORK_POINTER = SUB2;
  DO
        J = 1 TO 4;
        WORK_POINTER = WORK_POINTER->LINK;
END;
        WORK_POINTER->LINK = NULL;
        WORK_POINTER =
        ABSOLUTE_HEAD_ARRAY(I);
        WORK_POINTER->VALUE = SUB1;
        WORK_POINTER =
        WORK_POINTER->LINK;
        WORK_POINTER->VALUE = SUB2;
END;
        /* OPEN LFILE  AS OUTPUT FILE. */
 OPEN FILE (
        LFILE ) OUTPUT;
INPUT:
        DATA_AREA = EMPTY;
        DO WORK_POINTER = LIST1, LIST2;
        SUB1 = WORK_POINTER->VALUE;
        WORK_POINTER = WORK_POINTER->LINK;
        SUB2 = WORK_POINTER->VALUE;
        DO WHILE(SUB1 ¬= NULL);
        ALLOCATE DATA_ITEM
        IN(DATA_AREA) SET(D);
        GET EDIT(D->DATA_ITEM)(A(1));
        SUB1->VALUE = D;
        SUB1 = SUB1->LINK;
        COUNT = COUNT + 1;
END;
        DO WHILE(SUB2 ¬= NULL);
        ALLOCATE DATA_ITEM
        IN(DATA_AREA) SET(D);
        GET EDIT(D->DATA_ITEM)(A(1));
        SUB2->VALUE = D;
        SUB2 = SUB2->LINK;
        COUNT = COUNT + 1;
END;
        GET EDIT (BLANKS) (A(80 - COUNT));
        COUNT = 0;
 END;
        /* EMPTY RELOCATABLE_BODY_AREA. */
        RELOCATABLE_BODY_AREA = EMPTY;
        /* CONVERT ABSOLUTE LISTS OF
        LISTS (LIST1 AND LIST2) TO
        RELOCATABLE FORM. */
        CALL CON_LAR(ABSOLUTE_BODY_AREA,
        ABSOLUTE_HEAD_ARRAY,
        RELOCATABLE_BODY_AREA,
        RELOCATABLE_HEAD_ARRAY,
        DATA_AREA, #SUBS);
        /* WRITE RELOCATABLE DATA LISTS INTO
        LFILE. */
        CALL WRITE_RPL(LFILE,
        RELOCATABLE_HEAD_ARRAY,
        RELOCATABLE_BODY_AREA,BODY_SIZE,
        DATA_AREA, DATA_SIZE);
        /* PROCESS NEXT TWO INPUT CARDS. */
      GO TO
        INPUT;
OUTPUT:
        /* READ HEAD ARRAY, BODY AREA, AND
        DATA AREA FOR NEXT SET OF
        RELOCATABLE LISTS OF LISTS IN
        LFILE. */
        CALL READ_RPL(LFILE,
        RELOCATABLE_HEAD_ARRAY,
        RELOCATABLE_BODY_AREA, BODY_SIZE,
        DATA_AREA, DATA_SIZE);
        /* IF END OF LFILE  IS REACHED
        TERMINATE PROGRAM. */
   IF
        BODY_SIZE = 0
    THEN
      GO TO
        END_TRANS_L;
        /* EMPTY ABSOLUTE_BODY_AREA. */
        ABSOLUTE_BODY_AREA = EMPTY;
        /* CONVERT LISTS OF LISTS FROM
        RELOCATABLE TO ABSOLUTE FORM. */
        CALL CON_LRA(RELOCATABLE_BODY_AREA,
        RELOCATABLE_HEAD_ARRAY,
        ABSOLUTE_BODY_AREA,
        ABSOLUTE_HEAD_ARRAY,
        DATA_AREA, #SUBS);
        /* ASSIGN POINTER VALUES OF
        ABSOLUTE_HEAD_ARRAY TO HEAD
```

```
        POINTERS LIST1 AND LIST2 */
        LIST1 = ABSOLUTE_HEAD_ARRAY(1);
        LIST2 = ABSOLUTE_HEAD_ARRAY(2);
        /* PRINT ALL DATA VALUES OF LIST1 IN
        SUCCESSIVE POSITIONS ON ONE LINE AND
        THOSE OF LIST2 ON NEXT LINE. */
PRINT:
        DO WORK_POINTER = LIST1, LIST2;
        IF WORK_POINTER = NULL
        THEN GO TO END_PRINT;
        SUB1 = WORK_POINTER->VALUE;
        WORK_POINTER = WORK_POINTER->LINK;
        IF WORK_POINTER = NULL
        THEN DO;SUB2=NULL;GOTO FIRST;END;
        SUB2 = WORK_POINTER->VALUE;
        FIRST:
        WORK_POINTER = SUB1;
        IF WORK_POINTER = NULL
        THEN GO TO SECOND;
        DO WHILE(WORK_POINTER ¬= NULL);
        P = WORK_POINTER->VALUE;
        PUT EDIT(P->DATA_ITEM)(A);
        WORK_POINTER = WORK_POINTER->LINK;
        END;
        SECCND:
        WORK_POINTER = SUB2;
        IF WORK_POINTER = NULL
        THEN GO TO END_PRINT;
        DO WHILE(WORK_POINTER ¬= NULL);
        P = WORK_POINTER->VALUE;
        PUT EDIT(P->DATA_ITEM)(A);
        WORK_POINTER = WORK_POINTER->LINK;
        END;
     PUT
        SKIP;
END_PRINT:
END;
     PUT
        LIST('*****');
     PUT
        SKIP(2);
        /* PROCESS NEXT SET OF RELOCATABLE
        LISTS IN LFILE . */
      GO TO
        OUTPUT;
END_TRANS_L:
        CLOSE FILE(SYSPRINT);
        CLOSE FILE(LFILE);
END
        TRANS_L;
```

Figure 3.2A. The TRANS_L program, which illustrates the construction of relocatable lists of lists and transmission to and from a file

TRANS_L begins by allocating list components in the storage area ABSOLUTE_BODY_AREA and linking the components into an absolute list of available storage components called FREE. The program then uses the components in FREE to create two absolute lists of lists, LIST1 and LIST2, which contain two sublists each at the top level. Each sublist contains five data (D) components, as shown in Figure 3.2C.

```
CARD1 -----
CARD2 -----
CARD3 -----
CARD4 -----
CARD5 -----
CARD6 -----
CARD7 -----
CARD8 -----
CARD9 -----
CARD10  ----
```

Figure 3.2B. Sample input from SYSIN file

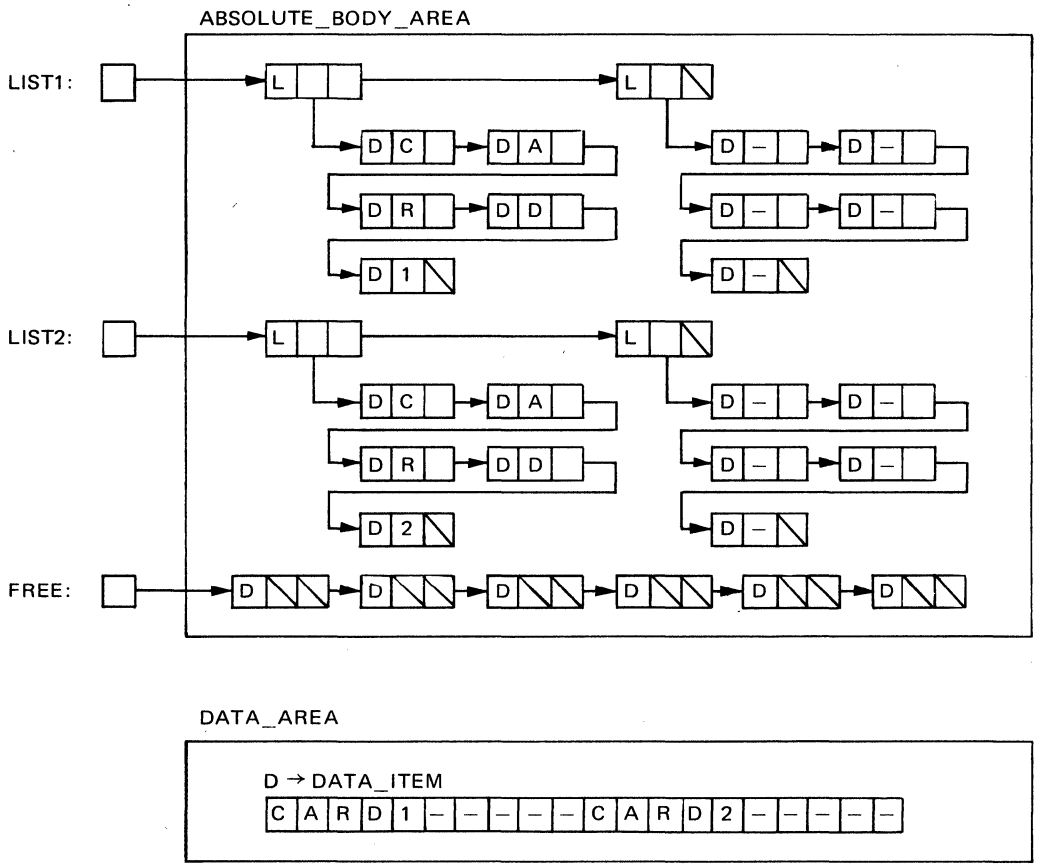


Figure 3.2C. Examples of absolute lists of lists

DATA_AREA serves as the storage area for the data values associated with LIST1 and LIST2. The based variable DATA_ITEM is allocated in DATA_AREA. DATA_ITEM specifies single characters whose addresses are assigned to the 20 value pointers in the data components of LIST1 and LIST2. Input is obtained from the standard system-input file, SYSIN, samples for which appear in Figure 3.2B. The characters in cc 1 through 10 of each two input cards are assigned to allocations of DATA_ITEM. Figure 3.2C illustrates the association between the data content of DATA_AREA and the lists of lists in ABSOLUTE_BODY_AREA. The diagram uses compact representation for LIST1 and LIST2 to avoid excessive usage of arrows.

TRANS_L now invokes the subroutine CON_LAR, which converts the absolute lists of lists in ABSOLUTE_BODY_AREA to relocatable lists in RELOCATABLE_BODY_AREA, as illustrated by Figure 3.2D. The array RELOCATABLE_HEAD_ARRAY contains the offset heads of the relocatable lists. At this point, subroutine WRITE_RPL writes RELOCATABLE_HEAD_ARRAY, RELOCATABLE_BODY_AREA, and DATA_AREA as separate logical records into the file LFILE (see Figure 3.2E).

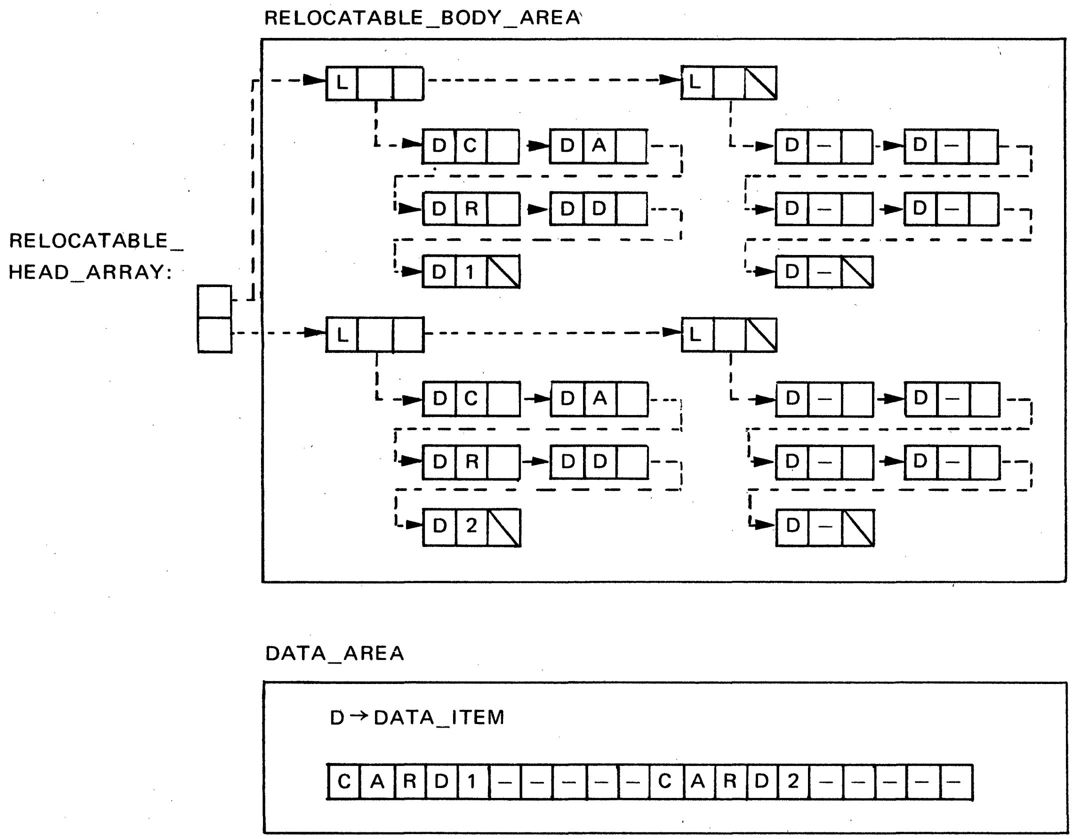


Figure 3.2D. Examples of relocatable lists of lists

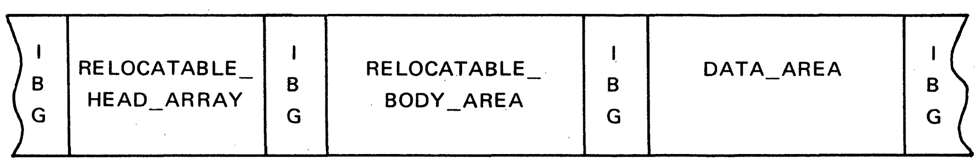


Figure 3.2E. Sample content of LFILE (unblocked)

TRANS_L contrinues processing pairs of input cards and generating relocatable output for LFILE. When the end of the SYSIN file is reached, LFILE is closed and reopened as an input file. The previous processing steps are now reversed. Subroutine READ_RPL retrieves each head array, body area, and data area from LFILE, and subroutine CON _LRA converts the retrieved lists from relocatable to absolute form. The data values of absolute list LIST1 are then printed in successive positions on a line in the standard system-output file, SYSPRINT. Similarly, the data values of LIST2 appear on the next line, as shown in Figure 3.2F. Each pair of output lines is followed by a line of five asterisks and a blank line. The program terminates when the end of LFILE is reached.

```
CARD1 -----
CARD2 -----
*****

CARD3 -----
CARD4 -----
*****

CARD5 -----
CARD6 -----
*****

CARD7 -----
CARD8 -----
*****

CARD9 -----
CARD10  ----
*****
```

Figure 3.2F. Sample output to SYSPRINT file

SUMMARY

This manual shows how to form a relocatable list by using offset variables rather than pointer variables as component links in the list. The values of the offset variables remain valid when the list is moved to a new location within internal storage or transmitted to and from a file.

A relocatable list can be treated as a collective unit by referring to the area in which the components of the list have been allocated and linked. Internal and external movement of the relocatable list is then achieved by transmitting the containing area.

The techniques are summarized below:

1. A list can be treated as a collective unit by referring to the area in which the list components have been allocated. Internal and external movement of a list then becomes possible by transmitting the containing area.
2. The assignment statement permits the contents of one area to be assigned to another area. However, pointer values in the assigned area become invalid in the receiving area.
3. No operators can be applied to area variables.
4. An area is made empty by assigning it the value of the builit-in function EMPTY or the value of another empty area.
5. Assignment of an area effectively frees all allocations in the receiving area and then assigns the content of the area to the receiving area.
6. All free-storage gaps are retained within an assigned area, so that allocations within the assigned area maintain their locations relative to each other.
7. When the source area is smaller than the receiving area, the assigned area is effectively extended with free storage. Similarly, when the source area is larger than the receiving area, truncation of free storage occurs at the end of the assigned area. However, if the truncation involves allocated storage and not just free storage, the AREA ON-condition occurs, and the contents of the receiving area become undefined.
8. A relocatable list is formed by using offset variables rather than pointer variables as component links in the list.
9. An offset variable has a relative address as its value and is declared with the OFFSET attribute, which has the following form:

   OFFSET(area-variable)

   The area variable in parentheses must be based and unsubscripted and must have an implied or explicit level number of one.
10. When the value of a pointer variable is assigned to an offset variable, the assigned pointer value is automatically adjusted so that it becomes relative to the beginning of the area associated with the receiving offset variable. The address computation is equivalent in effect to the following calculation:

    Offset value = (Pointer value) – (Absolute address of area)
11. When an offset value is assigned to a pointer variable, the offset value is automatically added to the absolute address of the area specified in the associated OFFSET attribute; the result becomes the value of the receiving pointer:

    Pointer value = (Offset value) + (Absolute address of area)
12. Assignment of an offset value to an offset variable is performed without address modification.
13. The programmer cannot apply explicit arithmetic operations to offset variables in the source program; however, comparisons of offset variables can be made with the operators equal (=) and not equal (¬=).
14. A null offset value is assigned to an offset variable through the built-in function NULLO.
15. A null offset value cannot be assigned to a pointer variable. Similarly, a null pointer value cannot be assigned to an offset variable.
16. An offset variable cannot qualify a based variable. The offset value must first be assigned to a pointer variable, which is then used to qualify the based variable.
17. The values of locator variables (offsets and pointers) cannot be converted to any other type of data, nor can any other type of data be converted to locator type.
18. Locator variables may be used as arguments and parameters. When an offset argument is associated with an offset parameter, both must be offset with respect to the same area.
19. Only record-oriented input and output statements can be used to transmit relocatable lists. The LOCATE statement is used to transmit lists to a file, and the READ statement is used to retrieve lists from a file.
20. The subroutines developed in this manual for processing relocatable lists fall into five categories:
    a. Converting absolute lists to relocatable form
    b. Converting relocatable lists to absolute form
    c. Moving relocatable lists
    d. Writing relocatable lists
    e. Reading relocatable lists

APPENDIX

The Recursive Function Procedure CONV

```
/* FUNCTION PROCEDURE CONV
   CAN BE USED WITH CON_LAR */
/* DECLARE CONV ENTRY
   (POINTER, AREA(*), AREA(*), AREA(*))
   RETURNS(OFFSET(DUMMY_BODY_AREA)),
   DUMMY_BODY_AREA AREA BASED
   (DUMMY_POINTER),
   DUMMY_POINTER POINTER; */
CONV:
      PROCEDURE(LIST,
         BODY_AREA1,BODY_AREA2,DATA_AREA)
         RETURNS(OFFSET(DUMMY_BODY_AREA))
         RECURSIVE;
         /* CONV IS A RECURSIVE FUNCTION
         PROCEDURE THAT CONVERTS A LIST OF
         LISTS IN BODY_AREA1 TO A
         RELOCATABLE LISTS OF LISTS IN
         BODY_AREA2. THE HEAD POINTER OF THE
         LIST TO BE CONVERTED IS PASSED TO
         CONV AS AN ARGUMENT. THE FUNCTION
         RETURNS THE OFFSET ADDRESS OF THE
         NEW LIST IN BODY_AREA2. */
 DECLARE
         LIST POINTER,
         O OFFSET(DUMMY_BODY_AREA),
         (DUMMY_BODY_POINTER,
         DUMMY_DATA_POINTER,C1,C2)POINTER,
         (BODY_AREA1,BODY_AREA2,DATA_AREA)
         AREA(*),
         DUMMY_BODY_AREA
         BASED(DUMMY_BODY_POINTER) AREA,
         DUMMY_DATA_AREA
         BASED(DUMMY_DATA_POINTER) AREA,
         1 COMPONENT1 BASED(C1),
         2 TYPE CHARACTER(1),
         2 VALUE POINTER,
         2 LINK POINTER,
         1 D_COMPONENT2 BASED(C2),
         2 D_OTYPE CHARACTER(1),
         2 D_OVALUE OFFSET(DUMMY_DATA_AREA),
         2 D_OLINK OFFSET(DUMMY_BODY_AREA),
         1 L_COMPONENT2 BASED(C2),
         2 L_OTYPE CHARACTER(1),
         2 L_OVALUE OFFSET(DUMMY_BODY_AREA),
         2 L_OLINK OFFSET(DUMMY_BODY_AREA);
   IF
         LIST = NULL
    THEN
         RETURN(NULLO);
         DUMMY_BODY_POINTER=ADDR(BODY_AREA2);
         DUMMY_DATA_POINTER=ADDR(DATA_AREA);
         C1 = LIST;
   IF
         C1->TYPE = 'D'
    THEN
  DO;
         ALLOCATE D_COMPONENT2 IN(BODY_AREA2)
         SET(C2);
         C2->D_OTYPE = 'D';
   IF
         C1->VALUE = NULL
    THEN
         C2->D_OVALUE = NULLO;
    ELSE
         C2->D_OVALUE = C1->VALUE;
         C2->D_OLINK=CONV(C1->LINK,
         BODY_AREA1,BODY_AREA2,DATA_AREA);
END;
    ELSE
  DO;
         ALLOCATE L_COMPONENT2 IN(BODY_AREA2)
         SET(C2);
         C2->L_OTYPE = 'L';
         C2->L_OVALUE=CONV(C1->VALUE,
         BODY_AREA1, BODY_AREA2, DATA_AREA);
         C2->L_OLINK=CONV(C1->LINK,
         BODY_AREA1, BODY_AREA2, DATA_AREA);
END;
         O = C2;
         RETURN(O);
END
         CONV;
```

The Recursive Function Procedure CON

```
/* FUNCTION PROCEDURE CON
   CAN BE USED WITH CON_LRA */
/* DECLARE CON ENTRY
   (OFFSET(DUMMY_BODY_AREA), AREA(*),
   AREA(*), AREA(*))RETURNS(POINTER),
   DUMMY_BODY_AREA AREA BASED
   (DUMMY_PCINTER),
   DUMMY_POINTER POINTER; */
CON:
    PROCEDURE(RLIST,
        BODY_AREA1,BODY_AREA2,DATA_AREA)
        RETURNS(POINTER)RECURSIVE;
        /* CON IS A RECURSIVE FUNCTION
        PROCEDURE THAT CONVERTS A
        RELOCATABLE LIST OF LISTS IN
        BODY_AREA1 TO AN ABSOLUTE LIST OF
        LISTS IN BODY_AREA2. THE OFFSET HEAD
        OF THE LIST TO BE CONVERTED IS
        PASSED TO CON AS AN ARGUMENT. THE
        FUNCTION RETURNS THE ABSOLUTE ADDRESS
        OF THE NEW LIST IN BODY_AREA2. */
 DECLARE
        RLIST OFFSET(DUMMY_BODY_AREA),
        (BODY_AREA1,BODY_AREA2,DATA_AREA)
        AREA(*),
        DUMMY_BODY_AREA
        BASED(DUMMY_BODY_POINTER) AREA,
        DUMMY_DATA_AREA
        BASED(DUMMY_DATA_POINTER) AREA,
        (DUMMY_BODY_POINTER,
        DUMMY_DATA_POINTER,C1,C2)POINTER,
        1 D_COMPONENT1 BASED(C1),
        2 D_OTYPE CHARACTER(1),
        2 D_OVALUE OFFSET(DUMMY_DATA_AREA),
        2 D_OLINK OFFSET(DUMMY_BODY_AREA),
        1 L_COMPONENT1 BASED(C1),
        2 L_OTYPE CHARACTER(1),
        2 L_OVALUE OFFSET(DUMMY_BODY_AREA),
        2 L_OLINK OFFSET(DUMMY_BODY_AREA),
        1 COMPONENT2 BASED(C2),
        2 TYPE CHARACTER(1),
        2 VALUE POINTER,
        2 LINK POINTER;
   IF
        RLIST = NULLO
    THEN
        RETURN(NULL);
        DUMMY_BODY_POINTER = ADDR(BODY_AREA1);
        DUMMY_DATA_POINTER=ADDR(DATA_AREA);
        C1 = RLIST;
   IF
        C1->D_OTYPE = 'D'
    THEN
  DO;
        ALLOCATE COMPONENT2
        IN (BODY_AREA2) SET(C2);
        C2->TYPE = 'D';
   IF
        C1->D_OVALUE = NULLO
    THEN
        C2->VALUE = NULL;
    ELSE
        C2->VALUE = C1->D_OVALUE;
        C2->LINK=CON(C1->D_OLINK,
        BODY_AREA1,BODY_AREA2,DATA_AREA);
END;
    ELSE
  DO;
        ALLOCATE COMPONENT2
        IN (BODY_AREA2) SET (C2);
        C2->TYPE = 'L';
        C2->VALUE=CON(C1->L_OVALUE,
        BODY_AREA1,BODY_AREA2,DATA_AREA);
        C2->LINK=CON(C1->L_OLINK,
        BODY_AREA1,BODY_AREA2,DATA_AREA);
 END;
        RETURN (C2);
END
        CON;
```

Index

GF20-0020-0

Techniques for Processing Relocatable Lists in PL/I Printed in U.S.A. GF20-0020-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

READER'S COMMENT FORM

Techniques for Processing Relocatable Lists in PL/I

GF20-0020-0

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM. If you wish a reply, be sure to include your name and address.

COMMENTS

fold

fold

fold

fold

• Thank you for your cooperation. No postage necessary if mailed in the U.S.A.
FOLD ON TWO LINES, STAPLE AND MAIL.

GF20-0020-0

YOUR COMMENTS PLEASE...

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

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

fold fold

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

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

POSTAGE WILL BE PAID BY...

IBM Corporation
1133 Westchester Avenue
White Plains, N.Y. 10604

Attention: Technical Publications

fold fold

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

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